A Non-Inclusive Memory Permissions Architecture for Protection Against Cross-Layer Attacks

Jesse Elwell[1], Ryan Riley[2], Nael Abu-Ghazaleh[1], and Dmitry Ponomarev[1]
[1]State University of New York at Binghamton
[2]Qatar University
{jelwell,nael,dima}@cs.binghamton.edu, ryan.riley@qu.edu.qa

Abstract

Protecting modern computer systems and complex software stacks against the growing range of possible attacks is becoming increasingly difficult. The architecture of modern commodity systems allows attackers to subvert privileged system software often using a single exploit. Once the system is compromised, inclusive permissions used by current architectures and operating systems easily allow a compromised high-privileged software layer to perform arbitrary malicious activities, even on behalf of other software layers.

This paper presents a hardware-supported page permission scheme for the 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. Such a permission mechanism, implemented as part of a processor architecture, provides a common framework for defending against a range of recent attacks. We demonstrate that such a protection can be achieved with negligible performance overhead, low hardware complexity and minimal changes to the commodity OS and hypervisor code.

1. Introduction

Modern computing systems employ increasingly complex multi-layer software stacks that often include a hypervisor to support virtualization, multiple operating systems and user-level applications running on top of them. As the complexity and the number of lines of code in these underlying system software layers continue to increase, so does the number of security vulnerabilities that can be exploited by the 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. A recent study [39] analyzed and classified hypervisor vulnerabilities and attack surfaces. According to this study, 59 vulnerabilities have been identified in Xen and 38 in KVM as of July 2012. More alarmingly, about half of these vulnerabilities can lead to security breaches in terms of confidentiality, integrity and availability of the hypervisor. As another example, as of November 2013 the Linux kernel had 228 known security vulnerabilities [44].

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 that it manages. For example, the OS has complete access to user-level memory and the hypervisor has access to both OS and user-level pages, allowing full compromise from a single exploit. Importantly, each layer implicitly controls the permissions for less privileged resources and can alter them at will.

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 [28], 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 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 [53] or an OS attacks user-level processes (for example, by mapping a page into an address space of a malicious process).

To protect systems against cross-layer attacks, in this paper we propose NIMP (Non-Inclusive Memory Permissions) — a hardware supported framework that assigns each privilege layer only the minimum set of permissions necessary to carry out its tasks, and does not implicitly grant memory access to any privilege layer. Physical memory pages are assigned distinct permissions for each privilege layer, and 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.

In the remainder of the paper, we present a complete hardware/software design of NIMP to support the non-inclusive architecture of memory page permissions.We also demonstrate the security of NIMP against a range of attacks. We evaluate its performance overhead using a cycle-accurate simulator, and estimate the hardware complexity and timing of a reference implementation. 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 zero 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. We also show that the extra pressure put on the cache by these permission bits is negligible and

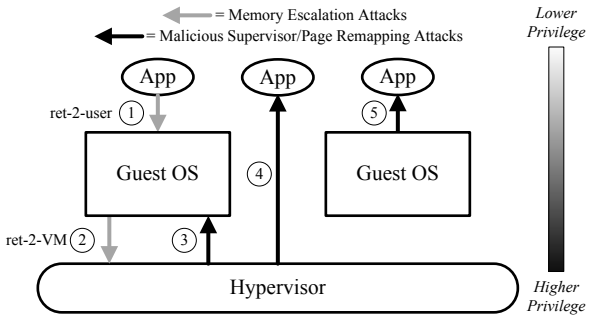


Figure 1: Cross-Layer Attacks

does not lead to decreased cache performance for the regular data. Furthermore, as the operations requiring changes to the page permission bits during program execution are relatively infrequent, the page zeroing overhead is also very small. Consequently, NIMP has a low performance overhead — around 1% on the average across SPEC 2006 benchmarks. In terms of hardware overhead, NIMP requires 28 bytes of on-chip Ternary Content-Addressable Memory (TCAM) storage for the permission rules, about 9% increase in the area of the TLBs, and an additional 64-bit register to point to the starting address of the protected memory region where the new permissions are securely stored.

The rest of the paper is organized as follows. Section 2 presents our threat model, followed by the overview of NIMP architecture in Section 3. Section 4 presents the implementation details of NIMP, followed by the description of how NIMP mitigates attacks considered in our threat model in Section 5. Section 6 evaluates performance impact and hardware complexity of NIMP. We discuss the related work in Section 7 and offer our concluding remarks in Section 8.

2. Threat Model & Assumptions

We assume that the Trusted Computing Base (TCB) of NIMP is limited only to the processor, physical memory, chipset, 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 our TCB is the loader to ensure that the application binaries themselves can not be tampered with.

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. This is consistent with the assumptions made by recent works such as HyperWall [53] and H-SVM [48]. We also note that if the physical memory is assumed to be untrusted, our proposal can be adjusted to that threat model by adding well-known techniques for memory integrity verification and encryption [51, 8, 10] to our design.

We do not protect against side-channels and covert channels; many previous works have addressed these issues [57, 58]. 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.

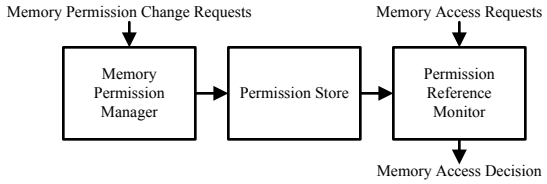


Figure 2: NIMP Design Overview

We are concerned with cross-layer memory attacks, where one software layer of the computing system 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.

- **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 get undesired access to user-level applications. Inclusive memory permissions used in conventional designs naturally allow such attacks to take place once a vulnerability in system software layers is found and exploited.
- **Page Remapping Attacks**. Here, the OS or 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 a malicious access.
- **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. This type of attack is usually paired with a software vulnerability in order to escalate privileges. As an example, 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 [28]. These attacks have affected all major operating systems and also targeted x86, ARM, DEC and PowerPC architectures [13, 14, 15, 17, 19, 20, 46]. One could also imagine a guest OS performing a similar attack against a hypervisor (*ret-2-VM)* [16].

Our system and the permission transition rules, as presented in this paper, are designed to prevent these attacks. The rule set can be extended to support additional attack flows and variations under the same common framework. The main purpose of this paper is to demonstrate that an approach based on non-inclusive permissions is a viable solution to create secure systems in a low-complexity manner.

3. NIMP Design Overview

Non-inclusive Memory Permissions (NIMP) are a lightweight form of mandatory access control enforced through hardware on the different layers of a virtualized system. An overview of the system is shown in Figure 2. Permissions are maintained at

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 1: Example Page Permissions

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 layer (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 a 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. In this section, we present an overview of NIMP and the design requirements for the different components necessary to implement it.

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. This means that a page could be readable by a user-level process, but not by the OS managing that process. NIMP allows for a 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 more permissions are included: the Shared (S) bit and the Page Table (PT) bit. The S bit signifies if a physical page in question 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 PT, and is used to identify writes to the page tables so that page mapping and unmapping events can be detected.

Table 1 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 the OS. Page 4 allows user-level execution, but no privileges at other levels. Such a page 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 remove 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 by a set of rules enforced by the processor.

Table 2 shows the complete set of rules for the NIMP system. The rules are built around two assumptions. First, once a page has its permissions assigned, those permissions should not change for the useful life 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 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.

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 from 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. This avoids code injection attacks. In conventional modern systems, this is accomplished by the NX bit.

Rules 5, 6 and 7 shown in Table 2 allow a page to go from writable to executable modes while retaining the contents of the pages. 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 zeroing out its contents. This is 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.

Two important aspects of this design need to be noted:

- Our rules do not permit user-level code to assign any permissions. Simply put, there is no need for applications to be concerned with assigning their own permissions. Instead, they should request the OS to assign them and then verify.
- 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.

		Initial Permissions				New Permissions					
		S/PT	Hyp.	OS	User	S/PT	Hyp.	OS	User		
Rule ID	Requester	S P	R W X	R W X	R W X	S P	R W X	R W X	R W X	Action	Note
1	Hypervisor	– –	– – –	– – –	– – –	* *	* * *	* * *	* * *	None	* = don't care
2	Hypervisor	* *	* * *	* * *	* * *	– –	– – –	– – –	– – –	Wipe Page	* = don't care
3	OS	– –	– – –	– – –	– – –	* *	– – –	* * *	* * *	None	* = don't care
4	OS	* *	– – –	* * *	* * *	– –	– – –	– – –	– – –	Wipe Page	* = don't care
5	Hypervisor	– –	– W –	– – –	– – –	– –	– – X	– – –	– – –	None	Hypervisor code page
6	OS	– –	– – –	– W –	– – –	– –	– – –	– – X	– – –	None	OS code page
7	OS	– –	– – –	– – –	– W –	– –	– – –	– – –	– – X	None	User-level code page

Table 2: Permission Assignment Rules to Mitigate Cross-Layer Attacks Considered in our Threat Model

3.3. PRM and Verification of Permissions

The Permission Reference Monitor (PRM) has two responsibilities. First, it ensures that a given memory access is allowed by the permissions specified for a given physical page. This is only a minor extension of the type of permission check performed by existing hardware.

The second responsibility of the PRM is to ensure that the permission specified for a given physical page is allowed by the requested memory access. To better understand this, 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 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 time-of-check race conditions.

In order to accomplish this, the instruction set is modified to allow memory access instructions to specify what permissions they explicitly do not want other privilege levels to have. This information can be encoded within memory instructions. When a memory instruction is executed, then the PRM performs verification and either allows or denies the request.

4. NIMP Implementation Details

In this section we describe our implementation of NIMP.

4.1. Permission Store

Permissions for each physical page are stored in a special area of memory called the permission store. The permission store is made of individual page permissions 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 (hypervisor, OS and user-level). In addition, the shared (S) and page table (PT) bits are also included. We assume that 2 bytes are needed in memory to store each of the PS entries, with five bits reserved for future use. We include the five reserved bits to make each PS entry byte aligned. Figure 3a shows the PS layout for a single physical page.

The PS entries for each physical page are securely stored in a protected region of memory accessible only by hardware that is in charge of enforcing these new permissions. Neither the OS nor the hypervisor have a direct access to this memory 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 very modest. For example, for a system with 16GB of physical memory and 4KB pages, the PS entries for all pages require only 8MB of memory (2 bytes for each of the 4M pages in the system), which represents 0.05% of the total memory space. 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 the regular caches, 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. In order to facilitate this, 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.

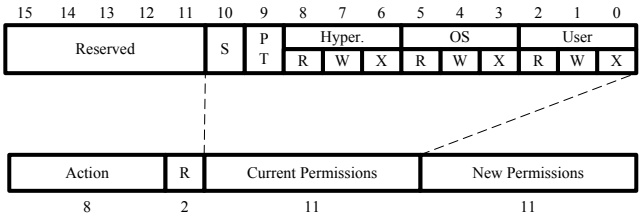
4.2. Memory Permission Manager

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 die 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,

Figure 3: Format of RD and PS Entries

(a) Format of a PS Entry



(b) Format of a RD Entry

the integrity of the BIOS is measured by the TPM [54] and then the rules are loaded into the RD.

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

- **Current permissions**: these 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**: this is necessary to distinguish between hypervisor, OS kernel or the user-level process. Two bits (we call them the R bits) are needed to differentiate between these three entities.
- **Action bits**: these specify any special actions needed to be performed on the page for the rule to be upheld, such as zeroing out the page, or encrypting it.

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 7 rules (that we use in our evaluation and that are shown in Table 2), the RD requires 28 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:

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

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 In a traditional system, a new physical page allocated for processes or virtual machines (VMs) becomes accessible after their corresponding physical frame numbers and protection bits are written into the page tables. In NIMP, the process of initial permission setup is different. We discuss it first for dynamic memory allocation and then for static memory allocation.

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.

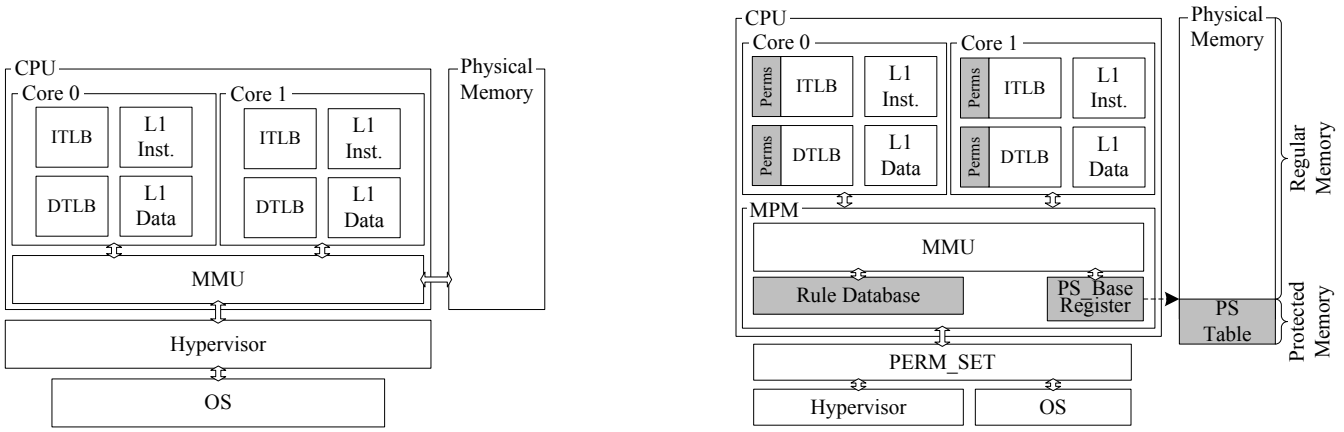
The PERM_SET instruction has the following format:

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. After that, the current set of permissions for the corresponding physical page (obtained from its PS entry) is read. The combination of the selected current PS entry and the new PS entry as specified in the instruction is then used 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 zeroing out the physical page) and then sets up the new permission bits both in the TLB and in the protected memory region. 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.

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 (e.g. by the loader). Then the OS would return from the system call and allow the loader to do its work. Once the loader has finished writing the code and read-only data to these pages, it makes another system call (such as *mprotect()*) to mark the code pages as executable. At this point the OS uses the PERM_SET instruction again to carry out this operation. This operation assumes that the loader is trustworthy, and that is why it is a part of our TCB.

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



(a) Traditional Hardware and System Software

(b) NIMP Hardware and System Software

Figure 4: Traditional Hardware vs. NIMP Hardware

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 NIMP 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 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.

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 prevent sharing, thus 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 [10, 51] in place of software-maintained ones, and storing these hardware PIDs as part of the PS entry, somewhat increasing the overhead of the NIMP logic. We leave the detailed quantification of this feature for future work, as this aspect is not central to the NIMP design.

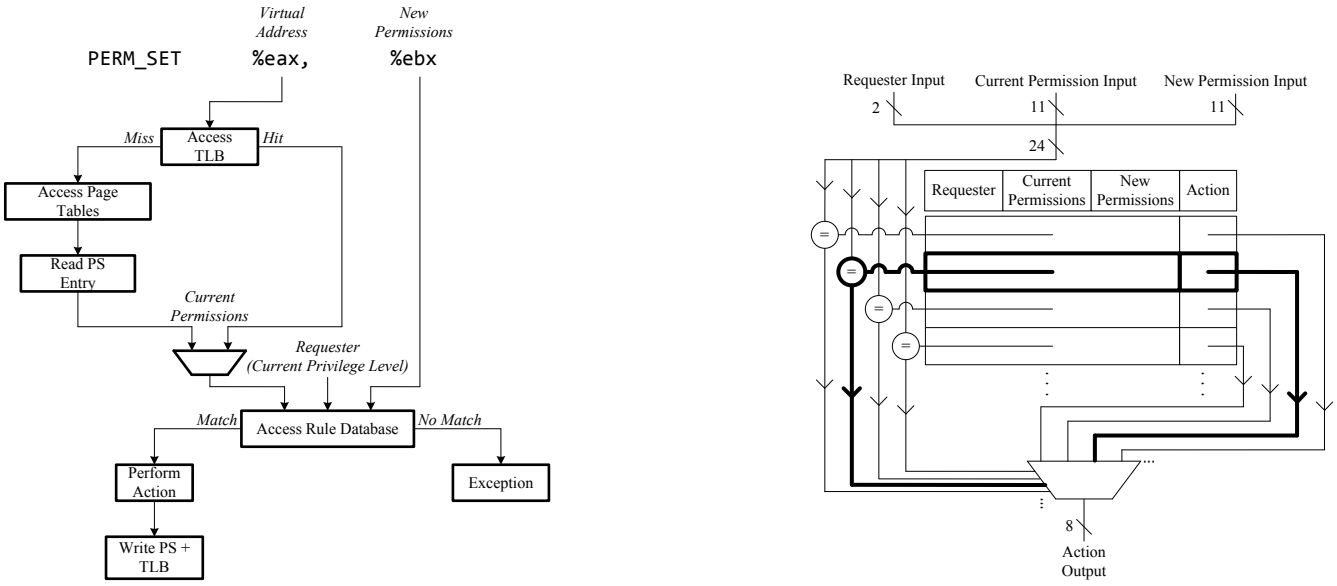
4.3. Permission Reference Monitor

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 responsibilities are similar to that of a traditional MMU, but some additional actions are also required.

When programs are compiled for the NIMP system, each load and store instruction is augmented with the permission that it expects other software layers to have for this particular section of data or code. This information can be communicated to the PRM hardware in several different ways, depending on the specifics of the instruction set used.

For example, for CISC ISAs (such as x86), which use variable-length instructions and opcodes, an additional byte can be added to every memory instruction to specify permissions of other layers. In this case, 4 bits can be used to check against the permissions of different privilege layers. Three bits specify the expected permissions of all other layers, as well as the expected S bit. We call these the Expected Permission (EP) bits.

Alternatively, if extending instructions is challenging due, for example, to the memory alignment issues (as would be the



(a) Memory Dataflow Triggered by PERM_SET Instruction

(b) Accessing the Rule Database

Figure 5: Activities Generated by the PERM_SET Instruction

case for fixed-length RISC ISAs) then the EP bits can either be conveyed through the opcodes (if they are available), or using compiler annotations, similar to what is used for static branch prediction.

Regardless of the implementation, when a LOAD or a STORE instruction enters the memory access stage of the pipeline, the PRM unit extracts the EP bits and compares them with the PS bits related to the OS/Hypervisor access rights. 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. In the figure we have assumed that an extra byte in the instruction is used to specify the EP bits. 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. Since the loader is trusted in NIMP, this ensures that the EP bits are tamper-proof, as is the rest of the program binary.

An alternative way to implement permission verification is to augment the context of each process with another register that holds the EP bits that are checked by all memory instructions. In this approach, a new instruction also needs to be added to the ISA to change the contents of this register as the EP bits change. Such a scheme avoids significant changes to the application binaries, but requires more changes to the library code and may have performance implications if the toggling of the new register occurs frequently.

4.4. 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, 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.
- **Secure Page Swapping** While NIMP prevents malicious supervisor software from getting access to the application memory pages, the same functionality also prevents 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, random nonce generated by the hardware and stored in the application's memory space, and the page's current permissions. The supervisor is unable to determine the 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. During a swap-in, the encrypted data can be read back from disk into memory, the correct permissions restored to the page, and the decryption performed by the hardware.
- **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.
- **Protecting from Multithreaded Attacks** A possible attack avenue exists when a malicious OS spawns a thread that was not requested by the program. In this case, the OS chooses the starting instruction address for that thread — for example, it can cause the movement of confidential data to an OS-readable page. To protect against such attacks, the act of setting the program counter for a newly spawned

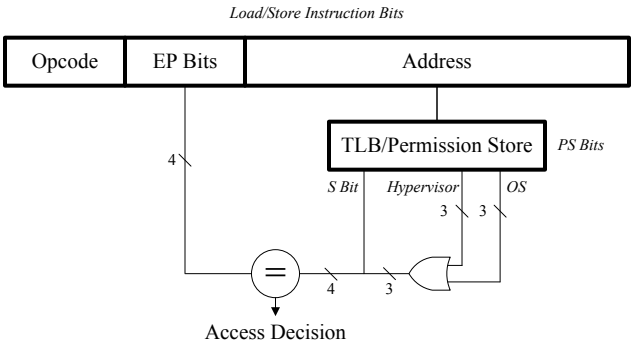


Figure 6: Permission Verification by the PRM

thread should not be performed without application or user-level library involvement.

5. Attack Mitigation

We now summarize how the new page permissions utilized by NIMP and the supporting infrastructure mitigate the various cross-layer attacks considered in our threat model.

5.1. Mitigating Malicious Supervisor Attacks

To protect against malicious supervisor attacks, some pages can be set up in a way similar to Page 3 shown in the example of Table 1. Such pages are configurable to be readable and writable only by the application layer. Therefore, the compromised supervisor layers will have no access to these pages. Furthermore, the supervisor layers will be incapable of granting themselves permissions for such pages and later accessing these pages with these permissions, because there is no rule in the RD to support such a transition. An 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.

5.2. Mitigating Page Remapping Attacks

Page Remapping Attacks can be performed in two styles. In the first attack variation, a target page is remapped within the same address space, but with a different set of permissions. For example, Page 3 shown in Table 1 can be unmapped and then a new page remapped in its place with a new set of permissions that now include the OS or the hypervisor's read/write permissions. While the current data on the page will be wiped out during this process, the new data written to this page by an unsuspecting application would then be accessible to the supervisor layers. The permission verification mechanism described in Section 4.3, however, prevents this attack. In this case, the application would detect the new permissions upon its first attempt to write, as the permission verification would fail.

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-sharable page is unmapped, its contents are zeroed out before a new mapping can be established. Mapping and unmapping events are detected when writes occur to pages marked with the PT bit, which is stored in each PS entry.

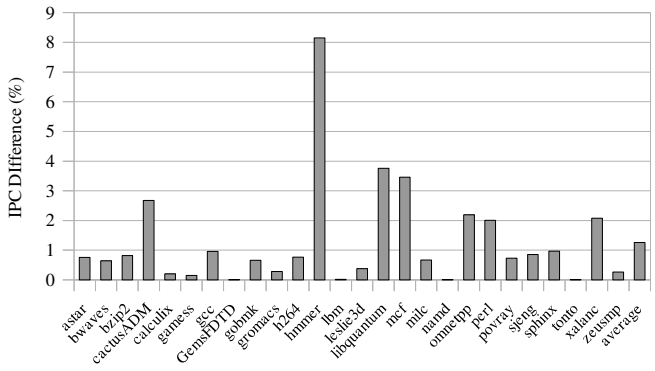


Figure 7: IPC Overhead of Caching Permissions

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 possible to use Rules 1,2,3 and 4 in Table 2 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.

6. Performance and Complexity Evaluation of NIMP

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

6.1. Performance Evaluation

There are two main sources of NIMP performance overhead due to the additional actions that need to be performed in this architecture for address translations and memory accesses. First, we need to access the permission bits that are stored separately from the regular page tables. While these bits are cached inside the regular TLBs and therefore do not impact the memory access latency on TLB hits, additional memory access is required on a TLB miss following the page table walk. Second, some of our new permission changing rules dictate that the corresponding physical pages are zeroed out by the hardware before the new permission settings can take effect — this also adds execution cycles.

For estimating the impact of the extra delays due to accessing the new permission bits, we used MARSSx86 [1] — a full-system x86-64 simulator. We modified the TLB miss handling code to perform the regular cache lookups and replacements for the PS data to estimate the impact of PS data on the number of cycles needed to execute applications. Since the permission bits are also stored in the cache hierarchy (and therefore, a DRAM access is not always needed to access them), we accurately simulated this impact as well. Our processor configuration is shown in Table 3.

Parameter	Configuration
Datapath	4-way superscalar, 128-entry ROB, 64-entry Issue Queue, 96-entry LSQ
Inst. & Data TLBs	64-entry, Fully Associative
L1 I & D Caches	32 KB, 8-way, 64B line, 2 cycles
L2 Unified Cache	256KB, 8-way, 64B line, 10 cycles
L3 Unified Cache	8MB, 16-way, 64B line, 40 cycles
Memory latency	150 cycles

Table 3: Configuration of the Simulated x86-64 Processor

We used 26 SPEC CPU2006 [50] benchmarks for our evaluations. For each benchmark, we simulated 2 billion instructions.

Figure 7 shows the decrease in committed instructions per cycle (IPC) for each of the simulated benchmarks due to the delay of accessing permission bits on TLB misses. The largest performance loss of 8.1% was observed for the *hmmer* benchmark, followed by 3.8% for *libquantum*. The rest of the benchmarks experience less than 2.5% degradation, with the average of 1.25%.

Further investigating cache performance, Figure 8 shows the hit rates to the various levels of caches for the PS data. On average 95% of the requests for PS data are satisfied from the L1 cache and 80% of the L1 misses are satisfied from the L2 cache. This means that the number of accesses to the L3 cache and main memory is very small. Figure 9 shows the impact of the PS bits on the cache miss rate of regular accesses. As seen from the figure, in general there is a slight increase in miss rate for the L1 cache due to contention with the PS bits.

Next, we estimate the performance overhead of zeroing out the page contents in hardware, as dictated by the permission change rules. To assess the frequency of operations requiring page permission changes, we profiled the Linux kernel using the built-in *ftrace* [41] 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. Specifically, we evaluated the Chromium web browser, the process of booting a VM with VirtualBox, and opening a spreadsheet in LibreOffice. We chose these interactive applications to illustrate this aspect of NIMP performance because they have a much higher number of page permission changing requests compared to the CPU-bound SPEC programs, and therefore provide close to a worst-case scenario. The SPEC benchmarks perform few system calls during their execution, so we observed near-zero performance impact for them.

For the three selected 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 [53] assumed 512 cycles to wipe off the entire page; under those assumptions our overheads will be even smaller).

Table 4 shows these statistics and performance overhead.

Application	Changes Per Second	Cycle Overhead
VirtualBox	2765	0.4%
Chromium	2973	0.4%
LibreOffice	8608	1.2%

Table 4: Cycle Overhead of Zeroing Pages

The first column shows the number of permission changing operations per second for these applications, and the second shows the impact on the total number of cycles assuming a 3GHz processor (e.g. 3B cycles per second of normal execution). As can be seen from these results, 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.

6.2. 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 [22]. 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.

We also evaluated the impact of slightly widening TLB entries to support storing of the new permission bits inside the TLB. In terms of the TLB area, this modification increases it by about 9%, while the impact on the TLB access delay is only 1%. 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 do not change.

7. Related Work

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

7.1. 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 [43] proposes a hypervisor to secure VM interactions, VMGuard [21] is a technique for protecting the management VM in Xen. A number of efforts use introspection to identify the presence of malicious code [4, 47, 33, 23, 27, 26, 37, 38]. Other works use the hypervisor to protect the guest OSes [45, 29, 40].

Several attacks on hypervisors have been demonstrated [42, 62, 30, 3, 59, 11, 12]. Therefore, some schemes protecting the hypervisor from attacks have been proposed [56, 5, 55].

Overshadow [9] protects applications from the compromised OS by presenting the OS with an encrypted view of physical memory, while NoHype [52] eliminates the hyper-

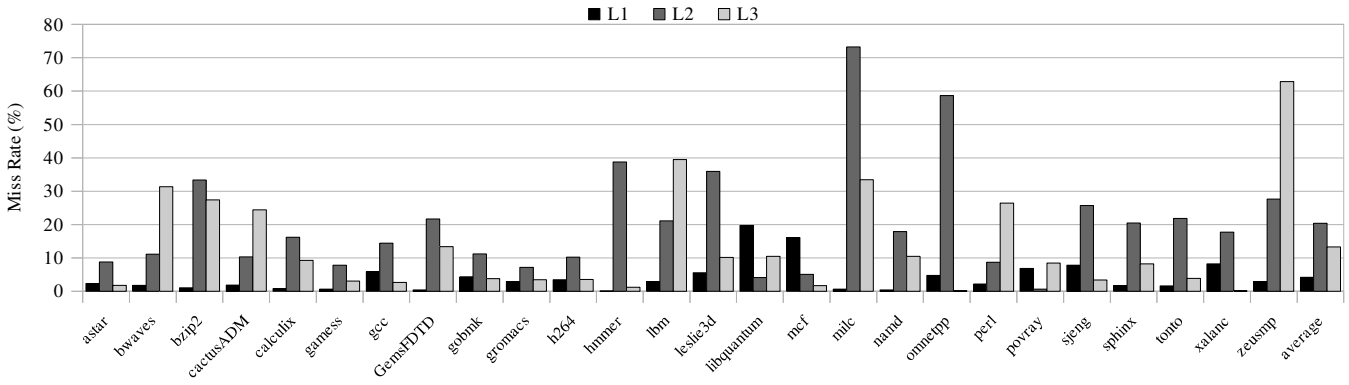


Figure 8: Cache Miss Rate for the PS Bits

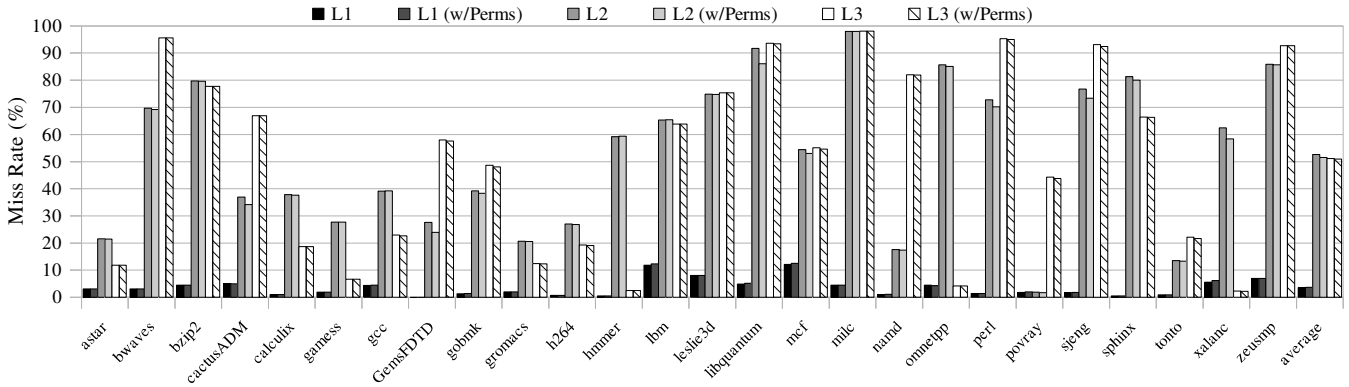


Figure 9: Impact on the Cache Miss Rate for Regular Data Accesses

visor layer altogether. Cloudvisor [61] uses a small security monitor below the hypervisor, using nested virtualization. The security of this approach depends on the integrity of the new monitor. Inktag [25] introduces paraverification mechanisms, forcing an OS to do additional computations to make it less complex for the hypervisor to verify its behavior.

Some previous efforts enhance traditional operating systems with mandatory access control (MAC) mechanisms. For example, SELinux [34] implements a policy-driven MAC framework for Linux kernel. SE Android [49] prototypes SELinux for Android's Linux kernel. FlaskDroid [7] builds on SE Android by providing MAC simultaneously on both Android's middleware and kernel layers.

7.2. Hardware-Supported Approaches

To address the limitations of software approaches, several hardware-supported schemes have been recently proposed. HyperWall [53] and H-SVM [48] are perhaps the closest proposals in spirit to NIMP. We describe them below and contrast our work to them in some detail.

The threat model of HyperWall [53] is to protect guest VMs from a malicious hypervisor. This is realized by the new CIP tables, accessible only to the hardware, that store the hypervisor and DMA access rights to physical memory pages. Every memory access by the hypervisor is checked by the hardware through CIP tables and the access is disallowed if the page permissions stored in the CIP tables are violated. This philosophy is similar to NIMP's approach of maintaining PS bits in the protected memory. However, there are some important differences between NIMP and HyperWall. First, HyperWall's threat model only assumes untrusted hypervisor, but the guest operating systems running inside the VMs are assumed to be trusted. NIMP's threat model is more restrictive, as we assume that neither the hypervisor, nor the guest OSes can be trusted and we protect applications from possible attacks that can be initiated by any level in the system software stack. Second, 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. Since NIMP accomplishes this using a small number of pre-specified generic rules with one set of rules shared by the entire system, the applications are isolated from the permission management tasks — the OS still manages the permissions, but the NIMP hardware just verifies that the page permission changes are allowed. In contrast, CIP tables in the HyperWall architecture require customer specifications to be set up. Third, page permissions in the CIP tables are set and checked in HyperWall at the granularity of VMs, while NIMP allows more fine-grained management at the application level.

Similar to NIMP, H-SVM [48] 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), our NIMP design only protects the page permission bits, which are decoupled from the main page table structures and are indexed by the physical page number. Since the protection in H-SVM is implemented for virtual pages, it is unclear how the situation would be handled where two virtual pages map the same physical page, but with different access rights. This may allow some attacks to still be possible. This is the reason why the NIMP design (as well as HyperWall solution) provide protection directly for physical pages. Finally, H-SVM protections are only implemented at the granularity of individual VMs. The NIMP design, however, directly applies to both virtualized systems and systems without virtualization, using the same unified framework to protect against various cross-layer attacks considered in our threat model.

HyperCoffer [60] is built on the idea of placing a new layer — the 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 security of HyperCoffer depends on the integrity of the VM-Shim code. NIMP, in contrast, does not rely on any software layers to be secure. Bastion [8] provides hardware-supported compartments to support secure execution environment for software modules. However, Bastion still relies on the security of the modified hypervisor to accomplish these goals. SecureMe [10] uses a security model where it automatically protects the entire address space of applications from a compromised OS using memory cloaking (a technique that was also used in Overshadow [9]), permission paging and system call protection. SecureMe relies on a small trusted hypervisor as part of its TCB.

There are also a number of other works that provide isolated environments for trusted software modules [32, 51, 31, 18, 35, 2, 24, 6, 36]. 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.

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. 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.

8. 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 demonstrated 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 showed that such protection is achieved with minimal performance loss, modest additional hardware and small changes to the OS and hypervisor code.

Acknowledgments

This publication was made possible by the support of the NPRP grant 4-1593-1-260 from the Qatar National Research Fund. The statements made herein are solely the responsibility of the authors.

References

[1] Marssx86: Micro-architectural and system simulator for x86-based systems, 2013. http://marss86.org. Simulator source code and documentation.

[2] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for cpu based attestation and sealing. In *Wkshp. on Hardware and Architectural Support for Security and Privacy, with ISCA'13*, 2013.

[3] Anonymous. Xbox 360 Hypervisor Privilege Escalation Vulnerability, 2007. Available online: http://www.securityfocus.com/archive/1/461489.

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

[5] A. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. Skalsky. HyperSentry: Enabling Stealthy in-context Measurement of Hypervisor Integrity. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2010.

[6] R. Boivie. Secureblue++: Cpu support for secure execution, 2012.

[7] S. Bugiel, S. Heuser, and A. Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *Proceedings of USENUX Security Symposium*, 2013.

[8] D. Champagne and R. Lee. Scalable architectural support for trusted software. In *Proceedings of HPCA*, 2010.

[9] X. Chen, T. Garfinkel, E. Lewis, P. Subrahmanyam, D. Boneh, J. D. Dan, and R. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of ASPLOS*, 2008.

[10] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic. Secureme: A hardware-software approach to full system security. In *Proc. International Conference on Supercomputing (ICS)*, June 2011.

[11] CVE-2007-4993: Xen guest root can escape to domain 0 through pygrub, 2007. Available online: http://cve.mitre.org/cgibin/cvename.cgi?name=CVE-2007-4993.

[12] CVE-2007-5497: Vulnerability in XenServer could result in privilege escalation and arbitrary code execution, 2007. Available online:http://support.citrix.com/article/CTX118766.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] R. de C Valle. Linux sock_sendpage() null pointer dereference, 2009. Available online: http://packetstormsecurity.com/files/81212/Linux-sock_sendpage-NULL-Pointer-Dereference.html.

[18] J. Dwoskin and R. Lee. Hardware-rooted trust for secure key management and transient trust. In *Proceedings of CCS*, 2007.

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

[20] EDB-17391: DEC Alpha Linux <= 3.0 Local Root Exploit, 2011. Available online: http://www.exploit-db.com/exploits/17391/.

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

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

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

[24] M. Hoekstra, R. Lal, P. Pappachan, C. Rozas, and V. Phegade. Using innovative instructions to create trustworthy software solutions. In *Wkshp. on Hardware and Architectural Support for Security and Privacy, with ISCA'13*, 2013.

[25] O. Hofmann, S. Kim, A. Dunn, M. Lee, and E. Witchel. Inktag: Secure applications on an untrusted operating system. In *Proceedings of ASPLOS*, 2013.

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

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

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

[29] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[30] K. Kortchinsky. Hacking 3D (and Breaking out of VMWare). In *BlackHat USA*, 2009.

[31] R. B. Lee, P. C. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang. Architecture for protecting critical secrets in microprocessors. In *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*, pages 2–13. IEEE, 2005.

[32] D. Lie, M. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of ASPLOS*, 2000.

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

[34] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of USENIX Annual Technical Conference*, 2001.

[35] F. McKeen, I. Alexandrovich, A. Berenzon, C.Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar. Innovative instructions and software model for isolated execution. In *Wkshp. on Hardware and Architectural Support for Security and Privacy, with ISCA'13*, 2013.

[36] E. Owusu, J. Guajardo, J. McCune, J. Newsome, A. Perrig, and A. Vadudevan. Oasis: On achieving a sanctuary for integrity and secrecy on untrusted platforms. In *Proceedings of CCS*, 2013.

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

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

[39] D. Perez-Botero, J. Szefer, and R. B. Lee. Characterizing hypervisor vulnerabilities in cloud computing servers. In *Proceedings of the 2013 international workshop on Security in cloud computing*, pages 3–10. ACM, 2013.

[40] R. Riley, X. Jiang, and D. Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *Recent Advances in Intrusion Detection (RAID)*, pages 1–20, 2008.

[41] S. Rostedt. ftrace – Function Tracer, 2008. https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[42] J. Rutkowska. Introducing the Blue Pill, 2006. Available Online:http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html.

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

[44] Cve details: The ultimate security vulnerability datasource, 2013. Accessed Nov. 2013 at http://www.cve-details.com/vulnerability-list.

[45] A. Seshadri, M. Luk, N. Qu, and A. Perrig. 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, pages 335–350, New York, NY, USA, 2007. ACM.

[46] BID-36939: Microsoft Windows Kernel NULL Pointer Dereference Local Privilege Escalation Vulnerability, 2009. Available online: http://www.securityfocus.com/bid/36939.

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

[48] S.Jin, J.Ahn, S.Cha, and J.Huh. Architectural support for secure virtualization under a vulnerable hypervisor. In *Proceedings of MICRO*, 2011.

[49] S. Smalley and R. Craig. Security enhanced (se) android: Bringing flexible mac to android. In *Proceedings of NDSS*, 2013.

[50] C. D. Spradling. Spec cpu2006 benchmark tools. *SIGARCH Comput. Archit. News*, 35(1):130–134, 2007.

[51] G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Aegis: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of International Conference on Supercomputing*, 2003.

[52] J. Szefer, E. Keller, R. Lee, and J. Rexford. Eliminating the hypervisor attack surface for a more secure cloud. In *Proceedings of CCS*, 2011.

[53] J. Szefer and R. Lee. Architectural support for hypervisor-secure virtualization. In *Proceedings of ASPLOS*, 2012.

[54] TPM Main Specification. Available online: http://www.trustedcomputinggroup.org/resources/tpm_main_specification visited Sept. 2013.

[55] J. Wang, A. Stavrou, and A. Ghosh. HyperCheck: A Hardware-Assisted Integrity Monitor. In *Proc. Recent Advances in Intrusion Detection (RAID)*, pages 158–177, 2010.

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

[57] Z. Wang and R. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proc. International Symposium on Computer Architecture (ISCA)*, June 2007.

[58] Z. Wang and R. Lee. A novel cache architecture with enhanced performance and security. In *Proc. International Symposium on Microarchitecture (MICRO)*, Dec. 2008.

[59] R. Wojtczuk. Subverting the Xen hypervisor. In *BlackHat USA*, 2008.

[60] Y. Xia, Y. Lin, and H. Chen. Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks. In *Proceedings of HPCA*, 2013.

[61] F. Zhang, J. Chen, H. Chen, and B.Zang. Cloudvisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of SOSP*, 2011.

[62] D. Zovi. Hardware Virtualization Based Rootkits. In *BlackHat USA, 2006*, 2006. Available Online: http://blackhat.com/presentations/bh-usa-06/BH-US-06-Zovi.pdf.