Hardening Extended Memory Access Control Schemes with Self-Verified Address Spaces

Jesse Elwell[*]
Vencore Labs
Basking Ridge, NJ
jelwell@vencorelabs.com

Dmitry Evtyushkin[†]
College of William and Mary
Department of Computer Science
Williamsburg, VA
devtyushkin@wm.edu

Dmitry Ponomarev
Binghamton University
Department of Computer Science
Binghamton, NY
dponomar@binghamton.edu

Nael Abu-Ghazaleh
University of California at Riverside
Department of Computer Science
Department of Electrical and Computer Engineering
Riverside, CA
nael@cs.usr.edu

Ryan Riley[‡]
Carnegie Mellon University Qatar
Computer Science Department
Doha, Qatar
rileyrd@cmu.edu

ABSTRACT

In this paper we revisit the security properties of extended access control schemes that are used to protect application secrets from untrusted system software. We demonstrate the vulnerability of several recent proposals to a class of attacks we call *mapping attacks*. We argue that protection from such attacks requires verification of the address space integrity and propose the concept of *self-verified address spaces (SVAS)*, where the applications themselves are made aware of the requested changes in the page mappings and are placed in charge of verifying them. SVAS equips an application with a customized verification model with several attractive functional and performance properties. We implemented the attacks and a complete prototype of SVAS in Linux and the QEMU emulator. Our results demonstrate that SVAS can prevent mapping attacks on extended access control systems with minimal performance overhead, hardware modifications and software complexity.

ACM Reference format:
Jesse Elwell, Dmitry Evtyushkin, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Ryan Riley. 2017. Hardening Extended Memory Access Control Schemes with Self-Verified Address Spaces. In *Proceedings of 2017 International Conference on Computer-Aided Design, Irvine, CA, November 13-16, 2017 (ICCAD 2017),* 8 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Computer systems rely on critical services provided by trusted system software layers. However, as the code bases and complexity of modern operating systems (OS) and hypervisors continues to grow, putting trust in these layers is no longer prudent [1, 2]. Attacks that exploit vulnerabilities in OS and hypervisor code to obtain privilege escalation and gain unauthorized access have been demonstrated [3–6]. It is therefore critical for secure operation to design systems in a way that protects application secrets even with untrusted system software. This problem recently received significant attention [7–18], with solutions grouped into two main categories:

- Isolated execution systems such as Intel's SGX [13, 14, 17]: in these systems, security critical components are placed into isolated compartments that are embedded in the application's address space. While these schemes provide strong security guarantees, they have a number of challenges associated with drastically different programming model, performance, flexibility, and hardware complexity.
- Extended Memory Access Control (EMAC) systems: a class of solutions that prevent privileged software layers (the OS and/or the Hypervisor) from accessing sensitive memory pages belonging to user processes by extending conventional memory access permissions [7–12, 15, 18].

While similar in goals, these two categories represent different design philosophies. EMAC solutions offer several attractive properties including low-overhead protection provided to user processes through natural extension of the permission structure present in virtual memory. Unlike isolation schemes, EMAC avoids the need of drastic changes to program's organization to separate security critical components into isolated entities. It also avoids the overhead of switching between the untrusted code and isolated components. In addition, restricted model of isolated environments and the inability to access libraries in isolated mode makes it challenging for programmers to create functional programs.

The EMAC solutions differ in where the access control is rooted: a trusted hypervisor [7, 8, 10, 15, 18], or the hardware [9, 11, 12]. Hardware-rooted designs, such as H-SVM [9], HyperWall [11] and NIMP [12] have an important advantage in that they do not add any extra software layers to the TCB.

In this paper, we revisit the security properties of hardware-rooted EMAC solutions. We show that these designs can be vulnerable to attacks that rely on manipulations of page mapping structures to gain access to critical data. We demonstrate these *mapping attacks (MAs)* and their application to defeat several existing hardware-rooted EMAC schemes. We describe the mechanics

[*]This work was performed while at Binghamton University
[†]This work was performed while at Binghamton University
[‡]This work was performed while at Qatar University, Doha, Qatar

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICCAD 2017, November 13-16, 2017, Irvine, CA
© 2017 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

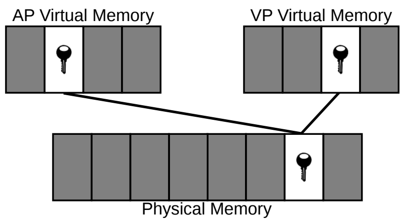

Figure 1: Double Mapping Attack: Memory Layout

and a complete implementation of MAs in Section 3. This is the first contribution of this paper. We also show that defending hardware-rooted EMAC systems against MAs requires the integrity of the address space mapping to be maintained to produce *Verified Address Spaces* (VAS). VAS can be supported in different ways based on when the verifications are triggered and where they are implemented; in this paper we introduce the concept of *self-verified address spaces (SVAS)* that allow efficient and flexible support of VAS without adding software to the TCB. The key idea of SVAS is to make the lower-privilege software layer (e.g. the victim application) itself aware that a page mapping change has been requested. The application is responsible for verifying the change and allowing or disallowing the new mappings. Involving applications in the verification process allows the system to make security-critical decisions in an application-specific way, and without being limited by the semantic gap.

2 THREAT MODEL & ASSUMPTIONS

We assume that the attacker completely controls the Highest Privileged Software Layer (HPSL) and the victim process is managed by the HPSL (and thus, the attacker). We also assume that the system protects physical memory from undesired direct accesses by the HPSL and limits the HPSL in the following ways.

- The HPSL cannot directly read or write the physical memory that holds the victim's critical data and the mechanism that enforces this restriction cannot be disabled by the HPSL.
- The HPSL cannot directly create new virtual memory aliases to expose the physical memory pages holding the victim's private sensitive data to other entities. We call this *double-mapping* of private pages.
- The HPSL cannot bypass the double-mapping restriction by unmapping a physical page from the victim's virtual address space and subsequently mapping it into a different virtual address space. We assume that when a physical page that is used for private data is unmapped from the victim's virtual address space, the page content is destroyed by some trusted entity.
- The HPSL cannot use DMA attacks [19, 20] to bypass existing memory protections.

Note that these protections are available in hardware-rooted EMAC models such as HyperWall [11] or NIMP [12].

We assume that the victim process or VM must share *some* non-sensitive pages with the HPSL and other entities to support high-performance inter-process communication and I/0. While the manipulations with the page tables are limited as detailed above, the attacker is allowed to dynamically insert pages into the victim's virtual address space or double-map non-private pages to support dynamic memory allocation and VM ballooning [21].

We assume that the CPU and physical memory are part of the Trusted Computing Base (TCB). As such, we do not protect against hardware-based attacks, such as probing physical memory or the memory bus. However, these physical attacks cannot occur remotely and well-known techniques to protect against them exist [7, 22–25]. We also do not protect against denial-of-service or side-channel attacks [26, 27] — a compromised operating system can always mount a denial-of-service attack. Finally, we do not protect applications from code reuse attacks [28, 29]. These attacks are rooted in a vulnerability within the application itself and they represent orthogonal attack vectors. Existing defenses for them can be used in conjunction with SVAS [30–33].

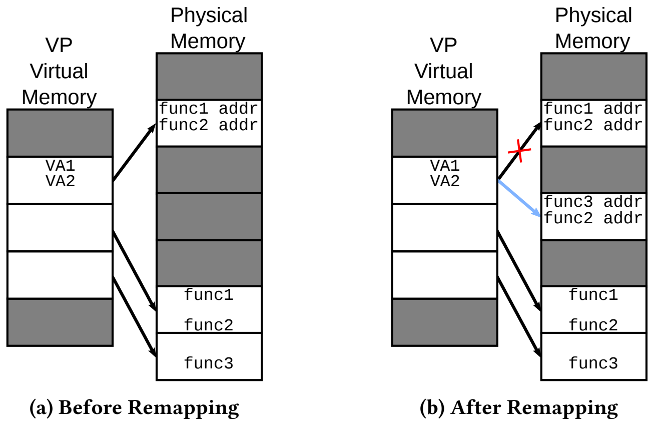

Figure 2: CFDA Mechanics

Finally, we assume that applications which require the highest level of security, including protection against all of the attacks described in this paper, must be either statically linked or linked at load-time to support shared libraries. The security benefits of SVAS do not extend to applications that use runtime linking (e.g. via `dlopen`). It is unreasonable to assume that a security-conscious user who does not even trust system software layers would tolerate security risks associated with runtime linking.

3 MAPPING ATTACKS ON EMAC SYSTEMS

To simplify the presentation, we assume a non-virtualized system, where the attacker controls the OS and the *victim process (VP)* runs as a user-level process, but similar attacks apply in a virtualized system. The attacker can create one or more *attack processes (APs)* to assist in performing the attack.

EMACs (described in Section 4) do not allow the attacker to directly access physical memory of a VP using kernel-level page mappings. Consequently, the attack needs to carry out these accesses at the VP's privilege level (user level in our prototype). The attacker can accomplish this by either adding code to the kernel or replacing existing code to establish virtual-to-physical page mappings for the user-level attack process (AP). Our prototype uses a system call from the AP for this purpose. We now describe three general attack vectors for the mapping attacks.

3.1 Double-Mapping Attack

With the kernel code in place, the attacker can perform a *double-mapping attack*. Recent works [11, 12] showed how simple double-mapping attacks can be defeated, we provide a brief description here only because the attack principles are used to create more advanced attacks described later. To perform a double-mapping attack, the attacker creates an alias in the address space of the AP so that the physical page holding the VP's sensitive information can be accessed using the AP's virtual addresses allowing the AP unconstrained access to the page. The virtual and physical memory

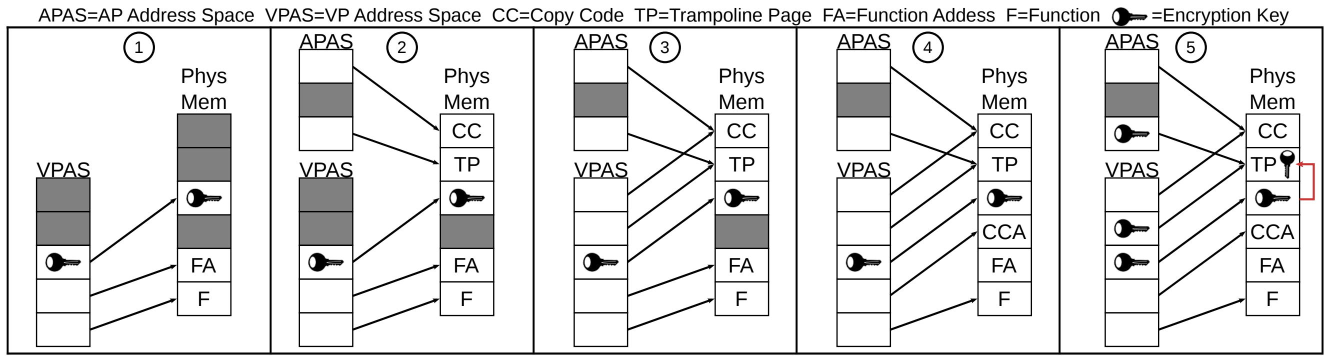
APAS=AP Address Space VPAS=VP Address Space CC=Copy Code TP=Trampoline Page FA=Function Addess F=Function =Encryption Key
APAS
VPAS
Phys Mem
CC
TP
CCA
FA
F

Figure 3: Injection-Based MA

layouts of the AP and the VP after the completion of the mapping operation are shown in Figure 1.

3.2 Control Flow Diversion Attack

Another category of MAs is *Control-Flow Diversion Attacks* (CFDAs). The goal of CFDAs is to alter the control flow of the VP. In addition to the malicious kernel code, the attacker exploits the presence of indirect control flow instructions in the VP, such as an indirect branch or a call.

CFDAs are carried out by strategically mapping a page containing data that is used by the VP to compute the target of the indirect call. This allows the attacker to change the data that the VP uses as the target. The data that the attacker changes via the mapping can be the target itself, or data used to derive the target, such as an index into a table of function pointers). Figure 2 depicts the first case.

Figure 2a shows the VP's virtual address space and the content of physical memory before the malicious mapping takes place. The attacker's goal is to get VP to call `func3` rather than `func1`. To accomplish this, the attacker creates a new physical page where the address of `func1` is replaced with the address of `func3`. The attacker then remaps the virtual page containing the function addresses to map to this new physical page. Figure 2b shows the VP's virtual address space after the remap operation (depicted as the blue arrow) has occurred.

3.3 Injection-Based MA

The double-mapping attack described above is mitigated by systems that offer low-privileged software layers the ability to specify sensitive pages and deny attempts to double-map them. However, some shared (double-mapped) pages in the VP must be allowed to support efficient communication with the OS, as described in Section 2. *Injection-based MAs* leverage this fact to bypass these protections.

The memory layout of the VP prior to the injection-based attack is shown in step (1) of Figure 3. This includes the VP's secret data (e.g. an encryption key), a function `F`, and a pointer to that function `FA`. The attack flow is the following: (2) The AP is augmented (by the attacker) with code that copies a data page from one position in the virtual memory space to another. In addition to this code, the AP also contains a page to be used as a trampoline page. (3) The attacker uses the double-mapping technique described in Section 3.1 to double-map the trampoline page and the page containing the copy code into the VP's address space. This creates the *data trampoline page* which is used to move data from the VP to the AP as part of the attack. (4) In this step, the attacker uses a CFDA

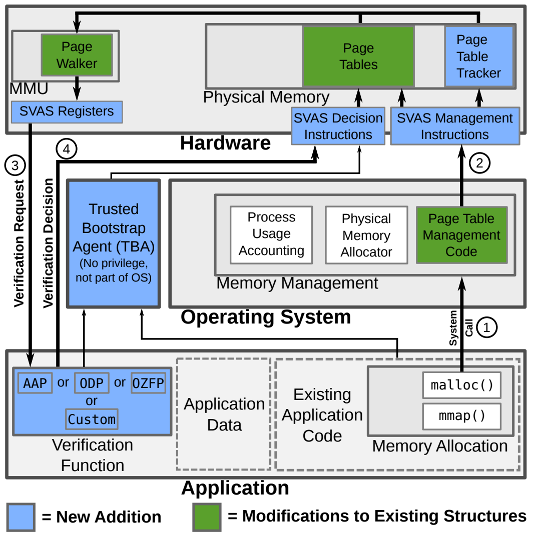

Figure 4: SVAS Overview

detailed in Section 3.2 to modify a function pointer in the VP to point to the copy code. (5) When the VP uses the function pointer, the control flow of the VP is diverted to the newly-added code page causing the VP to copy its sensitive data to the trampoline page that was created in step (3). At this time, the attacker can read the secret data from the trampoline page that is mapped into the AP's address space. This attack requires only the malicious kernel-level code and the user-level code to perform the copy.

4 IMPACT ON EXISTING DEFENSES

In this section, we evaluate the impact of MAs on the security of existing EMAC defenses in the context of our threat model. Specifically, we consider the hardware-rooted EMAC designs: HyperWall [11], NIMP [12], and H-SVM [9].

HyperWall [11] is designed to protect VMs from the hypervisor. The key idea is to rely on a new memory-resident table known as the Confidentiality and Integrity Protection (CIP) table. The CIP table has a single entry for each physical page in the system and is consulted by a modified memory management unit (MMU) to decide which physical pages can be accessed by the hypervisor. HyperWall allows VMs to specify which of their memory pages should be protected from hypervisor access during VM initialization, and the hardware applies these protections to the physical pages by making changes to the corresponding CIP table entries, as pages are mapped and unmapped from VMs.

The **NIMP** architecture [12] is similar to HyperWall in that the protections it offers are applied to physical memory pages. NIMP modifies memory permissions such that each software layer is presented with its own distinct set of permissions that are not necessarily hierarchical and inclusive. For example, NIMP allows memory pages to be configured in such a way that pages are accessible by an application, but not the OS.

H-SVM [9] also uses hardware to protect guest VMs from an untrusted hypervisor. The H-SVM architecture prohibits the hypervisor from directly modifying the nested page tables that map machine (i.e. host) physical memory to guest physical memory. Instead, the hypervisor must request updates to the nested page tables from the H-SVM hardware, which allows it to consult its page ownership tables to deny updates that would violate the memory isolation of guest VMs. Similar to the HyperWall architecture discussed above, some pages must be shared to allow efficient communication between the hypervisor and the guest VMs, presenting an attack opportunity.

HyperWall, NIMP and H-SVM provide mechanisms to detect and mitigate double-mapping attacks. HyperWall and H-SVM achieve this by not allowing a page which is already assigned to a VM to be assigned to another VM simultaneously, while NIMP explicitly supports a "shared" permission bit for each physical page, indicating whether the page can be mapped into multiple address spaces simultaneously.

Attacking EMAC Systems: While HyperWall, H-SVM and NIMP provide confidentiality of memory pages by zeroing them out when they are removed from an address space, none provide a complete integrity guarantee. As a result, an injection-based mapping attack against these systems can be launched as follows. First, the malicious hypervisor can remove a page from a VM's address space, causing it to be zeroed out by hardware. Second, the hypervisor can place new content of its choice on the now empty page. In the case of an injection-based mapping attack, the content consists of code which intentionally leaks data into the pages which are accessible by the hypervisor. Finally, the hypervisor can return the page to the same place in the victim's virtual address space. When the page is returned to the victim's address space, the protections will be reapplied, but the victim will still execute the code placed there by the hypervisor since the victim is unaware of this change. We note that during these operations the malicious hypervisor/OS would not make use of the support for swapping in these systems as the attack would be detected by the cryptographic protections applied to swapped pages.

Compared to isolated execution schemes, the EMAC solutions are less complex and require minimal (if any) changes to the programming model - a major advantage for practical adoption. Unfortunately, the current hardware-rooted EMAC schemes are vulnerable to MAs, as we demonstrated. To efficiently mitigate this threat, we propose the concept of *Self-Verified Address Spaces* (SVAS), where trusted applications themselves verify changes to their address spaces. Augmented with SVAS, hardware-rooted EMACs offer an attractive alternative to isolation for protecting against untrusted system software. We describe the details of SVAS in the next section.

5 SELF-VERIFIED ADDRESS SPACES

The general flow of handling mapping requests in SVAS is illustrated in Figure 4. The figure also highlights the responsibilities of the applications, system software layers, and hardware in handling these requests. Each application in SVAS is augmented with a Verification Function (VF) which is used to verify changes requested to the application's page mappings. The VFs are securely stored and are dispatched by SVAS when page mapping changes are detected. SVAS maintains some additional in-memory structures and adds several new instructions to implement its protections. The following subsections describe the SVAS design in more detail.

Table 1: Summary of SVAS Instructions

Management Instructions

Instruction	Description
`CRT_PT`	Creates a new page table
`DEST_PT`	Destroys an existing page table
`ADD_MAP`	Adds a new mapping to a page table
`RM_MAP`	Removes a mapping from a page table

Decision Instructions

Instruction	Description
`ACCEPT_MAP`	Accepts a new mapping
`REJECT_MAP`	Rejects a new mapping
`ACCEPT_IMM`	Accepts a new immutable mapping

5.1 SVAS Data Structures

In the SVAS system, each entry in a paging structure is augmented with a bit called the `REMAPPED` bit. This bit is added to every level of paging structure that can directly map a virtual page. The `REMAPPED` bit is set for a given Page Table Entry (PTE) by the hardware whenever that PTE is modified. The setting of the `REMAPPED` bit denotes that the virtual page mapped by this PTE has been remapped. The `REMAPPED` bit is consulted during address translation to ensure that the first access to a remapped page can be detected by the hardware. When such an access is detected, it triggers a verification event on behalf of the VP.

In addition to the `REMAPPED` bit, SVAS also uses two more bits for each PTE, the `LOCKED` bit and the `IMMUTABLE` bit. The usage of these two bits is discussed in Sections 5.5 and 5.6 respectively. Several unused bits already exist in an x86-64 paging structure, making it possible to support SVAS without increasing the page table size.

To accurately track remapping events, the system needs to be informed of all writes to the page tables. To this end, SVAS augments the CPU with a new in-memory data structure called the Page Table Tracker (PTT). The PTT is a bit-vector with one entry for each physical page, it is used to track which physical memory pages are used as page table frames. The PTT is indexed using the physical frame number (PFN) and is stored in a protected area of physical memory which is inaccessible by *any* software. Each entry in the PTT contains a single-bit flag, called the `IS_PT` bit, which denotes whether the physical page is currently in use as a page table frame. In addition, each entry contains a sufficient number of bits to denote what level of paging structure the physical page is being used for. We call these `PT_LEVEL` bits.

The PTT entries are consulted to support a number of security properties. First, the `IS_PT` bit is checked on each write to physical memory to deny writes to page table frames that do not use the special interface described below. These entries are also checked during hardware page walks to ensure that a physical page without the `IS_PT` bit set is not used during address translations. Furthermore, the `PT_LEVEL` bits are used to ensure that page tables that are incorrectly set up to either skip or duplicate a level of paging structure are not used.

5.2 Address Space Creation and Destruction

We now describe the new interface that is used in SVAS to create and destroy page tables and their individual entries. This interface is exposed using four new ISA instructions. These new instructions are summarized in Table 1.

The CRT_PT instruction is used to create a new page table. This instruction takes a single operand which corresponds to the physical address of the page table root for the newly-created address space (i.e. the value placed in the CR3 register for x86-64). When the CRT_PT instruction is executed, the hardware locates the PTT entry indexed by the instruction's operand, checks to ensure that the page is not already used as a page table frame, and sets the IS_PT and PT_LVL bits of this PTT entry. Once the PTT entry is set, this physical page can no longer be targeted by regular STORE instructions. At this point, the hardware zeroes out the physical page to avoid using any stale mappings. New mappings can only be added to the page using the SVAS interface. Finally, any existing TLB entries that could be used to target the physical page are invalidated.

The DEST_PT instruction is used to destroy a set of page tables. Similar to the CRT_PT instruction, the DEST_PT instruction takes the physical address of the page table root and clears the corresponding PTT entry. Before writing the PTT entry, the DEST_PT instruction must also walk any valid portions of the page table and remove any remaining entries in a manner very similar to the RM_MAP instruction described below.

5.3 Adding and Removing PTEs

To add and remove individual entries from the page tables created by the CRT_PT instruction, SVAS offers two instructions: ADD_MAP and RM_MAP.

At a high level, the ADD_MAP instruction inserts a new PTE specified by the entry operand into the page table rooted at pt_root. The virtual address of this entry is specified by the entry_vaddr operand. The first step taken by the ADD_MAP instruction is to translate the virtual address into a physical address using the TLB or a page walk. Note that this operation must take the pt_root operand to allow it to manipulate page tables other than the one currently pointed to by CR3. This is especially important for initial page table setup. Next, the instruction must ensure that the operation targets a page table frame to prevent the OS from using the ADD_MAP instruction to write arbitrary memory. This is accomplished by loading the PTT entry which corresponds to the page that is the target of the requested write operation and checking its IS_PT bit. In addition, the hardware must also ensure that the memory location specified by entry_vaddr contains all zeroes to prevent the OS from possibly overwriting any data already stored there. If either of these checks fail, an exception is generated.

The next step depends on the type of page being mapped, which is detected using the previously loaded PTT entry and the large page flag that is part of entry. If the page being mapped is itself another level of the paging structure (i.e. an internal node in the page table tree) the PTT entry of the page being mapped must first be checked to ensure that this page does not belong to another page table. Under SVAS, the sharing of page table frames that map user-level pages is explicitly disallowed by this check. Next, the IS_PT and PT_LEVEL bits are set to the proper values in the corresponding PTT entry. Additionally, the physical page is zeroed out for the reasons described above.

If the page being mapped is a leaf page (i.e. not part of the page tables) then the REMAPPED bit is set in entry before it is written into the page tables. We note that the leaf pages do not need to be zeroed as they may contain useful data. If a specific process wishes only to receive zero-filled pages, then it can ensure that this property is upheld during verification. Once these actions have been taken, the actual page table write is performed.

The RM_MAP instruction is used to destroy a mapping. Similar to ADD_MAP, its operands specify the virtual address of the entry to be removed and the root of the page table, which are used to translate from a virtual to physical address. The steps and checks performed by this instruction are similar to the ones describes above for the ADD_MAP instruction, so we omit the details due to space constraints.

5.4 Verification Functions

SVAS empowers applications to decide whether a new page mapping is acceptable using application-specific VFs. Table 2 shows a summary of different VFs utilized by our prototype which are explained below.

Accept All Pages (AAP): The simplest VF that has almost no performance impact contains only an ACCEPT_MAP instruction. Offering no additional security, such a VF can be used by applications that are oblivious to MAs and whose correct execution is not critical to other security-sensitive applications. This simple VF can be inserted into existing application binaries to achieve backward compatibility on a SVAS-enabled system or to allow the use of features such as runtime dynamically-linked libraries and just-in-time compilers.

Only Zero-Filled Pages (OZFP): Another variant of a VF is to only accept zero-filled pages, to allow the mapping of new pages to be used for stack and heap regions. The advantage of this function is that it offers a high level of security. However, it prohibits the application from receiving pages that contain useful content from the OS. Examples include pages of an mmap'ed file or a shared memory region that another process has already placed data into. This scheme only

Only Data Pages (ODP): To allow the use of mmap'ed files, this VF accepts any new data pages regardless of their contents. This VF checks only the page permissions so that pages that are not marked as executable are accepted, while executable pages are rejected. While ODP is more secure than AAP, control flow diversion attacks may still be possible. Namely, the application may accept a page whose data influences an indirect control flow instruction. However, applications are still able to defend themselves from injection-based attacks since any new code pages will be rejected.

5.5 Dispatching Verification Functions

When a page is accessed and its REMAPPED bit is set, the mapping has to be verified for correctness to flag a possible MA. For security and performance reasons, the VF is dispatched without OS involvement. To support such direct dispatching, the VF uses simple function call/return semantics. Specifically, the VF is placed at an architecture specific, well-defined virtual address which allows the hardware to dispatch the function in a manner very similar to the way that it handles a call instruction. Some additional actions must be taken by the hardware to provide the VF with the information that it needs to carry out the verification. In particular, the hardware takes the following steps:

- The hardware sets the LOCKED bit in the PTE that triggered the verification to avoid race conditions. When the LOCKED bit is set, the hardware will deny any changes to the PTE through checks performed as part of the ADD_MAP and RM_MAP instructions. Without the LOCKED bit, the OS could allow the application to verify a mapping, interrupt it just before it accepts that mapping, and replace the mapping with a new one causing the application to accept a mapping other than the one it verified.
- The hardware loads the PTE and virtual address which triggered the verification into the new SVAS registers provided specifically for this purpose.
- The hardware establishes a temporary TLB entry using the PTE in question to allow the application to inspect the content of the

Table 2: Summary of Verification Functions

Verification Function	Security	Flexibility	Overhead (Cycles)
AAP	None	High	0
ODP	Low	Medium	30
OZFP	High	Low	5296

page during verification. Regardless of the permissions specified in the PTE, the TLB entry is established such that only application reads will be allowed.

- The hardware saves the current stack pointer onto the VF's implementation defined stack base address and adjusts the stack pointer register to point to this stack. The VF stack is explained in more detail below.
- The hardware saves the address of the instruction which triggered the verification on the stack and jumps to the first instruction of the VF which resides at a well-known virtual address.

After the above steps are completed, the application's VF is executed, allowing the application to inspect a number of properties of the page. These include the page permissions, the PFN of the backing page, and even the contents of the page.

One important aspect of the VF's execution is that it requires it's own stack space to execute due to the fact that the VF needs to be executed when adding new (regular) stack pages to the application. If the VF does not use it's own stack and attempts to use the regular stack in response to such a request it is not hard to imagine that the VF itself would be recursively (and infinitely) triggered.

In the SVAS system, two new instructions — `ACCEPT_MAP` and `REJECT_MAP` — are used by applications to accept or reject a mapping respectively. Some common steps are taken by the hardware when either of these instructions are executed. First, the temporary TLB entry established during the verification dispatch is invalidated. This protects the application from using this TLB entry after it has rejected a mapping. The hardware also clears the `LOCKED` bit of the corresponding PTE.

When the `ACCEPT_MAP` instruction is executed, the hardware clears the `REMAPPED` bit in the PTE so that future accesses that use this mapping do not trigger verifications. Finally, the hardware returns from the VF by restoring the stack and instruction pointers to the value that was previously saved on the VF stack. In the case of the `REJECT_MAP` instruction, the hardware does not clear the `REMAPPED` bit, but instead raises an exception providing the virtual address which triggered the verification.

5.6 Secure Loading and VF Protection

Previous solutions addressed the problem of trusted loading in different ways. In HyperWall, hardware measures a VM's memory contents during initialization and provides clients with a signature of this measurement in addition to the relevant HyperWall state. The NIMP system relies on a trusted software module to perform a similar measurement of an application's initial state. NIMP releases an encryption key to the application if the application has been properly loaded. In both cases, the untrusted HPSL is allowed to place the VM/application into memory and the measurement agent verifies the integrity once the loading process has been completed.

SVAS requires a similar protocol. Specifically, the HPSL loads the application (including its VF) into memory and then passes control to the Trusted Bootstrap Agent (TBA). The TBA then accepts the application's code pages including the VF (and its stack pages). The TBA can be implemented either entirely in hardware in a similar fashion to HyperWall [11] or as an unprivileged but trusted software module like NIMP's KPIM [12]. One addition to the HyperWall model would be necessary to suit the needs of the TBA. Namely, rather than providing a signature as proof of correct loading the hardware needs to compare the measurement against a stored measurement and accept the application's pages if a match is found. This model requires extra hardware to measure the application's code and secure storage to maintain correct application and VF hashes, but eliminates the need for a trusted software module. The NIMP model requires the TBA to be loaded and protected during boot. Furthermore, to use the NIMP model the TBA must run in a CPU mode that does not trigger verifications to avoid jumping to unaccepted VFs. A variation of the `ACCEPT_MAP` instruction which explicitly specifies the mapping being accepted is required to support the TBA's operation in this model. This is necessary because `ACCEPT_MAP` is designed to respond to verification requests, but in this case no such requests are made. The final SVAS instruction — `ACCEPT_IMM` — is offered for this purpose, this instruction is only used by the TBA. For ease of implementation our prototype follows the model of NIMP's KPIM with the above mofifications.

It is also important to protect VFs from runtime attacks. For example, the HPSL may launch a MA against an application's VF to replace it with a VF that accepts all mappings. To protect VFs from such attacks, the new `IMMUTABLE` bit is added to each PTE. The `IMMUTABLE` bit is set by the TBA's use of the `ACCEPT_IMM` instruction.

The `IMMUTABLE` bit denotes that the PTE in which it is set is immutable. It is important to note that the system software can remove an immutable mapping to allow it to tear down applications. However, once an immutable mapping has been removed, a new mapping cannot be established in its place. This property is enforced by the implementations of the `ADD_MAP` and `RM_MAP` instructions. When `RM_MAP` targets a PTE with the `IMMUTABLE` bit set, the value of this bit is retained. A subsequent attempt to issue an `ADD_MAP` instruction targeting this PTE will find the PTE in a non-zero state, thus generating an exception. The final issue that must be considered when setting the `IMMUTABLE` bit is that it must be set in every level of the paging hierarchy.

5.7 Supporting Page Swapping

In the SVAS system, without special consideration, a page that was swapped out and later returned to memory would trigger a second validation. Since the application is unaware of the swapping activity, it may reject the required mapping changes. To alleviate this issue, SVAS must react to swapping events. When SVAS is combined with systems such as NIMP, HyperWall, and H-SVM (which already use hashing and encryption to address swapping problem), minimal additional work is required. In SVAS, once the hash has been verified and the page is decrypted, the CPU must additionally clear the corresponding `REMAPPED` bit in order not to trigger a validation upon the page's next access.

6 EVALUATING SVAS

To demonstrate that SVAS prevents all MAs described in Section 3, we implemented the SVAS logic and the new instructions within QEMU emulator [34] Specifically, we modified the full-system x86-64 code by adding the new SVAS instructions and appropriate checks of the PTT entries during hardware page walks. Lastly, we added code that calls the VF when a page with the `REMAPPED` bit set is accessed. We also modified the Linux kernel to make use of the new instructions during the relevant memory management tasks. We also implemented different VFs for the current prototype to mitigate various attack types.

Table 3: SVAS Instruction Memory Behavior

Instruction	Loads	Stores
CRT_PT	1	513
DEST_PT	Variable	Variable
ADD_MAP (I)	7	514
ADD_MAP (L)	6	1
RM_MAP	518	2

6.1 SVAS Performance

The performance overhead of SVAS comes from four sources: 1) Extra memory accesses due to the new mapping instructions; 2) Checking PTT entries on every memory access; 3) Memory accesses required after a TLB miss during hardware pagewalks; 4) Executing a VF when a remapped page is detected by the hardware.

Since the PTT is stored in memory, the SVAS mapping instructions require accessing memory to check and modify the PTT entries during mapping operations. In contrast, on a commodity system page table manipulations are performed by regular STORE instructions, thus requiring only a single memory write. Table 3 shows the number of word-sized (i.e. 8 bytes) memory accesses required by each of the SVAS mapping instructions. Two cases are shown for the ADD_MAP instruction depending on whether the mapping being added is an internal node (I) or a leaf node (L) in the page table tree. As seen from the results, the most expensive operation for these instructions is the checking/zeroing of memory pages which are being added or removed from the page table tree (required by CRT_PT, the internal variation of ADD_MAP, RM_MAP and DEST_PT). However, since this operation occurs relatively infrequently, its overhead is tolerable. The CRT_PT and DEST_PT instructions are used only once per application to create and destroy the application's page table. The specific number of ADD_MAP operations targeting internal page table nodes and RM_MAP depends on an application's memory requirements. However, when adding large portions of contiguous virtual address space are mapped, the ratio of added leaf nodes to added internal nodes is 512:1 (since each paging structure holds 512 entries). Note that the overheads required by the internal ADD_MAP and RM_MAP can be reduced by making use of large pages.

We instrumented our QEMU-based implementation of SVAS to track the frequency of each SVAS instruction's use during runtime. For the purposes of isolating specific activities, we implemented special instructions to log and reset these counters programmatically. Table 4 summarizes our results.

The first activity that we measured was booting the Linux kernel including all of the activities up to, but not including, the launch of the first user-level process (init). These results show that initializing something as complex as the OS kernel can be done using a relatively small number of mapping changes. Next, we evaluated the user-space boot process, including starting init, various system daemons, logging the user in, and starting a shell for the user. This experiment confirms the expected result that the ADD_MAP and RM_MAP are by far the most commonly used SVAS instructions. Furthermore, among the two categories of the ADD_MAP instruction, the version that does not require page zeroing is far more common than the one that does.

The next row in the table shows the number of instructions required to start a minimal application (i.e. only a return from main()). Interestingly, for the experiments detailed up to this point, the ratio of leaf nodes to internal nodes added to page tables is very far from the ideal case of 512:1. Booting the Linux kernel yields a ratio of about 11:1, the full boot to user space yields about 9:1 ratio, and running an empty application yields about 6:1 ratio. The reason is that the address spaces of the measured entities are fairly sparse and the virtual memory is discontiguous.

Table 4: Frequency of SVAS Instruction Use

Activity	CRT_PT	DEST_PT	ADD_MAP (I)	ADD_MAP (L)	RM_MAP
Boot Linux Kernel	1	0	526	6012	346
Full User space Boot	3791	3772	71459	627323	667569
Empty Application	1	1	60	352	391
Allocate Single Page	0	0	0	1	0
Free Single Page	0	0	0	0	1
Allocate 2MB	0	0	1	512	0
Free 2MB	0	0	0	0	513

The final set of experiments that we performed was to dynamically allocate pages during runtime. In the first experiment, we allocated a single 4KB page and in the second experiment we allocated a 2MB region. These results confirm our hypothesis that a single 4KB page can often be allocated by only adding a single new (leaf) mapping to the page tables. Furthermore, when a larger contiguous region is allocated (2MB in this case), the ratio of leaf to internal nodes indeed becomes 512:1.

Finally, we estimated the performance impact of SVAS on the SPEC2006 benchmarks. Since QEMU is a functional emulator, we used our prototype only to capture the frequency of SVAS instructions and verification events. We then executed the benchmarks on a real system with an Intel Core i7-4700MQ CPU running at 2.4GHz to obtain a baseline number of cycles for each benchmark. In addition to the SPEC benchmarks, we also ran several micro-benchmarks to evaluate the overhead of various activities, such as the execution of VFs (shown in Table 2), zeroing pages, and the calculations required by each of the instructions.

We used the results of the micro-benchmarks to calculate a conservative static cost for each of the SVAS instructions and VFs. The CRT_PT and ADD_MAP (internal) instructions require 4990 and 5070 respectively, while ADD_MAP (leaf) and RM_MAP require 22 and 5405 cycles respectively. In our prototype RM_MAP is used to clean up as many page table entries as possible leaving only entries with the IMMUTABLE bit set to be cleaned up by DEST_PT. In this case 26,295 cycles are required by DEST_PT to walk and clean up a page table assuming that the VF occupies a 2MB memory region made up of 4KB pages. In our experiments, none of the VFs required even this much space. For our analysis, we assumed the most expensive VF (i.e. OZFP) at a cost of 5,400 cycles to account for the VF as well as its dispatch. Finally, we combined the instruction counts reported by our prototype with the cost of each instruction to compute the number of additional cycles added to each benchmark. Figure 5 shows the cycle overheads expressed as a percentage of the baseline.

As can be seen from the results, most of the benchmarks incur less than 1% overhead under SVAS with the average being about 0.75%. There are a number of factors that contribute to this low overhead. First, some applications use a small and fairly static number of virtual pages (e.g. *gamess, gromacs, povray*). Second, the costs for some applications which use more memory are hidden by the large amount of computation that they perform using that memory (e.g. *calculix*). The highest overhead incurred by the SVAS system is that of *milc* — at around 10%. This benchmark allocates and frees memory frequently and the allocations are large enough to cause a high number of internal ADD_MAP instructions to be executed in addition to the required RM_MAP instructions. We note that this overhead could be reduced significantly by making use of large pages.

6.2 SVAS Complexity

The hardware additions and modifications required to realize SVAS are minimal. A section of physical memory must be reserved to

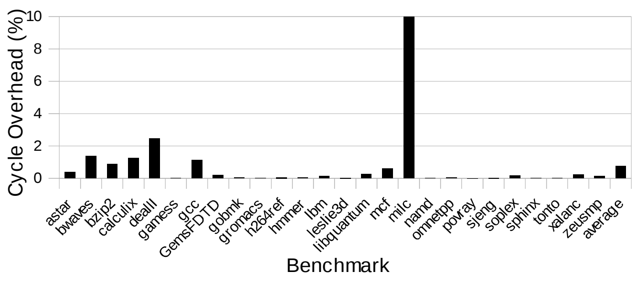

Figure 5: Performance Overhead of SVAS

hold the PTT. Two new user-readable registers are required to communicate the modified PTE and virtual address information to applications during verification. Seven new instructions must be added to the ISA, requiring some minimal additional logic to decode these instructions. Finally, the hardware page walker must be modified to consult PTT entries and `REMAPPED` bits during hardware page walks.

SVAS requires modest modifications to the OS kernel. For our prototype, we modified the source code of version 3.2.63 of the Linux kernel. Our changes to the kernel were implemented in three main steps. First, we added the malicious kernel code described in Section 3. Second, we used inline assembly to call the new SVAS instructions within the Linux kernel after these instructions were implemented in QEMU. Third, we modified the kernel to correctly use the new instructions. In total, these changes were implemented by inserting 235 lines of code to the Linux kernel and removing 13 lines.

7 CONCLUDING REMARKS

We demonstrated that previousy proposed hardware-based EMAC designs are vulnerable to advanced attacks that are based on manipulations of page mapping structures which are controlled by the OS. We developed and implemented three examples of such mapping attacks. To mitigate these attacks, we proposed the concept of self-verified address spaces (SVAS). The key idea is to provide applications with the capability to verify the requested changes to the mapping structures in a secure manner. We demonstrated that the protections offered by SVAS can be achieved with negligible performance impact, simple additional hardware, and minimal changes to application and system code. Augmenting hardware-rooted EMAC schemes with SVAS makes them an attractive alternative to isolated execution systems for protecting application secrets from untrusted system software.

ACKNOWLEDGMENTS

This paper was made possible by NPRP grant 8-1474-2-626 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors.

REFERENCES

[1] "Cve details: Linux kernel vulnerability trends over time," 2016. Accessed Mar. 2016 at https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.
[2] "Cve details: Xen vulnerability trends over time," 2016. Accessed Mar. 2016 at https://www.cvedetails.com/product/23463/XEN-XEN.html?vendor_id=6276.
[3] "CVE-2015-8539." Accessed April 2016 online at: http://www.cvedetails.com/cve/CVE-2015-8539/.
[4] "CVE-2014-6184." Accessed April 2016 online at: http://www.cvedetails.com/cve/CVE-2014-6184/.
[5] "CVE-2016-0723." Accessed April 2016 online at: http://www.cvedetails.com/cve/CVE-2016-0723/.
[6] "CVE-2015-4004." Accessed April 2016 online at: http://www.cvedetails.com/cve/CVE-2015-4004/.
[7] D. Champagne and R. Lee, "Scalable architectural support for trusted software," in *Proceedings of HPCA*, 2010.
[8] O. Hofmann, S. Kim, A. Dunn, M. Lee, and E. Witchel, "Inktag: Secure applications on an untrusted operating system," in *Proceedings of ASPLOS*, 2013.
[9] S.Jin, J.Ahn, S.Cha, and J.Huh, "Architectural support for secure virtualization under a vulnerable hypervisor," in *Proceedings of MICRO*, 2011.
[10] 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.
[11] J. Szefer and R. Lee, "Architectural support for hypervisor-secure virtualization," in *Proceedings of ASPLOS*, 2012.
[12] J. Elwell, R. Riley, N. Abu-Ghazaleh, D. Ponomarev, and I. Cervesato, "Rethinking memory permissions for protection against cross-layer attacks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 4, p. 56, 2015.
[13] 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.
[14] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. Abu Ghazaleh, and R. Riley, "Iso-X: A Flexible Architecture for Hardware-Managed Isolated Execution," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pp. 190–202, IEEE, 2014.
[15] J. Criswell, N. Dautenhahn, and V. Adve, "Virtual ghost: Protecting applications from hostile operating systems," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pp. 81–96, ACM, 2014.
[16] 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.
[17] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th USENIX Security Symposium (USENIX Security 16)*, (Austin, TX), USENIX Association, Aug. 2016.
[18] Y. Kwon, A. M. Dunn, M. Z. Lee, O. S. Hofmann, Y. Xu, and E. Witchel, "Sego: Pervasive trusted metadata for efficiently verified untrusted system services," 2016.
[19] P. Stewin and I. Bystrov, "Understanding dma malware," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 21–41, Springer, 2012.
[20] F. L. Sang, V. Nicomette, and Y. Deswarte, "I/o attacks in intel pc-based architectures and countermeasures," in *SysSec Workshop (SysSec), 2011 First*, pp. 19–26, IEEE, 2011.
[21] C. Waldspurger, "Memory Resource Management in VMWare ESX Server," in *Proc. of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.
[22] 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.
[23] G. Suh, C. O'Donnell, I. Sachdev, and S. Devadas, "Design and implementation of the aegis single-chip secure processor using physical random functions," in *Proceedings of ISCA*, 2003.
[24] "Intel Software Guard Extensions Programming Reference," 2016. Accessed April 2016 at https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.
[25] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving cost, performance, and security of memory encryption and authentication," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, (Washington, DC, USA), pp. 179–190, IEEE Computer Society, 2006.
[26] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games–bringing access-based cache attacks on aes to practice," in *Security and Privacy (SP), 2011 IEEE Symposium on*, pp. 490–505, IEEE, 2011.
[27] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE Symposium on Security and Privacy*, pp. 605–622, 2015.
[28] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of CCS*, pp. 552–61, ACM Press, Oct. 2007.
[29] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of ASIACCS*, pp. 30–40, ACM, 2011.
[30] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *22nd IEEE Symposium on High Performance Computer Architecture*, 2016.
[31] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent rop exploit mitigation using indirect branch tracing.," in *USENIX Security*, pp. 447–462, 2013.
[32] L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: a detection tool to defend against return-oriented programming attacks," in *Proceedings of ASIACCS*, pp. 40–51, ACM, 2011.
[33] L. Davi, A.-R. Sadeghi, and M. Winandy, "Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks," in *Proceedings of ACM STC*, pp. 49–54, ACM, 2009.
[34] F. Bellard, "Qemu, a fast and portable dynamic translator.," in *USENIX Annual Technical Conference, FREENIX Track*, pp. 41–46, 2005.