Rethinking Memory Permissions for Protection Against Cross-Layer Attacks

JESSE ELWELL, SUNY Binghamton
RYAN RILEY, Qatar University
NAEL ABU-GHAZALEH, University of California, Riverside
DMITRY PONOMAREV, SUNY Binghamton
and ILIANO CERVESATO, Carnegie Mellon University

The inclusive permissions structure (e.g., the Intel ring model) of modern commodity CPUs provides privileged system software layers with arbitrary permissions to access and modify client processes, allowing them to manage these clients and the system resources efficiently. Unfortunately, these inclusive permissions allow a compromised high-privileged software layers to perform arbitrary malicious activities. In this paper, our goal is to prevent attacks that cross system layers while maintaining the abilities of system software to manage the system and allocate resources. In particular, we present a hardware-supported page permission framework for physical pages that is based on the concept of non-inclusive sets of memory permissions for different layers of system software (such as hypervisors, operating systems, and user-level applications). Instead of viewing privilege levels as an ordered hierarchy with each successive level being more privileged, we view them as distinct levels each with its own set of permissions. In order to enable system software to manage client processes, we define a set of legal permission transitions that support resource allocation but preserve security. We show that the model prevents a range of recent attacks. We also show that it can be implemented with negligible performance overhead (both at load time and at run time), low hardware complexity and minimal changes to the commodity OS and hypervisor code.

Categories and Subject Descriptors: C.1.0 [**Processor Architectures**]: General; D.4.6 [**Operating Systems**]: Security and Protection—*Access Controls*

General Terms: Architecure, Security

Additional Key Words and Phrases: Architecture, Security, System Software

ACM Reference Format:
Jesse Elwell, Ryan Riley, Nael Abu-Ghazaleh, Dmitry Ponomarev and Iliano Cervesato. 2015. Rethinking Memory Permissions for Protection Against Cross-Layer Attacks *ACM Trans. Architec. Code Optim.* V, N, Article A (January YYYY), 25 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Modern computing systems employ increasingly complex multi-layer system software stacks such as operating system kernels and hypervisors. As the complexity and the number of lines of code in these layers continues to increase, so does the number of security vulnerabilities that can be exploited by attackers. For example, hypervisors are growing to be large pieces of code with a large attack surface — Xen 4.0 has over 190 thousand lines of code. As of May 2015, according to [CVE Details 2015] 127 vulnerabilities have been identified in Xen and 4 have been identified in the KVM code. More alarmingly, about half of these vulnerabilities can lead to security breaches in terms of

Acknowledgments
This paper was made possible by NPRP grant 4-1593-1-260 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors.

confidentiality, integrity and availability of the OS/hypervisor. Exacerbating the problem is the monolithic nature of operating system (OS) kernels and hypervisors, where a single exploit can compromise the entire system.

Current processor architectures and system software layers are centered around inclusive memory permissions, where software running at a higher-privilege layer has unconstrained access to the code and data of the lower-privilege layers it manages. For example, the operating system has complete access to user-level pages, and the hypervisor has access to both OS and user-level pages, allowing full compromise from a single exploit.

We call the attacks that cross privilege layers *cross-layer attacks* (Figure 1). Attacks from a less privileged to a more privileged layer (arrows 1 and 2 in the figure) require an exploit of a vulnerability to achieve privilege escalation. In some cases, such as *ret-2-user* attacks [Kemerlis et al. 2012], a process obtains OS-level privileges by leveraging the fact that the OS can execute code from pages assigned to processes. Attacks from a more to a less privileged layer (arrows 3, 4, and 5 in the figure) do not require a software vulnerability. Once a layer is compromised due to inclusive permissions, nothing prevents it from extracting secrets or tampering with operation of less privileged layers under its control. As a result, attacks have been demonstrated where a malicious hypervisor attacks a guest OS [Szefer and Lee 2012] or an OS attacks user-level processes.

To defend from cross-layer attacks, we propose NIMP (Non-Inclusive Memory Permissions) — a hardware-software framework that assigns each privilege layer only the minimum set of permissions necessary to carry out its tasks; NIMP does not implicitly grant memory access to any privilege layer. Physical memory pages are assigned distinct permissions for each privilege layer, preventing privileged layers from having arbitrary access to less privileged layers. At the same time, to enable supervisor layers to manage their clients, a hardware-based Memory Permission Manager (MPM) follows a set of static rules to ensure that requests to change permissions will not allow code in other layers to compromise the confidentiality or integrity of memory pages. With these mechanisms, NIMP simultaneously allows supervisor layers to manage their clients, while preventing arbitrary access and closing avenues for cross-layer attacks.

The performance overhead of NIMP arises from two factors: 1) the need to access permission bits on every TLB miss, in the worst case resulting in an additional memory access; and 2) the need to wipe out the contents of some pages in hardware, as dictated by the permission modification rules. However, we demonstrate that most of the time permission bits can be found in the CPU caches thus avoiding the extra memory access. Furthermore, as the operations requiring the page permission bits to be changed during program execution are relatively infrequent, the page wiping overhead is also small. Consequently, NIMP has a low performance overhead — 2% on the SPEC 2006 benchmarks. In terms of hardware overhead, NIMP requires about 28 bytes of on-chip Ternary Content-Addressable Memory (TCAM) storage for the permission rules, about 9% increase in the area of all TLBs, a new register to hold the expected permission bits, and an additional 64-bit register to point to the starting address of the protected memory region where the new permissions are securely stored.

This paper is an extended version of an earlier conference paper that appeared in HPCA 2014 [Elwell et al. 2014].

2. THREAT MODEL & ASSUMPTIONS

We assume that the Trusted Computing Base (TCB) of NIMP consists of the processor, physical memory, TPM and system buses. We also assume that system software layers including the hypervisor and the OS may be compromised, and these layers are not part of the TCB. The only software that is part of NIMP's TCB is a new module to ensure that the application binaries themselves can not be tampered with while they are being loaded from disk by the OS/hypervisor. This module is very simple and is described in detail in Section 7.

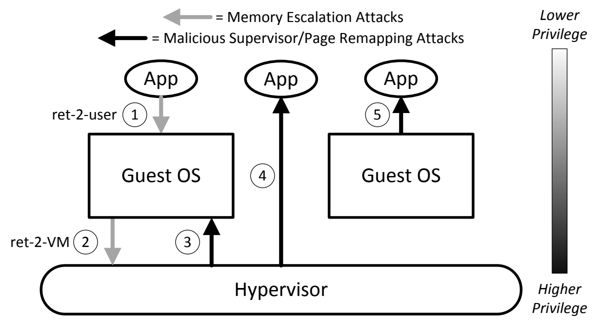


Fig. 1: Cross-Layer Attacks

We assume that hardware attacks (such as snooping on the memory bus or probing physical memory) are not part of the threat model. We make this assumption for two reasons. First, it is more difficult to probe physical hardware than to perform software attacks. Second, if the proposed architecture is deployed in a cloud environment, then it is reasonable to assume that a cloud operator will offer physical security of the system to protect its reputation.

We are concerned with cross-layer memory attacks, where one system software layer attempts to compromise the confidentiality or integrity of memory in a different layer. Specific attack categories that our proposed architecture protects against are the following.

— **Memory Escalation Attacks**. In this case, a lower-privilege level application attempts to alter or add to the memory footprint of a high-privilege level application. As an example of this, consider *ret-2-user* attacks. In this attack, an application writes malicious code into a page located in user-space and then exploits a vulnerability in the OS in order to overwrite the instruction pointer to cause a return to the code in user space with OS-level privileges [Kemerlis et al. 2012]. These attacks have affected all major operating systems and also targeted x86, ARM, DEC and PowerPC architectures [CVE-2009-1897 2009; CVE-2009-3527 2009; CVE-2010-4258 2010; de C Valle 2009; edb1 2009; edb2 2011; Security Focus 2009]. A guest OS can perform a (*ret-2-VM)* attack against a hypervisor [CVE-2012-5513 2012].
— **Malicious Supervisor Attacks**. In this case, a compromised higher-privilege software layer can read and/or modify data belonging to lower privilege-level software. For example, a hypervisor can use its unlimited memory access rights to steal a VM's private data. Similarly, the OS can access user-level applications. Inclusive memory permissions used in conventional designs naturally allow such attacks.
— **Page Remapping Attacks**. Here, the OS or the Hypervisor leverages its control over memory mappings/permissions to either map a private page into the address space of a possibly malicious process, or change permissions to allow itself malicious access. NIMP successfully defends against these types remapping attacks. However, rather than mapping private data outside of a process or changing the permissions in place, a malicious OS can also inject new code into a running process to intentionally leak the private data to an OS-readable location. The current implementation of NIMP mitigates these attacks using a validation based scheme described in Section 3.2.
— **Load-Time Attacks** Defeating the above listed attacks implies that the application is protected once it has been loaded into memory. However, the loading process itself presents attackers a potential opportunity to tamper with the application's code and/or static data to influence its execution. To protect against these load-time attacks, the NIMP incorporates a new module called the Key Permission and Integrity Module (KPIM), which is described in Section 7.

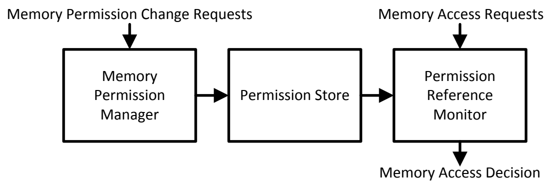

Memory Permission Change Requests
Memory Access Requests
Memory Permission Manager
Permission Store
Permission Reference Monitor
Memory Access Decision

Fig. 2: NIMP Design Overview

The NIMP system also protects applications against DMA-based memory attacks as described in Section 4.5

We do not protect against side-channels and covert channels; many previous works have addressed these issues [Wang and Lee 2007; Wang and Lee 2008; Domnitser et al. 2012]. We also do not protect against Denial-of-Service attacks initiated by higher privilege levels against lower levels, as higher levels can always deny service by preventing lower levels from executing.

3. NIMP DESIGN OVERVIEW

The main idea behind the design of the NIMP architecture is to make memory permissions non-inclusive. However, if privileged software layers were in charge of setting these permissions, they could simply allow themselves arbitrary access to any page, thus defeating security benefits of NIMP. To address this problem, NIMP restricts the ability to change page permissions only to the components that are part of the hardware TCB. To support complete system functionality and to allow the operating system and the hypervisor to manage the system's resources without the ability to directly manipulate page permissions, the NIMP design offers these system layers an interface to request these changes through a new ISA instruction. As a result, the NIMP system effectively decouples the tasks of memory access control (which is performed using a new instruction) and memory management (which is still performed by the OS and the hypervisor as in traditional designs).

Non-inclusive Memory Permissions are a lightweight form of mandatory access control [DOD 1985] enforced through hardware on the different layers of a virtualized system. An overview of the system is shown in Figure 2. NIMP makes permissions non-inclusive and enforces hardware validation of permission changes to close cross-layer attacks. Permissions are maintained at the physical memory page granularity in the permission store: a secure memory region inaccessible directly by software. To allow a layer (e.g., the OS) to manage another (e.g. user processes), permission changes are necessary as pages are allocated and assigned. Legal permission changes are specified by a set of rules that are stored in the memory permission manager and checked when page permission change requests are made. Permissions are enforced during run-time by the permission reference monitor which verifies that each memory access does not violate the permissions on the page it is accessing.

3.1. Description of Permissions

Under NIMP, access permissions for a physical page are expanded to include read, write, and execute permissions at each privilege level supported by the processor. Permissions are maintained with physical pages, rather than virtual pages, to prevent remapping attacks (where access to a page is accomplished by remapping it to a new virtual page with different permissions). Non-inclusive permissions support situations where a page is readable by a user-level process, but not by the OS managing that process. NIMP allows fine-grained access control for pages in a manner that supports non-inclusive access rights by various software layers in the system. In addition, two additional permissions are included: the Shared (S) bit and the Page Table (PT) bit. The S bit indicates whether a physical page is allowed to be mapped by the OS or the hypervisor into multiple page tables. The PT bit specifies that a page is part of the

Page	Other Bits (S/PT)	Hypervisor	OS	User
	S P	R W X	R W X	R W X
1	- -	- - -	- - -	- - -
2	S -	R W -	R W -	R W -
3	- -	- - -	- - -	R W -
4	- -	- - -	- - -	- - X

Table I: Example Page Permissions

page tables (PT), and is used to identify writes to them so that page mapping and unmapping events can be detected. Finally each entry contains a *map_count* field to track the number of page table entries that map a physical page. The *map_count* field and the PT bit are used in conjunction with the S bit to regulate sharing of physical pages, which is described in Section 4.4. Note that a 2-bit *map_count* field, resulting in a total of 16 bits for a PS entry, will only allow up to 4 entities to share a physical page by having them mapped simultaneously. To support a higher number of entities each PS entry could be expanded to 32 bits, leaving 18 bits for the *map_count* field and supporting about 250K simultaneous mappings. Finally if it is necessary to support sharing a page among every entity in the system, the PS entry can be expanded to include a map_count field that is as large as the PIDs used by system software. For example, in modern OS kernels PIDs are usually 32-bit integers, so a 64-bit PS entry with a 32-bit map_count field would allow every entity running on this system to map each page. Alternatively the OS can make due with a smaller mapping limit by unmapping the shared page from the process that least recently used it to provide a new mapping for another process. The performance implications of increasing the size of PS entries is explored in Section 8.

Table I shows some example permission sets. Page 1 in this table has no permissions available for any level. This state is the default state for a page that is not in use. Page 2 has read and write permissions for all privilege levels and is a shared page. This page could be used as a buffer to share data between various levels of the system. Page 3 allows user-level reading and writing from a page, but does not allow any access by higher levels. Such a page is used to store application data that is protected from both the hypervisor and operating system. Page 4 allows user-level execution, but no privileges at other levels. A page such as this could be used to store application code, while preventing against an attack such as *ret-2-user*, where the OS executes application code.

3.2. MPM and Assignment of Permissions

The rules for permission setting and changing are controlled in NIMP by the Memory Permission Manager (MPM). In this section, we describe its high-level functionality.

In existing systems, a higher-privileged layer not only inherits all permissions of the layers it manages, it is also empowered with the ability to set permissions itself. Thus, in addition to limiting the permissions of lower layers, a NIMP design must also restrict the responsibility of managing permissions to the trusted computing base; otherwise, a malicious layer can simply overwrite permissions to enable its attack. As such, assigning and altering a page's permissions is controlled using a set of rules enforced by the processor.

Table II shows a functional set of rules for the NIMP system. The * and # symbols have special meanings in this table. A star, which may appear on either side of the rule (current or requested permissions), matches either a zero or one and is known as a ”don't care” bit. The hash symbol may only appear in the requested permissions and denotes that this permission bit cannot change during this transition. It should be noted that the Key Permission and Integrity Monitor (KPIM) is the component in

		Initial Permissions					New Permissions					
		S/PT	**KPIM**	**Hyp.**	**OS**	**User**	**S/PT**	**KPIM**	**Hyp.**	**OS**	**User**	
Rule	**Requester**	S P	R W X	R W X	R W X	R W X	S P	R W X	R W X	R W X	R W X	**Action**
1	Hypervisor	– –	– – –	– – –	– – –	– – –	* –	– – –	* * –	* * –	* * –	None
2	Hypervisor	* *	– – –	* * *	* * *	* * *	– –	– – –	– – –	– – –	– – –	Wipe Page
3	OS	– –	– – –	– – –	– – –	– – –	* –	– – –	– – –	* * –	* * –	None
4	OS	* *	– – –	– – –	* * *	* * *	– –	– – –	– – –	– – –	– – –	Wipe Page
5	Hypervisor	– –	– – –	– W –	– – –	– – –	– –	– – –	– – X	– – –	– – –	None
6	OS	– –	– – –	– – –	– W –	– – –	– –	– – –	– – –	– – X	– – –	None
7	OS	– –	– – –	– – –	– W –	– – –	– –	– – –	– – –	– – –	– – X	None
8	Hypervisor	* –	– – –	* * *	* * *	* * *	* P	– – –	R W –	* – –	* – –	Wipe Page
9	OS	* –	– – –	– – –	* * *	* * *	* P	– – –	* * –	R W –	* – –	Wipe Page
10	Hypervisor	* –	* * *	* * *	* * *	* * *	# –	# # #	– – –	# # #	# # #	None
11	OS	* –	* * *	* * *	* * *	* * *	# –	# # #	# # #	– – –	# # #	None
12	KPIM	* –	* * *	* * *	* * *	* * *	# –	– – –	# # #	# # #	# # #	None

Table II: Permission Assignment Rules to Mitigate Cross-Layer Attacks

a NIMP system responsible for verifying the permissions and integrity of applications when code pages are loaded. It is described in detail in Section 7.

The rules are built around two assumptions. The first assumption is that once a page has its permissions assigned, those permissions should not change for the lifetime of that page. For example, Rule 1 says that the hypervisor is permitted to take any page with no permissions set (the null-state) and assign it any set of permissions. This operation would be performed right after a currently unused page is mapped into a page table and before it is put to use. Rule 2 dictates that the hypervisor can return any page to the null-state, but the hardware will automatically wipe the page and fill it with random data when this occurs. This operation is performed when a page is being removed from a page table, and ensures that no confidential information from the page can be leaked. However, under certain circumstances this could be used by an attacker to perform a non-control data attack [Chen et al. 2005]. To protect against the possibility that even random page contents could be exploited for an attack, NIMP can be extended to ensure that an application accepts a remapped page after the system remaps it. In particular, a remapped page is marked as invalid, causing a trap the first time it is accessed by the application, which can then abort if the remap event was not requested.

The second assumption is that a given privilege level can only assign permissions for itself and lower. This means, for example, that the OS can grant permissions for itself and user-level code to a page, but it cannot grant permissions to the hypervisor. This prevents a compromised OS from loading code onto a page, granting the hypervisor execute permissions, and then exploiting a hypervisor bug to execute the code with escalated privileges. Rules 3 and 4 capture this assumption by specifying the same intentions of rules 1 and 2, but for the OS. If the OS requires a page that has hypervisor permissions, then it asks the hypervisor to set it up and then verifies those permissions using the methodology described in Section 3.3.

For code pages, special care has to be taken to prevent pages from being both writable and executable at the same time to avoid code injection attacks. In conventional modern systems, this is provided by the NX bit.

Rules 5, 6 and 7 shown in Table II allow a page to transition from writable to executable modes while retaining the contents of the pages. Rules 8 and 9 allow the hypervisor and/or OS to allocate pages to be used as page table pages while ensuring that the PT bit is set. In addition these rules specify that the page should be wiped during this transition (zeroing out the page is fine here). This is used to protect against page table entry ”injections” that would otherwise bypass NIMP checks. These checks are described in Section 4.4. Finally, rules 10, 11, and 12 allow a given layer to drop its own permissions on a given page while leaving the contents and remaining permissions intact. This is used to allow the OS to load code and/or data from the disk and write it to a page, and then give up its write permissions when it is finished.

All other transitions are either disallowed, or the contents of a page are wiped out during the permission change, as dictated by Rules 2 and 4. In addition, the transition from writable to executable mode is only allowed once for a page without wiping out its contents. This restriction is allowed because our permission transition rules do not allow a page to transition from executable back to writable, unless the page is first brought into a null-state and its contents are wiped out.

Please note the following two aspects of the design.

— NIMP rules do not permit user-level code to assign any permissions. Applications request permission changes from the OS.
— Page tables are still managed as they currently are: the OS manages them for applications, and the hypervisor (depending on its implementation) manages them for the OS.

3.3. PRM and Verification of Permissions

The Permission Reference Monitor (PRM) has two responsibilities: (1) it ensures that a given memory access is allowed against the permissions of the physical page. This component is only a minor extension of the permission check performed by existing memory management units; and (2) conversely, it ensures that the permission specified for a given physical page is allowed by the requested memory access. To better understand this check, consider a potential attack against the NIMP permission system that may allow a compromised OS to violate the confidentiality of a user-level page. Assume a page has permissions `- - - | - - - | R W -`. An application may plan to use this page to store confidential information. During the application's run-time, suppose that the OS returns the page to the null-state using rule 4 (wiping the page in the process) and then uses rule 3 to set the page's permissions to `- - - | R - - | R W -`, hence allowing itself read access. Although existing confidential data on the page was wiped, any future data written by the application could now be read by the OS.

One solution to this problem would be to simply not permit the OS to alter the permissions of the page, and instead make that the sole responsibility of the application. The problem, however, is that then the OS cannot reclaim memory from a killed or misbehaving application, which is an unacceptable outcome. A better solution is for the application to be able to verify the permissions of the page prior to accessing it. Then, if the permissions have changed, the write should not occur. This check needs to be atomic with the write in order to prevent any sort of time-of-check race condition

In order to support this check, a new register is added for various software layers to specify what permissions they expect other software layers to have for all loads and stores executed. When a memory instruction is executed, this register is read then the PRM performs verification and either allows or denies the request. This is described in more detail in Section 4.3.

4. NIMP IMPLEMENTATION DETAILS

The section is organized into the three main components of the design: (1) the permission store; (2) the memory permission manager; and (3) the permission reference monitor.

4.1. Permission Store

Permissions for each physical page are stored in a special area of memory called the permission store consisting of a set of individual page permissions each stored in a Permission Structure (PS). The PS entry for each physical page specifies the currently active permissions for this page, such that separate "read", "write" and "execute" bits are provided for each of the privilege layers (KPIM, hypervisor, OS and user-level). In addition, the shared (S) bit, page table (PT) bit, and the map_count field are also included. We assume that 2 bytes are needed in memory to store each of the PS entries, although this can be increased with negligible impact. Figure 3a shows the PS layout for a single physical page. The permission store memory region is accessible only by hardware. Neither the OS nor the hypervisor have a direct access to this memory

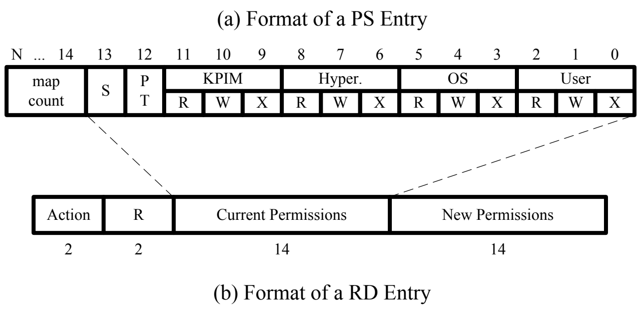


Fig. 3: Format of RD and PS Entries

and every request to set up or change the permissions has to go through the NIMP hardware.

Physical memory demand for storing the PS bits is modest: for a system with 16GB of physical memory, 4KB pages and 2 byte PS entries, the PS entries for all pages require only 8MB of memory (2 bytes for each of the 4M pages in the system), which represents 0.0005% of the total memory space. Even with 64-bit PS entries only 0.002% of the total memory space is used. Additionally, the PS bits are also cached in the instruction and data TLB entries, just like regular permission bits are cached in traditional systems. Therefore, the access to PS data in memory is only needed following a TLB miss. The PS data is also stored in regular caches as it is accessed, similar to other system-level data, such as the page tables.

It is important to observe that the permission bits are not modified directly by any software layer. All changes must be approved by the MPM; we add a new instruction called PERM_SET to the ISA to perform the validity check against the Rule Database and setup the page permission. This new instruction is described in detail below.

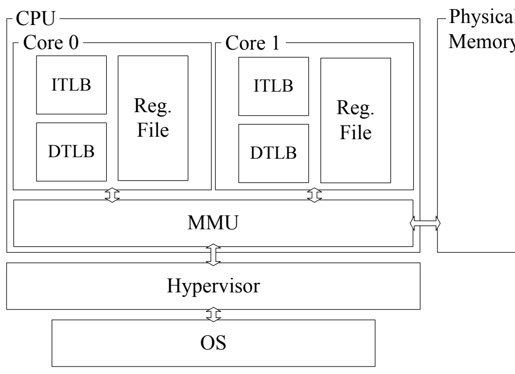
4.2. Memory Permission Manager

In this section, we describe the MPM implementation.

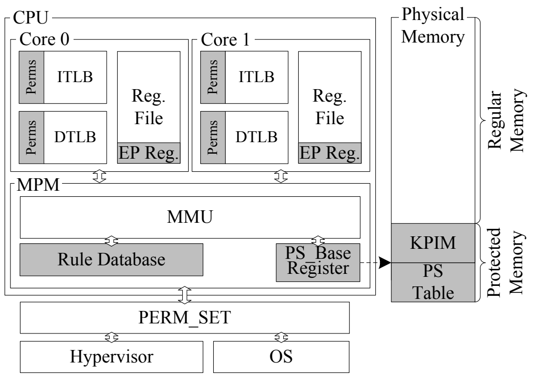
4.2.1. Rule Database and Secure System Boot. To modify permissions in NIMP, we rely on the use of securely stored permission modification rules — only the transitions specified by the rules are allowed, and this is directly controlled by the MPM hardware. All transitions not specified in the Rule Database are disallowed. These modification rules are stored in a dedicated Rule Database (RD), which is located inside a processor in a small TCAM structure. Once loaded at boot time, the contents of the RD never change. Initially, the rules are stored as part of the system BIOS. At system boot time, the integrity of the BIOS is measured by the TPM [TPM 2013] and then the rules are loaded into the RD.

Each RD entry has the following fields, shown in Figure 3b:

— **Current permissions**: which store the currently active permissions for a physical page, as specified by its PS entry.
— **New permissions**: this field has the same format as the current permissions field, but it specifies requested new permissions. The RD entry dictates whether or not the transition from the current set to the requested set of permissions is allowed.
— **The requester of the permission change**: knowledge of the requester allows NIMP to distinguish between hypervisor, OS kernel or a user-level process. Two bits (we call them the R bits) are needed to differentiate between these three entities.
— **Action bits**: which specify any special actions that need to be performed on the page for the rule to be upheld, such as wiping out the contents of the page, or encrypting it.



(a) Traditional Hardware and System Software



(b) NIMP Hardware and System Software

Fig. 4: Traditional Hardware vs. NIMP Hardware

4.2.2. Hardware Support for NIMP. We now describe the hardware support required to realize the MPM. The modified processor is depicted in Figure 4b. First, the processor is augmented with the cache-like structure that implements the RD. The RD is a fully-associative cache that is implemented as a TCAM (associatively-addressed memory that supports “don’t care” bits in the search key: this takes care of the “don’t care” bits in the rules) and its search key is composed of the tuple <Requester, Current PS, New PS>. Each RD entry is 4 bytes long. For a system with 12 rules (that we use in our evaluation and that are shown in Table II), the RD requires 48 bytes of storage plus the logic to implement a fully-associative search in TCAM. We show in the evaluation section that the access delay of such a TCAM is below that of an integer ALU.

In addition, the new hardware includes a 64-bit register (called PS_Base) that points to the beginning address of the PS table stored in memory. This register is protected from all software layers and is securely setup at boot time, along with the initialization of the RD. The index into the PS table to access the permissions associated with a physical page is computed in the following manner:

$Index = PS\_Base + (phys\_page\_number * sizeof(PS))$

Another register, whose size must be at least that of a PS entry (16-bits), is added to the architectural register file. This register, known as the EP register, is used to specify the Expected Permissions of any memory operation that is executed.

Finally, existing TLB entries (for both instruction and data) need to be augmented with PS entries for each page stored in the TLB, so that the PS bits can be read out directly from the TLB without requiring a memory access on a TLB hit. Since existing TLB entries already have 16 bytes (8 bytes each for virtual and physical page number cached), then the addition of 2 extra bytes of PS data results in an overhead of about 9% in terms of the TLB area (peripheral logic does not get impacted). Note that the PS bits are added as an extra field to each TLB entry, leaving the traditional protection bits unchanged. If the system uses the traditional way of managing permissions, these permissions bits are still available in the TLB and in the page tables and can be used by the processor and the OS.

4.2.3. Initial Page Permission Setup. When an application requests a dynamic page allocation (for example, using *sbrk* system call via *malloc()*), the OS or the hypervisor would locate a free physical page and establish a corresponding page table entry. Once this step has completed the OS or hypervisor will use the new PERM_SET instruction to assign permissions in a controlled way.

The PERM_SET instruction has the following format:

PERM_SET <virtual page address, new PS entry>

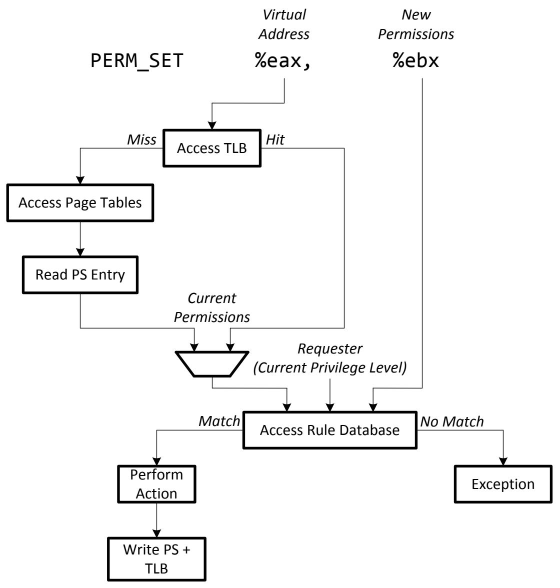
The activities triggered by this instruction are as follows. First, it performs an address translation using the virtual page address, TLBs and page tables. Since this page has been recently setup by the OS, the translation will be found in the TLB. If a translation is not found in the TLB, a hardware page-walk is performed and the corresponding PS bits are fetched. After that, the current set of permissions for the corresponding physical page (obtained from its PS entry) are read from the TLB entry. The combination of the selected current PS entry and the new PS entry as specified in the instruction are then used along with the current privilege level of the processor, to identify the requester, as a key to search through the RD. On a match in the RD rules, the hardware first takes any action specified in the actions field of the matched RD entry (such as wiping the physical page) and then sets up the new permission bits both in the TLB and in the Permission Store. If no match in the RD occurs for this type of transition, then the transition is disallowed and a permission violation exception is raised. All these activities are performed by the MPM. The execution of the PERM_SET instruction is illustrated in Figure 5a, and the process of accessing the RD is depicted in Figure 5b. It is worth noting that no matter who executes the PERM_SET instruction it is handled directly by hardware, specifically the MPM, so it cannot be trappable. However the Rule Database can be used to essentially limit who can use the instruction. For example, given the Rule Database in Figure II, since there are no rules that specify that the user-level can change permissions, user-level code will never be able to successfully use the PERM_SET instruction. The rules could easily be changed to allow user-level code to use PERM_SET instruction, which may increase security at the cost of more changes to user-level applications.

For the initial setup of permissions for static memory pages, such as code and static data, a similar approach is used. On a system call such as *exec()* or *CreateProcess()*, the OS first sets up the necessary amount of memory by creating virtual to physical mappings. Next, the OS uses the PERM_SET instruction to assign the initial set of permissions which allow writing to the page by the OS. The OS would then load the program by reading from the disk and placing it in memory. Finally the OS would relinquish its own permissions on any private user-level pages. This process represents a prime opportunity for a compromised OS to change the code and/or data of the application, and the mitigation of these types of attacks is described in detail in Section 7.

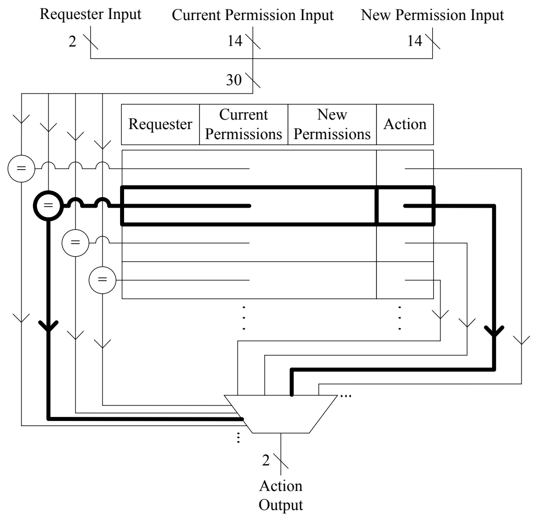
4.2.4. Permission Changes During Execution. There are situations during the normal execution of a system that the permissions of a page may need to be changed, for example, to support copy-on-write semantics. To request changes to the existing page permissions, any software layer requesting such a change does so through the hardware interface provided by the PERM_SET instruction described above. Regardless of what layer is invoking this instruction, it directly communicates with the MPM hardware. Notice that because neither the hypervisor nor the OS directly set the actual PS bits, the PERM_SET instruction should not be trappable or emulatable by the hypervisor.

We now describe the means by which the PERM_SET instruction is initiated by the various software layers. To enable the OS kernel to use the page permission change interface using the PERM_SET instruction, the implementation of the paging mechanism is slightly modified to call this new instruction after setting up the page table entry. Note that the existing kernel implementation of managing the protection bits (which stores them as part of the page table entries) can still remain intact. When the processor executes in the mode, then these page table entry bits can be ignored. Alternatively, they can still be consulted and the most restrictive of the two permissions (traditional and NIMP) will be enforced.

For the hypervisor, the permission changing process is similar to that of the OS kernel described above. Specifically, after the (nested) page table entry is setup, the call to PERM_SET instruction is initiated with appropriate operands. The hypervisor only needs to perform this activity for its own pages, as the pages belonging to the OS or the user-level processes are handled separately, as described above. The addition of the PERM_SET instruction to the paging implementation is the only modification to the hypervisor/OS code required by NIMP.



(a) Memory Data flow of the PERM_SET Instruction



(b) Accessing the Rule Database

Fig. 5: Activities Generated by the PERM_SET Instruction

In the NIMP design, there is currently no distinction between various user-level programs from the standpoint of managing page permissions. In traditional systems, software PIDs managed by the OS play this role. However, since the OS is untrusted in our threat model, we cannot rely on these software PIDs because they can be easily forged by a compromised OS. In our design, some level of protection is already presented by the S bit, which is a part of every PS entry. Specifically, if this bit is not set for a page, then this page cannot be mapped in more than one page table at a time, ensuring that it cannot be shared with any other application. It is reasonable to assume that if security is needed for some pages, then the application owning those pages would not set the S bit to avoid potentially exposing critical data. However, if a more flexible design is desired where several applications can securely share data in a controlled fashion, NIMP can easily adapt to them by adding hardware-generated PIDs [Chhabra et al. 2011; Suh et al. 2003] in place of software-maintained ones.

4.3. Permission Reference Monitor

We now describe how the last component of the NIMP system – the Permission Reference Monitor (PRM) – is implemented. The PRM's purpose is to enforce (in hardware) the new permissions, while also allowing the lower-privilege level software to verify that these permissions have not been tampered with. The PRM's permission enforcement responsibilities are similar to that of a traditional MMU, but some additional actions are also required to support permission verification.

In the NIMP system each load and store specifies the permissions that other software layers are expected to have for the targeted physical page, these are known as the Expected Permissions (EP). During the execution of each memory instruction, the EPs are compared against the actual permissions of the corresponding physical page and the result of the comparison determines whether the instruction completes successfully. This allows lower-privileged software layers to atomically check the page's permissions before writing any secrets to the page, avoiding any time-of-check to time-of-use issues.

To implement the specification of the EPs, the processor is augmented with a new register that holds the EPs in the same format as a PS entry. This register represents the EPs that are checked by all memory instructions. This new register is freely accessible by any layer of software and reads/writes to this register are not trappable or emulatable.

When a LOAD or a STORE instruction enters the memory access stage of the pipeline, the PRM unit extracts the EPs and compares them with the PS bits related to the physical page. On a match, the memory access is allowed, and on a mismatch, the access is not performed and an exception is raised. The exception handling actions depend on the specifics of the mismatch. The details of this process for a user-level verified memory access are shown in Figure 6. Note that the regular permission check (i.e. the user-level permission check in this case) is not depicted since it is similar to traditional checking.

4.4. Page Sharing in NIMP

Sharing pages between multiple applications in the NIMP framework is accomplished through the use of the S bit of each PS entry. The S bit in each PS entry denotes whether or not the corresponding physical page is allowed to be mapped into multiple page tables simultaneously by the OS. In addition to the S bit, the NIMP hardware utilizes the PT bit to capture all writes to page tables and the *map_count* field to identify the different cases and act accordingly during an attempted mapping.

To enforce the S bit's shared property the NIMP hardware must be able to intercept every page table write to allow checking (and updating) of the *map_count* field and S bit to deny a mapping if the property is being violated. To intercept every write to page tables, the NIMP system leverages the PT bit to identify physical pages that are being used as page table frames. Only pages with the PT bit set will be considered valid page table frames and thus used by the hardware during page table walks. To ensure that malicious mappings are not written to a page before the PT is set, Rules 7 and 8 in Table II are the only rules that allow the PT bit to be set and the associated action requires the page to be wiped, thus eliminating any entries that existed prior to the PT bit being set.

When the NIMP system intercepts a page table write it must check the corresponding PS entry. The state and associated action taken can be broken down into the following three cases:

— *map_count* $==$ 0: Since this physical page is being mapped for the first time, it doesn't matter if the S bit is set or not, the mapping is allowed.
— *map_count* $==$ 1, S bit unset: The shared bit is not set, so the page should only be mapped once. Since it has already been mapped once (*map_count* $==$ 1) this mapping is denied.
— *map_count* $\geq$ 1, S bit set: The page has already been mapped at least once, but since the S bit is set it can be mapped again.

Once a shared page has been successfully mapped into a process, the process will access it just like an other memory page. This means that after the initial setup a shared page will incur no additional overheads while it is being utilized by the process(es).

4.5. Other Considerations

To complete the NIMP implementation, a few additional system-level considerations have to be taken into account.

Secure Context Switches and Interrupts: During context switches and interrupts, the registers of a running entity may be exposed. In order to prevent such an exposure, register contents need to be saved in protected memory by the NIMP hardware, and then wiped before control is transferred to a higher-privilege interrupt handler. The NIMP hardware will then have to be involved in restoring the register state from the protected memory when the process is resumed. This can be accomplished, for example, by allocating a page for each process and mapping it at an implementation-

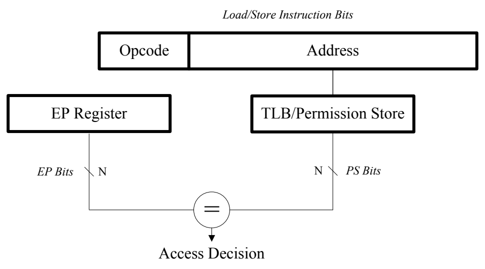


Fig. 6: Permission Verification by the PRM

specific virtual address (for example the first virtual page) with only user-level read and write permissions. The NIMP hardware could then use this page to store and restore registers.

Secure Page Swapping: While NIMP prevents malicious supervisor software from getting access to the application memory pages, this same functionality also prevents that software from reading pages in order to swap them to disk. In order to support swap-out, the rules can be extended to allow a supervisor to add read permission for itself on a page, but the associated action encrypts the page using a key derived from a hash of the application's code space, a random nonce generated by the hardware and stored in the application's memory space, and the page's current permissions. In addition, for every swapped page, the hardware computes an HMAC to be sent along with the encrypted page to the disk and back. The supervisor is unable to determine the encryption key without knowing the nonce, and including a hash of the code in its derivation ensures that the page can only be swapped back into the same application. Furthermore the HMAC protects against malicious modification of the encrypted page while it is on disk. During a swap-in, the encrypted data can be read back from disk into memory, the correct permissions restored to the page, the HMAC verified and the decryption performed by the hardware. The random nonce can be stored in the virtual address space of a process similar to the register context, as described above.

Supporting DMA: To handle DMA operations, the NIMP system uses the same set of permissions as those assigned for the software privilege level responsible for initiating DMA requests. It would be possible to assign DMA its own set of permissions under the NIMP framework, but the security benefits of this are not clear.

4.6. Software Changes to Support NIMP

To secure running systems, NIMP requires a number of changes to the software layers. First, the hypervisor and the OS must be modified to make use of the PERM_SET instruction in order to manipulate the newly added PS entries. This task is simplified by the fact that these layers are already equipped with capabilities to modify page table entries. Specifically, the functions that modify the page table entries must be extended to use the PERM_SET instruction. In addition, systems software layers usually have their own architecture-independent memory permissions, which are translated to the permissions that a given architecture actually supports. NIMP support will therefore require translating these permissions to the NIMP system for use with the PERM_SET instruction.

Applications running at the user level must be adapted to manipulate the register that holds the EP bits. An existing application can be quickly configured to run almost unmodified by simply configuring the register such that any permission setup is allowed. This will not provide any additional security, but will get the application running. From there, it can be modified in phases by slowly adding the correct register settings for various parts of the code. There are a number of ways that changes to EP register can be incorporated into the software. In some cases it may be sufficient to allow the standard libraries to set the register to a reasonable default value. As an example, before the `main()` function is called, the register could be set to allow read and write permission by any layer. When a library is used to perform a sensitive task, such as encryption, it can set the register to a more restricted value such as allowing read and write operations only for user-level code. For restricting the permissions specified by EP register in non-library code, compiler support (for example, in the form of annotations) could be leveraged to allow developers to mark the data structures that contain sensitive data. When these data structures are accessed, the register can be configured appropriately.

Finally, the system software must be augmented to invoke the Key Permission and Integrity Module (KPIM, described in Section 7), after it has finished loading an application or a VM. Overall, the NIMP system requires rather modest changes to system and application-level software.

5. SECURITY ANALYSIS OF NIMP

The NIMP design was partially motivated by the concept of capabilities. The key difference between the implementation of NIMP and the classical definition of capabilities is that the capabilities in NIMP (i.e. the memory permissions stored in PS entries) are stored in a central location rather than stored by the entities themselves. However, from a security standpoint the two are quite similar. Entities in the NIMP system request the capability to access memory, either directly or indirectly through another software layer, and the NIMP hardware either grants or denies the request. Capabilities are then checked implicitly when an entity issues a memory operation. While a formal proof of NIMP security is left for future work, the security properties achieved by NIMP are comparable to those that are offered by capability-based architectures. A more detailed comparison between these two approaches can be found in Section 9.

We now summarize how the NIMP architecture mitigates cross-layer attacks considered in our threat model and compare it to Intel's SMEP/SMAP technology.

5.1. Mitigating Malicious Supervisor Attacks

To protect against malicious supervisor attacks, most user-level pages will be set up with permissions that match those of Page 3 shown in Table I (with the exception of OS communication buffers). Such pages are configured to be readable and writable only by the application layer and do not have the shared bit set. The OS cannot access such a page directly, due to the absence of any OS permissions in the corresponding PS entry, and neither can it create any new mappings to the physical page due to the shared bit being cleared. The only way a malicious OS could attempt to compromise data residing on such a page is by changing the corresponding PS entry to include OS read/write permissions (or to set the shared bit). However, there is no single rule in the RD that allows the OS to grant itself these permissions, and any attempt to perform such an unspecified page permission change will result in the generation of a security exception by the MPM hardware, as shown in Figure 5a. Alternatively, the OS may attempt to use more than one permission transition to grant itself read/write access. These types of attacks are classified as Page Remapping Attacks which are addressed in the following subsection.

In contrast, Intel's SMEP/SMAP technology must offer a way for the OS to temporarily disable SMAP protection for the purposes of handling user-level data during system calls. This is done via the STAC and CLAC instructions which can be used by a malicious OS to grant itself complete access to all of the user-level pages. This means that SMEP/SMAP cannot be used to protect a user-level application's memory from a malicious OS. Furthermore, unlike NIMP, SMEP/SMAP does not generalize to include the hypervisor and thus cannot protect virtual machines from a malicious hypervisor.

5.2. Mitigating Page Remapping Attacks

Page Remapping Attacks can be performed in two ways. In the first attack variation, a target page is remapped within the same address space, but with a different set of permissions. For this example we will again assume that the page in question initially has permissions that match those of Page 3 from Table I. The OS could use Rule 4 from Table II to remove all permissions from the physical page, and then use Rule 3 to re-initialize its PS entry with both OS and user-level permissions. In this case, the OS *must* utilize Rule 4, since it is the only rule whose initial permissions would match and as a consequence any existing content on the page would be wiped off, thus preserving its confidentiality. However, any new data written to this page by an unsuspecting application would subsequently be accessible to the OS. The permission verification mechanism described in Section 4.3, however, prevents this. In this case, the application would detect the new permissions upon its first attempt to write, as the permission verification would fail before any data was written to the page.

The second variation of remapping attacks involves remapping a page to a different address space, such as that of another process. To prevent this type of attack, NIMP ensures that when a non-shared page is unmapped, its contents are zeroed out before a new mapping can be established. Mapping and unmapping events are detected when write accesses occur to pages marked with the PT bit, which is stored in each PS entry. For example, if the OS unmaps a private page from a process, the NIMP hardware would be alerted of this change, at which point the physical page will be zeroed out and the map_count field of the corresponding PS entry will be set to zero. Only after the map_count field has been set to zero (and thus, the content of the page wiped) could a new mapping to the same physical page be created.

In contrast, Intel's SMEP/SMAP mechanism does not protect either the page tables or the underlying physical pages that are mapped by them, so a malicious OS has opportunities to launch a page remapping attack. For example, nothing prohibits the OS from creating a new mapping in its own address space with completely different permissions that allow it to read or write any data placed on the remapped page.

5.3. Mitigating Memory Escalation Attacks

In current systems, these types of attacks leverage the fact that a page marked as executable by a user-level application can also be executed in a hypervisor/OS context. Under NIMP, it is theoretically possible to use Rules 1 and 3 in Table II to create a page where a higher privilege level has "execute" permission, while some lower privilege level has "write" permission, thus creating an environment for these attacks. However, the only way that such a combination of permissions is possible is when the victim layer itself gives the "write" permission to the lower privileged layer that initiates the attack. It is not possible for the attacking (lower-privileged) layer to set up the "execute" permission for a higher-privileged layer. This is guaranteed by the fact that the transitions in the Rule Database that are usable by the OS cannot be used to grant any hypervisor permissions. Similarly, there are no transition rules that are usable by user-level code that grant any hypervisor or OS permissions. This means that to successfully perform such an attack, the attacker must minimally be able to execute a PERM_SET instruction in the context of the victim. To reuse an existing page, the attacker requires two PERM_SET instructions, the first to clear the permissions and the second to re-initialize them with the desired value. Alternatively the attacker could create a new mapping to a previously unused page and use a single PERM_SET instruction to set the permissions. Intel's SMEP/SMAP is designed specifically to stop these types of attacks (SMEP in particular). Similarly to NIMP, this protection can theoretically be broken if the attacker can disable SMEP, which is done via clearing a bit in the CR4 control register. Since writing to CR4 is a privileged operation, it requires the attacker to be able to minimally execute an instruction to write CR4 in the context of the OS. However, since SMEP/SMAP does not generalize to the hypervisor, it can only be used to mitigate attacks against the OS in a non-virtualized system whereas NIMP can protect the hypervisor as well.

5.4. Mitigating Load-Time Attacks

In an environment where an application is protected once it has been loaded into memory, the loading process itself presents attackers a potential opportunity to tamper with the application's code and/or static data to influence its execution. In the NIMP system, the Key Permission and Integrity Module (described in Section 7) is designed specifically to thwart such attacks. This new module, which is invoked after the OS has loaded the program, cryptographically measures the application's code and read-only data sections. If this measurement succeeds, the module releases an encryption key to the application to encrypt and decrypt sensitive data, thus protecting it from a possibly malicious loader/OS. This allows developers of applications to ship sensitive data in encrypted form. The data can only be decrypted and subsequently accessed if the application is loaded properly without modification.

6. IMPLEMENTING SMEP/SMAP IN NIMP

The NIMP approach to managing memory permissions provides a general framework that can be used to implement or emulate a number of previously proposed memory protection schemes. In this section, we demonstrate the generality of NIMP by using the permission change rules to implement Intel's SMEP/SMAP mechanism [Intel 2014]. While the emulation of SMEP/SMAP in NIMP wouldn't offer any security benefits over SMEP/SMAP, it shows the flexibility of NIMP in terms of "backwards compatability". A NIMP system could be configured this way and run nearly unmodified code and then gradually move to a set of rules more like those in Table II.

SMEP (Supervisor Mode Execution Prevention) and SMAP (Supervisor Mode Access Prevention) are new security mechanisms available in recent Intel processors that protect user-level pages from being executed and/or accessed by supervisor mode code. Essentially, SMEP/SMAP alter the semantics of the User/Supervisor bit of a page table entry. Whenever that bit is set to user mode and an OS access to the page takes place in SMEP/SMAP mode, a page fault is generated by the hardware indicating an access violation. Table III shows the required NIMP permission change rules to implement functionality similar to SMEP/SMAP.

The first two rules from this table (rules 1 and 2) are used to create the OS-level pages. These pages are not accessible by user-level code. Similarly, the next two rules (rules 3 and 4) are used to create user-level pages. However, for functionality reasons the OS must be sometimes allowed to access user pages. To accommodate this, SMAP-capable processors provide two instructions to temporarily allow such accesses. The CLAC instruction clears a flag in the processor which then allows the OS to access user-level pages. In the NIMP framework, this instruction would rely on Rule 5, which allows the OS to give itself either read or write permission for a user-level page. The corresponding STAC instruction sets the SMAP flag once again, denying the OS access to a user-level page. This instruction is emulated by Rule 6. It should be noted that unlike SMAP, SMEP does not need to be toggled at runtime for correct functionality, since the OS should never execute code from a user-level page. This property is embedded into Rule 5, which explicitly states that the OS may give itself read or write, but not execute permissions for a user-level page. Note that the hypervisor permissions have been left out of this table, since it is unclear how SMAP interacts with the hypervisor. Furthermore, since no additional actions need to be performed by SMEP/SMAP hardware, the action column has also been omitted.

In comparison to SMEP/SMAP, the NIMP system has some significant advantages. First the NIMP implementation of SMEP/SMAP can be trivially extended to support a hypervisor simply by adding more rules to the rule database, which highlights it's programmability. Second, SMAP relies on the OS to toggle the enforcement of the it's policy for correct functionality, this means that SMAP cannot be used to defend against a malicious and/or compromised layer of supervisor software since it can simply turn off SMAP's enforcement temporarily, or change the permissions associated with the page. Finally, SMEP/SMAP cannot protect against remapping attacks of any kind, since the OS remains in full control over both mappings and permissions.

Other memory protection schemes, such as that proposed in HyperWall [Szefer and Lee 2012] can also be implemented within the NIMP framework by adjusting the rules accordingly.

		Initial Permissions			New Permissions			
		S/PT	OS	User	S/PT	OS	User	
Rule	Requester	S P	R W X	R W X	S P	R W X	R W X	Note
1	OS	- -	- - -	- - -	- *	* * -	- - -	OS data page
2	OS	- -	- W -	- - -	- -	- - X	- - -	OS code page
3	OS	- -	- - -	- - -	* -	- - -	* * -	User-level data page
4	OS	- -	- - -	- W -	- -	- - -	- - X	User-level code page
5	OS	* -	- - -	* * *	* -	* * -	* * *	CLAC
6	OS	* -	* * -	* * *	* -	- - -	* * *	STAC
7	OS	* *	* * *	* * *	- -	- - -	- - -	Revoke Page

Table III: Permission Assignment Rules to Emulate SMEP/SMAP

7. SECURE DEPLOYMENT AND LOADING

The NIMP architecture protects an application from a compromised hypervisor and OS once its code and data have been loaded into memory. In this section, we examine approaches to preventing malicious accesses *before* the application reaches memory. We accomplish this by making use of a secure module known as the Key Permission and Integrity Module (KPIM). The Key Permission and Integrity Module is loaded into memory by the OS, but runs in a NIMP-protected memory space and is verified by the hardware itself. It has special privileges because it is part of the TCB.

Trustworthy loading of applications is not a new technique, however the KPIM has a somewhat specific and unique set of requirements that must be met and features that it must support. The KPIM, being the only trusted software in the NIMP system, must be as simple as possible so that it can be implemented in a trustworthy manner. The KPIM cannot depend on privileged software layers to perform any of it's tasks, such as using a driver to access hardware external to the CPU to aid it. In contrast to other works that provide trustworthy loading of applications, the KPIM is designed specifically for use in the NIMP system so it can be simpler and more light-weight than other solutions. For example, the Trusted Platform Module (TPM) [TPM 2013] supports remote attestation of applications, a feature that would go unused in the NIMP system as the integrity of the application's code needs only to be proven to the application itself. Furthermore, if the TPM were used by the KPIM then it would require direct access to the TPM to communicate with it. The KPIM would also need to include any drivers necessary to do so to avoid trusting the OS. This bloats the size and complexity of the KPIM which reduces its trustworthiness. Other solutions such as binary encryption as proposed in [Cappaert et al. 2006] offer more protection than is required in the NIMP system at the cost of run-time performance due to the dynamic nature of decrypting binaries on-the-fly. Finally, some works in this area have a significantly different goal or threat model than the NIMP system, for example [Payer et al. 2012] aims to use a trusted loader to prevent the host system from being compromised by an untrusted application that is being loaded.

In developing the KPIM, we make the following assumptions: 1) There is an external party, the *client*, who is interested in securely running an application on a NIMP system. 2) Each CPU has a public/private key pair (CPU_{pub}, CPU_{priv}) embedded within it during manufacturing. 3) Each CPU manufacturer has a public/private key pair (MAN_{pub}, MAN_{priv}). The manufacturer's public key is embedded in every CPU during manufacturing.

7.1. KPIM Bootstrap

The KPIM is part of the TCB. To this end, we assume that a range of physical pages is reserved in the NIMP architecture for use by the KPIM. To ensure that it has not been tampered with, the KPIM is signed by the processor manufacturer using the manufacturer's private key (MAN_{priv}). This signature will be used to provide integrity verification for the KPIM during each boot. During boot, the OS will place the KPIM and the manufacturer's signature of the KPIM into the reserved memory range. The first time control jumps to the KPIM, the NIMP hardware will hash the KPIM's reserved memory region and verify its signature using the embedded public key of the manufacturer (MAN_{pub}). If this check succeeds then the KPIM is enabled by first correctly setting the NIMP permissions for the KPIM's pages and then setting a CPU status bit to denote that the KPIM is secure. This flag will be used during run time to allow the hardware to decide whether or not to decrypt data on behalf of the KPIM. The *client* is assured that this verification is successful by the availability of key k_e, which is described below.

7.2. Application Deployment

The deployment of the application binaries and of its initial data offers a prime opportunity for a compromised hypervisor/OS to pry on confidential data and to alter data and/or code before it even reaches the file system.

The code and data of an application are typically loaded from disk. In a typical scenario, the data is sensitive but the application itself is not (for example, the application may be a commercial DBMS, whose binaries are easily obtained, with valuable data held in databases). We must therefore protect both the confidentiality and integrity of the data files, but ensuring only the integrity of the application files is sufficient. Confidentiality of data is guaranteed by encrypting it while in the file system. The data can be decrypted by the application itself using a key passed to it by the hardware as it is needed. This key could be k_e itself or a key that is encrypted using k_e to provide flexibility in encryption algorithms. To ensure integrity, the KPIM relies on a keyed hash to verify that the application has not been tampered with (details below). This approach is quite general: sensitive binaries could be kept encrypted on disk if the application comes with a decryption wrapper; at the other end of the spectrum, public data can be left in the clear on disk; configuration files are encrypted if deemed sensitive.

To deploy an application on NIMP-protected hardware, the client begins by generating two symmetric keys, k_e that will be used to encrypt sensitive data files, and k_h that will be used to compute a hash of the non-sensitive binaries — both could be derived from a single master key using one-way functions. The client then encrypts each sensitive file of the application using k_e and computes a keyed hash of each binary using k_h. The client then uses the CPU's public key (CPU_{pub}) to encrypt k_e, k_h, and the keyed hash. This is known as the *load credential*. Next, the client sends all application files and the load credential to the provider — this does not need to take place over a secure channel as sensitive files are either encrypted or hashed (or both).

7.3. Loading an Application

When asked to load an application, the OS puts the application into memory, just like any other application. In addition, the load credential is also placed in memory at a predetermined virtual address where the KPIM will find it. Once the OS has placed them into memory, it calls the KPIM. The KPIM asks the CPU to decrypt the load credential using its private key (CPU_{priv}), which only succeeds if the KPIM is executing. The KPIM then verifies the integrity of the application using the keyed hash from the decrypted load credential. If this check is successful the KPIM will place k_e and k_h at an implementation defined virtual address for the application to use. Finally, the KPIM verifies any permissions (including verifying access rights to the page(s) containing k_e and k_h), and transfers control to the application.

In the above design, the application is responsible for ensuring the confidentiality and/or integrity of its files. It will perform hashing and/or decryption itself using keys supplied by the KPIM, except for the binaries whose integrity is handled by the KPIM. Applications designed for in-house use rarely take such precautions, but we believe this is prudent engineering for deployment in an untrusted environment such as the cloud systems NIMP seeks to secure. Were this unacceptable, our design can easily be modified so that the KPIM (or a companion module) handles the decryption of sensitive files and more extensive integrity checks. Batch applications, where the client supplies (encrypted) input and later retrieves (encrypted) results could then run without modifications. Further engineering would be needed, however, for applications that need

Parameter	**Configuration**
Window Size	8-way issue, 128-entry ROB, 32-entry Issue Queue, 48-entry LSQ
Data TLB	128-entry, 4-way
Instruction TLB	64-entry, 4-way
L1 I-Cache	32 KB, 2-way, 64B line, 1 cycle hit
L1 D-Cache	32 KB, 4-way, 64B line, 1 cycle hit
L2 Unified Cache	256 KB, 16-way, 64B line, 10 cycle hit
L3 Unified Cache	8 MB, 32-way, 64B line, 30 cycle hit
Memory latency	300 cycles

Table IV: Configuration of the Simulated Processor

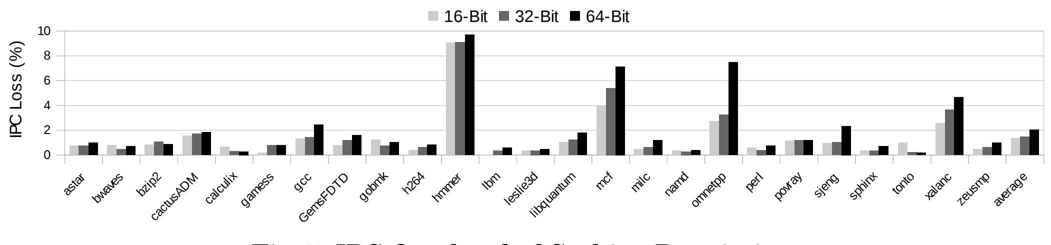


Fig. 7: IPC Overhead of Caching Permissions

to communicate with the external world, as some system-bound data would need to be encrypted (databases for example) while other should not (e.g., web pages to be sent to a user, or files used to communicate with other applications).

8. PERFORMANCE AND COMPLEXITY

In this section we evaluate the performance impact and hardware complexity of NIMP.

8.1. Performance Evaluation

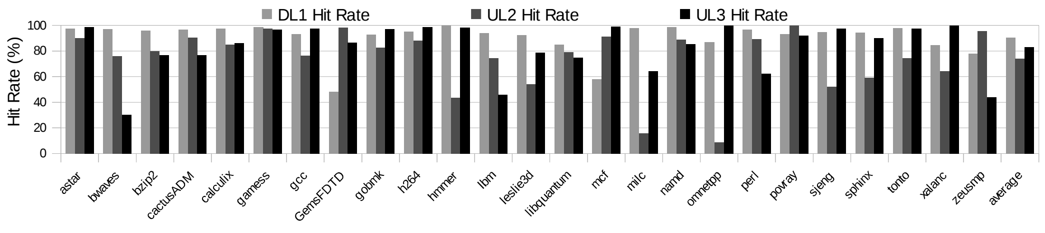
For estimating the impact of the extra delays due to accessing the new permission bits, we used MARSSx86 [MARSS 2013] — a full-system x86-64 simulator. Our processor configuration is shown in Table IV.

Figure 7 shows the decrease in IPC for each of the simulated benchmarks due to the delays of accessing permission bits on the TLB misses for three different PS entry sizes. The largest performance loss of 9.7% was observed for *hmmer* benchmark, followed by 7.5% for *omnetpp* using 64-bit PS entries. The majority (68%) of the benchmarks experience less than 1.5% degradation, with the average being about 2%.

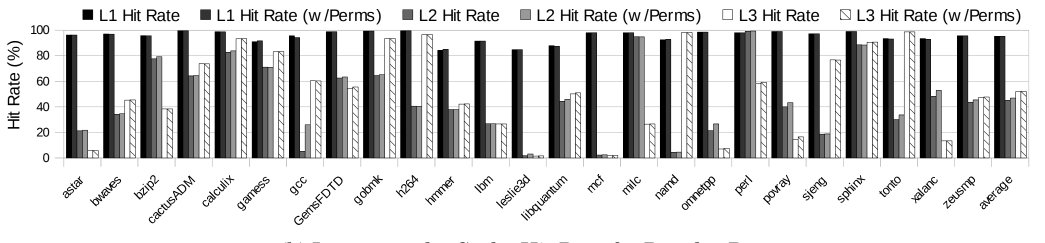
Figure 8a shows the hit rates to the various levels of caches for the PS data. On the average, 91% of the requests for PS data are satisfied from the L1 cache, and 74% of the L1 misses are satisfied from the L2 cache, thus keeping the number of accesses to the L3 cache and main memory very small.

Finally, Figure 8b shows the impact of the PS bits on the cache hit rate of the regular accesses using 64-bit PS entries. As seen from the figure, there is no noticeable impact for all levels of caches for all benchmarks that we simulated.

Next, we conservatively estimate the performance overhead of zeroing out the page contents in hardware, if this is dictated by the permission change rules. To access the frequency of operations requiring page permission changes, we profiled the Linux kernel using the built-in *ftrace* utility. Specifically, we collected the information about all system calls and filtered out the ones that resulted in a request to a page permission change by calling appropriate kernel functions. We then evaluated how often permission change requests occur on a variety of applications. We studied both client-side and server-side applications. For client-side applications, we evaluated the Chromium web browser, the process of booting a VM with VirtualBox, and opening a spreadsheet in LibreOffice.



(a) Cache Hit Rate for PS Bits



(b) Impact on the Cache Hit Rate for Regular Data

Fig. 8: Impact of NIMP on Caches

For these three client-side applications, we conservatively assumed that the most expensive action (e.g., zeroing out a page) is required on every such permission change request, and evaluated the combined overhead of this operation. We then also conservatively assumed a latency of 8 CPU cycles for writing 64 bits to memory. Since a 4KB page contains 512 such lines, 4096 cycles are needed to wipe off the entire page (for comparison, HyperWall assumed 512 cycles to wipe off the entire page; under those assumptions our overheads will be even smaller). It should be noted that while our latency is very conservative (1 cycle/byte) we did not model the impact that caching such pages would have on other applications. As seen in Table V, performance overheads are very small. Even combined with additional overhead due to the PS bit accesses, the overall loss does not exceed 1% in most cases.

Next, we examined a number of server-side applications. These applications include a MySQL database server daemon, an Apache webserver daemon, and an OpenSSH server daemon. Since these applications tend to go through different phases of execution, especially in terms of memory management, we measured these different phases separately. For MySQL the phases we measured included daemon initialization, database preparation, handling a series of queries serially with the database in read-only mode, and cleaning up a database. For Apache, only two phases were measured: daemon initialization and handling a series of requests for a 10MB file from a number of clients simultaneously. For OpenSSH, we measured only a secure copy operation (i.e. using *scp*) of a larger file (roughly 540MB) from the host machine to a remote machine.

Table VI shows the results obtained for the server-side workloads. In general, all of these applications generate a majority of their permission change requests during the initialization phase of their daemon in a relatively short amount of time. Once their execution reaches a more steady state, they tend to request far less changes.

We also estimated the overhead that is added by the KPIM when it cryptographically hashes binaries during the loading process. The results of these experiments are shown in the “KPIM Cycles” columns of Tables V and VI. For each program, the reported number represents the average of 50 measurements of the given binary. These experiments were performed on a machine with 2.8GHz CPU with Nehalem microarchitecture running Debian Linux. The binaries measured by KPIM were obtained from the standard Debian repositories and were not custom-compiled. The measurement

Application	Changes Per Second	Zeroing Cycle Overhead	KPIM Cycles
VirtualBox	2765	0.4%	39M
Chromium	2973	0.4%	1.1B
LibreOffice	8608	1.2%	77K

Table V: Initialization Overhead
(Client Workloads)

Application	Changes Per Second	Zeroing Cycle Overhead	KPIM Cycles
MySqld Initialization	9648	1.3%	102M
MySql Preparation	816	0.1%	
MySql Queries	0.2	2.7e-5%	
MySql Cleanup	43	5.9e-3%	
Apache Initialization	7639	1.0%	6.6M
Apache Benchmark	1289	0.2%	
sshd scp	125	0.02%	13.1M

Table VI: Initialization Overhead(Server Workloads)

results are reported for only the code and read-only data sections, dynamically linked libraries were not measured.

The exact overhead of KPIM depends on the size of the applications being loaded. In some cases, notably *LibreOffice*, the application binary is essentially just a launcher that is responsible for loading the rest of the required modules. *LibreOffice* loads this small launcher to determine the type of file being operated on (text document, spreadsheet, presentation), and subsequently loads the specific modules to handle that type of file. In this case, the overhead of KPIM is very small - just 77K cycles. On the other hand, large applications incur significantly longer KPIM delay. For example, for Chromium the delay is 1.1B cycles. However, even in this case it is below half a second, and this delay is only incurred once during the loading of the application. On newer CPUs that feature special instructions to accelerate hashing, the delays of KPIM would be further reduced.

8.2. Evaluating NIMP Hardware Complexity

To evaluate the delay and area overhead of the additional hardware required by NIMP, we implemented the NIMP logic in Verilog HDL using Xilinx ISE WebPACK 14.6 [Xilinx 2013]. Because the absolute timing on the target FPGA platform is slow, for comparison purposes we also implemented other basic CPU logic, such as a 64-bit integer ALU. Assuming that this basic operation is implemented in a single cycle on a typical CPU, we compared the delays of the NIMP logic with the ALU delay and the results showed that the access to the Rule Database can be performed within a cycle, even if the Rule Database was extended to 16 entries. Due to the small size (48 bytes to implement the policy specified in Section 3) and infrequent access of the Rule Database, the power and area overheads of such a structure are negligible.

Finally, we evaluated the impact of slightly widening TLB entries to support storing of the new permission bits inside the TLB. This results in 9% increase in the TLB area, and 1% increase in the TLB access delay. This is because only the word select delay increases slightly, while the delays of associative search, bitline delays, and the delays of sense amplifiers and other peripheral logic for reading out the data are not impacted.

9. RELATED WORK

We subdivide previous solutions into two categories: software-only and hardware-supported schemes.

9.1. Software Approaches

A limitation of many software security schemes that do not include OS or hypervisors in the TCB is that they rely on some other trusted layer. Unless this layer is formally verified, it is not impossible to devise attacks on it. sHype [Sailer et al. 2004] proposes a hypervisor to secure VM interactions. VMGuard [Fang et al. 2010] is a technique for protecting the management VM in Xen. A number of efforts use introspection to identify the presence of malicious code [Azab et al. 2009; Sharif et al. 2009; Litty et al. 2008; Garfinkel and Rosenblum 2003; Jiang et al. 2007; Jiang and Wang 2007; Payne et al. 2007; Payne et al. 2008]. Other works use the hypervisor to protect the guest OSes [Seshadri et al. 2007; Klein et al. 2009; Riley et al. 2008].

Overshadow [Chen et al. 2008] protects applications from the compromised OS by presenting the OS with an encrypted view of physical memory, while NoHype [Szefer et al. 2011] eliminates the hypervisor layer altogether. Cloudvisor [Zhang et al. 2011] uses a small security monitor below the hypervisor, using nested virtualization. Inktag [Hofmann et al. 2013] introduces paraverification mechanisms, forcing an OS to do additional computations to make it less complex for the hypervisor to verify its behavior.

9.2. Hardware-Supported Approaches

To address the limitations of software approaches, several hardware-supported schemes have been recently proposed. HyperWall[Szefer and Lee 2012] protects guest VMs from a malicious hypervisor. Unlike NIMP, HyperWall's threat model only assumes untrusted hypervisor, but the guest operating systems running inside the VMs are assumed to be trusted. Moreover, instead of checking for the validity of page permissions, NIMP hardware checks for the validity of permission transition rules, which are expressed with the purpose of disallowing the most common attacks, but without inhibiting normal functionality of programs.

Also similar to NIMP, H-SVM [S.Jin et al. 2011] reduces the trusted computing base only to hardware. Nested page tables are stored in a protected memory region, which is only accessible to the H-SVM hardware. The H-SVM hardware validates all page table updates initiated by the hypervisor through a series of microcode routines. While H-SVM focuses on the integrity of page tables (that are themselves stored in the protected space), NIMP only protects the page permission bits, which are decoupled from the main page table structures and are indexed by the physical page number.

HyperCoffer [Xia et al. 2013] is built on the idea of placing a new layer — VM-Shim — in-between a VM and the hypervisor. Each VM-Shim instance executes in a separate protected context and only declassifies necessary information designated by the VM to the hypervisor and external environments. Some hardware modifications are also needed. The Bastion architecture [Champagne and Lee 2010] provides hardware-supported compartments to support secure execution environment for software modules.

Capability-based addressing (or capability machines) [Woodruff et al. 2014; Carter et al. 1994; Fabry 1974] are similar in spirit to NIMP and aim to protect memory by forcing applications to use capabilities to access regions of virtual memory. Some capability-based architectures went unused due to the amount of work involved in porting operating systems and applications to run on these architectures [Carter et al. 1994]. However, in other cases where more backward compatability is offered ([Woodruff et al. 2014]) the scheme trusts the OS to perform some critical operations, such as saving and restoring capability state on context switching which means that they could be vulnerable to malicious supervisor attacks. Furthermore, in works such as [Woodruff et al. 2014] capabilities are layered on top of virtual memory which means that a malicious supervisor can potentially violate security guarantees by manipulating virtual to physical translations (i.e. a page remapping attack). This leads to a significantly different threat model than NIMP, where various different modules (for example libraries) that make up an application are mutually untrusted and must be protected from one another.

A related industry development is the recent introduction by Intel of its SMEP/SMAP mechanism to protect some user-level pages from being executed and/or accessed by supervisor mode code [Intel 2014]. However, to support proper functionality, SMAP has to be toggled on and off to allow the OS to access user-space buffers. Trusting the OS to toggle SMAP removes any protection against a malicious OS, which is at the core of our threat model. We demonstrate in this paper that SMEP/SMAP can be implemented within NIMP framework.

There are a number of other works that provide isolated environments for trusted software modules [Lie et al. 2000; Suh et al. 2003; Lee et al. 2005; Dwoskin and Lee 2007; McKeen et al. 2013; Anati et al. 2013; Hoekstra et al. 2013; Boivie 2012; Owusu et al. 2013; Evtyushkin et al. 2014]. Haven [Baumann et al. 2014] demonstrated a system to achieve protected execution on unmodified binaries using SGX hardware support [McKeen et al. 2013]. Although the end result of these schemes may be similar to solutions such as NIMP, HyperWall and H-SVM, the threat models, TCB assumptions, as well as techniques and mechanisms for achieving security are different.

Mondrian memory protection [Witchel et al. 2002] is another work whose goals are similar to those of the NIMP architecture. In the Mondrian memory protection (MMP) architecture, permissions are stored in memory in a permissions table which is controlled by a priveleged supervisor domain. The supervisor then offers entities an API through which they can change permissions. One of the key differences between MMP and NIMP is the presense of this new supervisor domain in MMP. The goal of NIMP is to remove ambient authority from any supervisor-level software. Also, similarly to capability-based architectures, if MMP is layered on top of virtual memory then MMP cannot protect against page remapping attacks.

10. CONCLUSIONS

In this paper, we proposed NIMP — a new architecture to support non-inclusive permissions for the physical memory pages across different privilege levels of software. In contrast to the traditional designs where a higher-privileged software layer has all access rights to the pages of a lower-privileged layer, NIMP gives each layer its own minimal set of permissions sufficient to carry out its functionality. Changes to the page permissions are controlled by a set of rules and are enforced by the hardware — the OS and the hypervisor cannot change the page permissions if this request is not approved by the NIMP hardware. This essentially removes both the hypervisor and the guest OSes from the TCB and limits the TCB only to hardware and the loader. We demonstrate that such a permission management scheme retains all system functionality, while at the same time stopping many types of recent attacks that are due to the vulnerabilities either in the OS or in the hypervisor. We demonstrate that such protection is achieved with minimal performance loss, modest additional hardware and small changes to the OS and hypervisor code.

REFERENCES

I. Anati, S. Gueron, S. Johnson, and V. Scarlata. 2013. Innovative Technology for CPU Based Attestation and Sealing. In *Wkshp. on Hardware and Architectural Support for Security and Privacy, with ISCA'13*.

A. Azab, P. Ning, E. Sezer, and X. Zhang. 2009. HIMA: A Hypervisor-Based Integrity Measurement Agent. In *Proc. Annual Computer Security Applications Conference (ACSAC)*. 461–470.

A. Baumann, M. Peinado, and G. Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *Symposium on Operating Systems Design and Implementation*.

R. Boivie. 2012. SecureBlue++: CPU Support for Secure Execution. (2012).

Jan Cappaert, Nessim Kisserli, Dries Schellekens, and Bart Preneel. 2006. Self-encrypting code to protect against analysis and tampering. In *1st Benelux Workshop Inf. Syst. Security*.

Nicholas P Carter, Stephen W Keckler, and William J Dally. 1994. Hardware support for fast capability-based addressing. In *ACM SIGPLAN Notices*, Vol. 29. ACM, 319–327.

D. Champagne and R. Lee. 2010. Scalable Architectural Support for Trusted Software. In *Proceedings of HPCA*.

Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. 2005. Non-control-data Attacks Are Realistic Threats. In *Proceedings of the 14th Conference on USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA. http://dl.acm.org/citation.cfm?id=1251398.1251410

X. Chen, T. Garfinkel, E. Lewis, P. Subrahmanyam, D. Boneh, J. Dwoskin Dan, and R. Ports. 2008. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of ASPLOS*.

S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic. 2011. SecureME: A Hardware-Software Approach to Full System Security. In *Proc. International Conference on Supercomputing (ICS)*.

CVE-2009-1897 2009. CVE-2009-1897: NULL dereference and mmap of /dev/net/tun in Linux kernel allows privilege escalation. (2009). Available online: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3527.

CVE-2009-3527 2009. CVE-2009-3527: Race condition in Pipe (IPC) close in FreeBSD allows privilege escalation. (2009). Available online: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1897.

CVE-2010-4258 2010. CVE-2010-4258: do_exit does not properly handle a KERNEL_DS value allowing privilege escalation. (2010). Available online: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-4258.

CVE-2012-5513 2012. CVE-2012-5513: XENMEM_exchange handler does not properly check the memory address allowing privilege escalation. (2012). Available online: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-5513.

CVE Details 2015. CVE Details: The Ultimate Security Vulnerability Datasource. (2015). Accessed May 2015 at http://www.cvedetails.com/.

Ramon de C Valle. 2009. Linux sock_sendpage() NULL Pointer Dereference. (2009). Available online: http://packetstormsecurity.com/files/81212/Linux-sock_sendpage-NULL-Pointer-Dereference.html.

DOD 1985. *Trusted computer system evaluation criteria*. Technical Report 5200.28-STD. US Department of Defense. http://csrc.nist.gov/publications/history/dod85.pdf

L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev. 2012. Non-monopolizable Caches: Low-complexity Mitigation of Cache Side Channel Attacks. *ACM Transactions on Architecture and Code Optimization* 8, 4 (Jan. 2012).

J. Dwoskin and R. Lee. 2007. Hardware-rooted Trust for Secure Key Management and Transient Trust. In *Proceedings of CCS*.

edb1 2009. EDB-9477: sock_sendpage() Local Root Exploit in Linux. (2009). Available online: http://www.exploit-db.com/exploits/9477/.

edb2 2011. EDB-17391: DEC Alpha Linux ¡= 3.0 Local Root Exploit. (2011). Available online: http://www.exploit-db.com/exploits/17391/.

J. Elwell, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev. 2014. A Non-Inclusive Memory Permissions Architecture for Protecting Against Cross-Layer Attacks. In *Proc. International Symposium on High Performamce Computer Architecture (HPCA)*.

D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. Abu-Ghazaleh, and R. Riley. 2014. Iso-X: A Flexible Architecture for Hardware-Managed Isolated Execution. In *Proc. International Symposium on Microarchitecture (MICRO)*.

Robert S. Fabry. 1974. Capability-based addressing. *Commun. ACM* 17, 7 (1974), 403–412.

H. Fang, Y. Zhao, H. Zang, H. Huang, Y. Song, Y. Sun, and Z. Liu. 2010. VMGuard: An Integrity Monitoring System for Management Virtual Machines. In *Proc. of International Conference on Parallel and Distributed Systems (ICPADS)*.

T. Garfinkel and M. Rosenblum. 2003. A Virtual Machine Intersopection Based Architecture for Intrusion Detection. In *Proc. Network and Distributed Systems Security Symposium*. 191–206.

M. Hoekstra, R. Lal, P. Pappachan, C. Rozas, and V. Phegade. 2013. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Wkshp. on Hardware and Architectural Support for Security and Privacy, with ISCA'13*.

O. Hofmann, S. Kim, A. Dunn, M. Lee, and E. Witchel. 2013. InkTag: Secure Applications on an Untrusted Operating System. In *Proceedings of ASPLOS*.

Intel. 2014. Intel 64 and IA32 Architectures Software Developer's Manual. (2014). Accessed Feb. 2014 at http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

X. Jiang and X. Wang. 2007. Out-of-the-box Monitoring of VM-based High-Interaction Honeypots. In *Recent Advances in Intrusion Detection (RAID)*. 198–218.

X. Jiang, X. Wang, and D. Xu. 2007. Stealthy Malware Detection through VMM-based out-of-the-box Semantic View Reconstruction. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*.

V.P. Kemerlis, G. Portokalidis, and A.D. Keromytis. 2012. kGuard: lightweight kernel protection against return-to-user attacks. In *Proceedings of the 21st USENIX conference on Security symposium*. USENIX Association, 39–39.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 207–220. DOI:http://dx.doi.org/10.1145/1629575.1629596

Ruby B Lee, Peter CS Kwan, John P McGregor, Jeffrey Dwoskin, and Zhenghong Wang. 2005. Architecture for protecting critical secrets in microprocessors. In *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*. IEEE, 2–13.

D. Lie, M. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. 2000. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of ASPLOS*.

L. Litty, H. Lagar-Cavilla, and D. Lie. 2008. Hypervisor Support for Identifying Covertly Executing Binaries. In *Proc. 17th Usenix Security Symposium*.

MARSS 2013. MARSSx86: Micro-ARchitectural and System Simulator for x86-based Systems. (2013). http://marss86.org. Simulator source code and documentation.

F. McKeen, I. Alexandrovich, A. Berenzon, C.Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Wkshp. on Hardware and Architectural Support for Security and Privacy, with ISCA'13*.

E. Owusu, J. Guajardo, J. McCune, J. Newsome, A. Perrig, and A. Vadudevan. 2013. OASIS: On Achieving a Sanctuary for Integrity and Secrecy on Untrusted Platforms. In *Proceedings of CCS*.

Mathias Payer, Thomas Hartmann, and Thomas R Gross. 2012. Safe loading-a foundation for secure execution of untrusted programs. In *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 18–32.

B. Payne, M. Carbone, and W. Lee. 2007. Secure and Flexible Monitoring of Virtual Machines. In *Proc. of the Annual Computer Security Applications Conference*.

B. Payne, M. Carbone, M. Sharif, and W. Lee. 2008. Lares: An architecture for secure active monitoring using virtualization. In *Proc. IEEE Symposium on Security and Privacy*.

Ryan Riley, Xuxian Jiang, and Dongyan Xu. 2008. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *Recent Advances in Intrusion Detection (RAID)*. 1–20.

R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. 2004. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proc. of the 13th Usenix Security Symposium*.

Security Focus 2009. BID-36939: Microsoft Windows Kernel NULL Pointer Dereference Local Privilege Escalation Vulnerability. (2009). Available online: http://www.securityfocus.com/bid/36939.

Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. 2007. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. ACM, New York, NY, USA, 335–350. DOI:http://dx.doi.org/10.1145/1294261.1294294

M. Sharif, W. Lee, W. Cui, and A. Lanzi. 2009. Secure In-VM Monitoring using Hardware Virtualization. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*.

S.Jin, J.Ahn, S.Cha, and J.Huh. 2011. Architectural Support for Secure Virtualization under a Vulnerable Hypervisor. In *Proceedings of MICRO*.

G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. 2003. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *Proceedings of International Conference on Supercomputing*.

J. Szefer, E. Keller, R. Lee, and J. Rexford. 2011. Eliminating the Hypervisor Attack Surface for a More Secure Cloud. In *Proceedings of CCS*.

J. Szefer and R. Lee. 2012. Architectural Support for Hypervisor-Secure Virtualization. In *Proceedings of ASPLOS*.

TPM 2013. TPM Main Specification. (2013). Available online: http://www.trustedcomputinggroup.org/resources/tpm_main_specification visited Sept. 2013.

Z. Wang and R. Lee. 2007. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *Proc. International Symposium on Computer Architecture (ISCA)*.

Z. Wang and R. Lee. 2008. A Novel Cache Architecture with Enhanced Performance and Security. In *Proc. International Symposium on Microarchitecture (MICRO)*.

E. Witchel, J. Cates, and K. Asanović. 2002. Mondrian Memory Protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*. ACM, 304–316.

Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceeding of the 41st annual international symposium on Computer architecture*. IEEE Press, 457–468.

Y. Xia, Y. Lin, and H. Chen. 2013. Architecture Support for Guest-Transparent VM Protection from Untrusted Hypervisor and Physical Attacks. In *Proceedings of HPCA*.

Xilinx 2013. Xilinx 7 Series FPGAs Overview. (2013). Available online: http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf visited Sept. 2013.

F. Zhang, J. Chen, H. Chen, and B.Zang. 2011. Cloudvisor: Retrofitting Protection of Virtual Machines in Multi-Tenant Cloud with Nested Virtualization. In *Proceedings of SOSP*.