Secure MMU: Architectural Support for Memory Isolation among Virtual Machines

Seongwook Jin and Jaehyuk Huh
Computer Science, KAIST (Korea Adanced Institute of Science and Technology)

***Abstract*—In conventional virtualized systems, a hypervisor can access the memory pages of guest virtual machines without any restriction, as the hypervisor has a full control over the address translation mechanism. In this paper, we propose *Secure MMU*, a hardware-based mechanism to isolate the memory of guest virtual machines from unauthorized accesses even from the hypervisor. The proposed mechanism extends the current nested paging support for virtualization with a small hardware cost. With Secure MMU, the hypervisor can flexibly allocate physical memory pages to virtual machines for resource management, but update nested page tables only through the secure hardware mechanism, which verifies each mapping change. With the hardware-rooted memory isolation among virtual machines, the memory of a virtual machine in cloud computing can be securely protected from a compromised hypervisor or co-tenant virtual machines.**

I. Introduction

Cloud computing has emerged as a promising future computing model, which enables elastic resource provisioning for unpredictable demands [1], [2]. In cloud computing, users run their programs on 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 [3], [4].

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. There have been concerns over vulnerabilities in hypervisors [5], [6], and several recent studies address the problem by improving the security of hypervisors [7], [8], [9]. However, hypervisors, which have been increasing their code size for better performance and more features, have become too complex to simply verify their secure execution [9]. 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 VMs. For such memory isolation, current virtualization techniques are based on traditional software and hardware support for virtual memory. A hypervisor maps the memory of guest VMs to the physical memory with guest-physical to machine address translation. Although several different techniques are used to support efficient address translation to machine addresses, all these techniques rely on hypervisors to prevent any illegal memory access across virtual machines [10], [11], [12]. In the current memory virtualization techniques, hypervisors, at the highest privilege level, can read or modify the memory allocated for guest VMs.

The main cause of the privacy concern for guest VMs is that hypervisors control the memory protection mechanism without any restriction. In the current support for virtual memory management, two aspects, *the memory resource management with paging*, and *the memory isolation through address translation*, are under the control of hypervisors. 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. Whenever a VM is scheduled to a core, the hypervisor sets the register pointing to the nested page table for the VM, so that the hardware TLB walker can access the appropriate mapping table. One crucial disadvantage of the traditional approach is that a successful attack on the hypervisor can completely expose the memory of guest virtual machines to the attacker.

In this paper, we propose the decoupling of memory isolation from memory resource management, both of which are currently performed by the hypervisor. With the decoupled mechanism, called *Secure MMU*, the role of a hypervisor is limited to the resource management to allocate limited physical memory to guest VMs to improve system throughput and fairness. Hardware processors are now responsible for updating the page mapping and setting the pointer to the nested page table between VM switches. Guest virtual machines can directly communicate the hardware controller to validate if the memory pages are accessible neither by the hypervisor, nor by the other virtual machines.

There have been several prior studies to propose mechanisms to protect memory across processes or virtual machines. AEGIS uses encryption techniques to protect any data leaving a processor [13]. Any data fetched from the memory into on-chip caches must be decrypted and evicted data from on-chip caches must be encrypted before leaving the processor. NoHype suggests partitioning memory pages for each core, and a VM is running on a fixed core [14]. Those techniques require complete re-designs of HW architectures and system software. Unlike prior techniques, this paper takes an evolutionary path to minimize the changes in hardware and software systems. In the proposed mechanism, flexible memory allocation and core scheduling are still supported with the same mechanism as the conventional systems.

In this paper, we discuss that the hardware-rooted memory protection is feasible with a small change in the current microarchitectures and hypervisors. Only the hardware controller can change the nested page tables to allocate or deallocate physical memory for virtual machines. A hypervisor can decide which memory pages are allocated for a VM, but they cannot directly update nested page tables. With this separation, a compromised hypervisor cannot access the physical memory already allocated for a guest VM. In this paper, we show that such hardware mechanism can be implemented with a modest complexity increase from the current architectural support for virtualization.

II. Background

A. Memory Isolation among VMs

Memory isolation in current virtualization techniques is based on the support for virtual memory with hardware address translation and page tables. In processors, virtual addresses are translated to physical addresses 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 the HW processor. For each context switch between address spaces, the page table register, which points the top-level page table entry, must be properly set, so that the walker can traverse the correct page table. In 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) just like non-virtualized systems, and, in addition, the guest-physical address is translated to a machine address in 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 native operating systems, the hypervisor maintains a direct translation table, called shadow page tables, to map virtual addresses to machine addresses. The recent advancement of architectural support for virtualization allows a two-level hardware page table walker to traverse both per-process page tables and per-VM nested page tables [11]. Figure 1 depicts the two-level address translation with a per-process guest page table and a per-VM nested page table. Each core has two registers pointing each table, one for the guest page table (`gCR3`), and the other for the nested page table (`nCR3`).

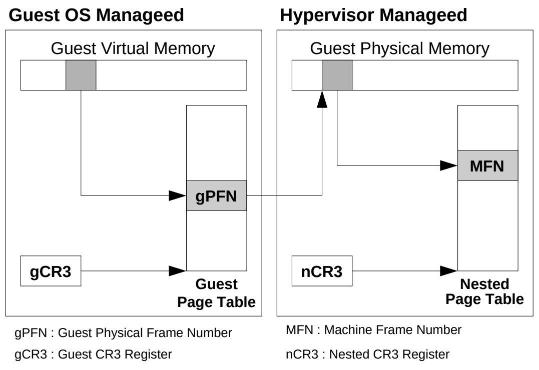

Fig. 1. Memory Translation with Nested Paging

To isolate the memory of each VM, a hypervisor must protect nested page tables from illegal modification 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. The hypervisor manages the 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 same 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.

B. Requirements for Secure MMU

To guarantee the isolation among VMs even with a compromised hypervisor, hypervisors must not be able to change nested page tables arbitrarily. To enable memory isolation even with a compromised hypervisor, Secure MMU modifies nested page tables with a protected hardware controller. Such HW-rooted memory protection must meet the following requirements:

- Secure MMU must still allow the hypervisor 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.
- Secure MMU must guarantee that the content of the deallocated pages are cleared before it can be accessed by the hypervisor or other guest VMs. Furthermore, hypervisors may evict some memory pages of guest VMs to the swap disk. During the page swap-out, the content of memory pages must not be readable.
- Physical pages can be shared among VMs or between a VM and the hypervisor. A VM and the hypervisor access shared pages to communicate data for I/O operations. Such page sharing must still be supported, but participating VMs must agree to the sharing of certain pages.
- Virtual machines can be suspended, creating the checkpoint of their states, and later resumed. Furthermore, VMs may migrate to different systems. Secure MMU must provide mechanisms to checkpoint the memory content securely, without exposing the memory content.

C. Threat Model

The goal of the proposed decoupling of memory allocation and isolation is to protect the memory of virtual machines from a malicious co-tenant sharing the same physical system. The trusted computing base (TCB) to protect the memory of guest VMs in the proposed system includes only the hardware system, not the hypervisor. As the hardware system must be included in the TCB, we do not consider any hardware attack, such as probing of external buses [15] or reading DRAM after power-off [16]. We assume 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 legal obligation not to access the costumer's data without explicit permission, but also has a strong business interest in protecting their reputation.

However, a hypervisor is not included in the TCB. We assume that hypervisors are vulnerable from attacks by malicious guest virtual machines. The threat model assumes that an attacker 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 person cannot compromise the physical servers directly. With proper security measures on server rooms, getting accesses to the server room, and physically compromising the systems is much harder than getting the root permission by remote attacks.

The memory protection mechanism in this paper

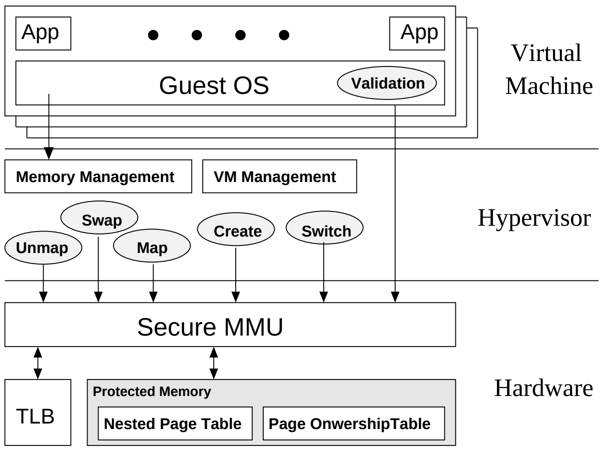

Fig. 2. Secure MMU Architecture

address the isolation among virtual machines or from the hypervisor. It does not improve the security of guest operating systems and applications by themselves.

III. Architecture

A. Overview

Secure MMU requires only a small change in the existing architecture. In the proposed architecture, hypervisors cannot directly change per-VM nested page tables. Only the dedicated hardware controller (Secure MMU), which extends a conventional HW page walk controller, can update the nested page tables. Figure 2 presents the overall architecture of Secure MMU. Nested page tables of all VMs are stored in protected memory pages, which are accessible only by Secure MMU. The protected memory region is just part of the physical memory. The hardware controller does not allow any accesses even by the hypervisor to the protected memory region.

Secure MMU provides two important interfaces to the hypervisor and guest VMs. Firstly, the hypervisor, in the privileged mode, can request to map or unmap a physical memory page to a virtual-physical page of a VM. Since the hypervisor does not have access permission on the nested page table, Secure MMU must update the nested page table to add or delete the entry. Secondly, a guest VM can request to validate whether a page is accessible only by itself. A page can be declared as a *VM-private* or *shared* page. Before using a memory page for the first time, the operating system of the guest VM can make validation requests to Secure MMU, without the intervention of the hypervisor by executing an untrappable instruction.

Secure MMU protects the nested page table area by allowing access permission only to itself. Guest VMs uses memory pages after validating their sharing mode and write permission. After the validation, Secure MMU does not allow a physical page to be shared, even if the privileged hypervisor requests such change. To change a VM-private page to a shared page, the owner VM must explicitly request the status change to Secure MMU. To check the sharing status of physical pages, Secure MMU tracks the ownership status for each physical page. For fast checking, Secure MMU maintains the current owning VM for each physical memory page in *Page Ownership Table* shown in Figure 2. The page ownership information is stored in the protected memory region.

In Secure MMU, the hypervisor conducts normal memory management functions for all VMs. It determines the allocation and deallocation of memory pages for VMs by checking usage patterns. For memory management, the hypervisor can maintain its own copy of nested page tables just to keep track of page usages. An alternative way is to allow the hypervisor to read the original nested page tables managed by Secure MMU, although updating the tables are not permitted to the hypervisor. The hypervisor can read the nested page tables only with a read-only permission. The rest of the protected memory area must not be accessible by the hypervisor.

For the switch between VMs, Secure MMU is responsible for setting the address of the nested page table (`nCR3`). The hypervisor makes a request to Secure MMU by executing a privileged instruction to place a VM to a physical core. Furthermore, to protect the content of registers of a VM, Secure MMU may save registers and clear them for exceptions and interrupts, before transferring the control to the hypervisor.

B. Basic Functions

In this section, we describe five basic functions, which must be included in Secure MMU. Current HW page table walkers with virtualization support can traverse guest and nested page tables. In addition, Secure MMU must maintain nested page tables for VMs, and the page ownership table to trace the owner of each physical memory pages. Such data must be stored in the protected region of DRAM. Secure MMU must block any attempt to access the protected region by declining page mapping requests to the protected pages. To maintain nested page tables and check illegal attempts to access guest memory, Secure MMU must support the following basic functions:

VM creation: When a hypervisor creates a VM, it requests Secure MMU to create a new nested page table for the VM. Secure MMU must know the identity of each VM running on the system, and maintain the nested page table. Also, Secure MMU creates a per-VM encryption key, which will be used for page swapping.

Page map: To assign a physical memory page to a VM, the hypervisor should request a *page map* operation to Secure MMU. A page map operation maps a machine memory page to a guest-physical page by updating a nested page table entry. When a nested page table entry is updated for a VM by the request of the hypervisor, the page is set to be VM-private. Before updating the nested page table entry, Secure MMU 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. This checking mechanism prevents a compromised hypervisor from creating any illegal mapping to VM-private pages.

Page unmap: To deallocate a physical memory page from a VM, the hypervisor makes a *page unmap* request to Secure MMU. Secure MMU modifies the corresponding nested page table entry, and clears the content of the memory page before completing the operation. Secure MMU also resets the owner of the page in the page ownership table. The hypervisor may make a page map request for the unmapped page later, but its content has already been cleared. With clearing, the memory content of free pages cannot contain any information from guest VMs.

VM switch: When a hypervisor needs to schedule a VM to a core, the hypervisor makes a *VM switch* request to Secure MMU. The hypervisor cannot directly update the page table pointer to the nested page table (nCR3). The hypervisor makes a VM switch request with a VM identifier, and Secure MMU sets up nCR3 with the corresponding address of the top-level page table. The reason for disabling direct changes of nCR3 by the hypervisor is that the hypervisor can create a compromised copy of a nested page table for a VM, and make a VM use the compromised nested page table. As only Secure MMU can update the nested page table pointer, the hypervisor cannot make a running VM to use a compromised nested page table.

This support for Secure VM switch can be extended to protect register contents. Although protecting the register content is beyond the scope of this paper, Secure MMU can include the mechanism to save and restore the VM state in processor registers for interrupts and exceptions. Before transferring the control to the hypervisor after an interrupt occurs, the register contents unrelated to the interrupt are saved and cleared. With the support, the hypervisor cannot directly read the register contents of guest VMs.

Validation: When a VM is created, the guest OS can request Secure MMU to validate the state of its nested page table for each page. Such request is made by executing an untrappable instruction, so the hypervisor cannot intervene. Upon the request, Secure MMU checks whether the requested page is VM-private or shared.

C. Advanced Functions

More complex operations on the memory allocated for virtual machines may be necessary for flexible management of VMs. These advanced functions are necessary to support page swapping, VM checkpointing and migrations.

Page swap: In virtualized systems, two-level paging techniques are used to allocate the physical memory efficiently. Firstly, when a system is running out of the physical memory, the hypervisor enforces each VM to swap out some unused memory pages to the swap disk of the VM. As the guest OS on each VM can make much better decisions on selecting victim pages than the hypervisor, such *ballooning* allows the guest OS to pick victim pages, and to reduce its memory usage [17]. If a guest OS decides to evict a memory page to its swap disk, the guest OS is responsible for encrypting the content. As the memory of the guest VM is protected, the encryption key is securely protected.

In addition to ballooning, the hypervisor also needs to evict memory pages to its disk swap space directly. However, the direct page swaps by hypervisors are less commonly used than the ballooning technique [17]. As the hypervisor writes the memory pages to the swap disk, the content of evicted memory pages must be encrypted by Secure MMU before being accessible by the hypervisor. As the hypervisor must not be able to read the pages of guest VMs, Secure MMU must include a encryption component for swapped memory pages. When a VM is created, Secure MMU creates a encryption key for the VM. When the hypervisor picks a victim page, Secure MMU 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.

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 [18], [19]. By default, Secure MMU allows a physical memory page to be mapped only to a single VM. However, to makes the page sharable, Secure MMU 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.

VM checkpointing: Virtual machines can be suspended and resumed by using checkpoints. However, to create a checkpoint, the hypervisor must read the memory contents of a VM. For checkpointing, Secure MMU can use the same encryption mechanism as the one used to support memory page swap by the hypervisor. To create a checkpoint, the hypervisor requests encrypted memory pages from Secure MMU. Secure MMU uses a per-VM encryption key to encrypt or decrypt the memory pages.

VM migration: VM migration is more complicated than VM checkpointing, as it involves encryption key management between two physical systems. To migrate a VM to a different system, encrypted memory contents must be transferred through the hypervisor and networks. We assume that each system has a private-public key pair integrated in each processor chip. The private key never leaves the hardware processor. A similar technique is used for TPM (trusted platform module) [20]. The public key is registered in the managing system. To migrate the memory content of a VM, Secure MMU encrypts the pages with a symmetric key, and the symmetric key is encrypted with the public key of the managing system. The managing system can forward the key to another system, where the VM migrates to, after decrypting the key and encrypting the key with the public key of the target system. The managing system, which runs in a separate physical system from guest VMs, must be protected and included in TCB.

Identifying Secure MMU: Guest OSes must be able to verify whether the physical system is using Secure MMU, as the hypervisor may emulate compromised Secure MMU functions. Such identification uses the private key integrated in each processor chip, as discussed in the VM migration mechanism. The managing system, which is in TCB, has the public keys of the physical processors maintained by the managing system. Guest OSes can validate the existence of secure MMU by getting the public key of the processor from the managing system, and by checking the signature signed with the processor private key. The managing system is a separate entity dedicated to cloud system management, and thus it is less vulnerable to outside attacks.

IV. Implementation Costs

Secure MMU extends the current architectural support for nested address translation for virtualization, incurring a modest increase of design complexity.

Hardware costs: Hardware costs for Secure MMU comes from the increased complexity of hardware controller compared to the conventional MMU. Unlike the conventional MMU, Secure MMU must update nested page table entries as requested by the hypervisor for mapping and unmapping. For basic functions, we expect the additional complexity in the controller is small. The page map operation requires an extra step to check whether the newly mapped page is used by other VMs. The unmap operation must clear the page contents. Those operations require a modest increase of the controller complexity. As Secure MMU uses a part of physical memory to store nested page tables, the page ownership table, and per-VM information such as the encryption key, Secure MMU does not require any extra on-chip storage.

Supporting advanced functions require more complex operations in the controller. However, much of the increased complexity is for performing page encryption. An alternative implementation for the advanced functions for Secure MMU is to use secure execution modes in processors, such as ARM TrustZone [21]. A complex operation of supporting advanced functions may be executed in the normal computing core, but in the secure execution mode.

Modification to hypervisors and guest OSes: We believe the modification to hypervisors is small, as the nested page modification routines are just moved to Secure MMU from hypervisors. Instead of modifying nested page tables directly, the hypervisor needs to make requests to Secure MMU. By defaults, guest OSes do not need to be modified, if a VM always uses VM-private pages with guest-level page swapping disabled. To support page sharing, guest OSes must be slightly modified to add features to accept page sharing requests. To enable guest-level swapping, guest OSes must encrypt pages before sending them to guest swap spaces.

Performance: The performance overheads of Secure MMU occur during page map and unmap operations. During a page map operation, Secure MMU looks up the page ownership table, which may need several memory lookups. However, a page allocation operation is an 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. For page swapping, encryption is necessary, but compared to long disk access latencies, extra computation latencies for encryption will be minor.

V. Conclusions

In this paper, we proposed a hardware-based mechanism called Secure MMU to isolate VM memory securely even under a compromised hypervisor. We believe the costs of extra hardware are modest with a small increase of MMU complexity. As future work, we will implement Secure MMU on a hardware simulator and also open source processor designs with FPGA.

Acknowledgments

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 (2010-0025511). This work was also partly supported by the IT R&D Program of MKE/ KEIT (2011-KI002090, Development of Technology Base for Trustworthy Computing).

References

[1] Amazon Elastic Compute Cloud (Amazon EC2), “http://aws.amazon.com/ec2,” 2008.

[2] Windows Azure Platform, “http://www.microsoft.com/windowsazure/,” 2010.

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

[4] Security Is Chief Obstacle To Cloud Computing Adoption, “http://www.darkreading.com/securityservices/security/perimeter/showAr%ticle.jhtml?articleID=221901195,” 2009.

[5] Secunia Vulnerability Report: Xen 3.x, “http://secunia.com/advisories/product/15863/,” 2010.

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

[7] 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*, pp. 151–160.

[8] 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*, pp. 380–395.

[9] 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*, pp. 38–49.

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

[11] Advanced Micro Dvices, “AMD-V Nested Paging,” 2008.

[12] G. Neiger, A. Santoni, F. Leung, D. Rodger, and R. Uhlig, “Intel Virtualization Technology: Hardware Support for Effcient Processor Virtualization,” *Intel Technology Journal*, vol. 10, no. 03, pp. 167–178, 2006.

[13] 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*, pp. 160–171.

[14] 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*, pp. 350–361.

[15] R. Anderson and M. Kuhn, “Low Cost Attacks on Tamper Resistant Devices,” in *Security Protocols: 5th International Workshop, LNCS*, 1997, pp. 125–136.

[16] 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*, vol. 52, pp. 91–98, May 2009.

[17] C. A. Waldspurger, “Memory Resource Management in VMware ESX Server,” in *Proceedings of the 5th symposium on Operating systems design and implementation, OSDI’02*. New York, NY, USA: ACM, 2002, pp. 181–194.

[18] 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.

[19] 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*, 2008, pp. 309–322.

[20] Trusted Platform Module, “http://www.trustedcomputinggroup.org/developers/trusted_platform_modu%le.”

[21] T. Alves and D. Felton, “TrustZone: Integrated hardware and software security,” ARM, Tech. Rep., Jul. 2004.