Secure Guest Virtual Machine Support in Apparition

Ethan Johnson
Department of Computer Science
University of Rochester
Rochester, New York, USA
ejohns48@cs.rochester.edu

Komail Dharsee
Department of Computer Science
University of Rochester
Rochester, New York, USA
kdharsee@cs.rochester.edu

John Criswell
Department of Computer Science
University of Rochester
Rochester, New York, USA
criswell@cs.rochester.edu

Abstract

Recent research utilizing Secure Virtual Architecture (SVA) has demonstrated that compiler-based virtual machines can protect applications from side-channel attacks launched by compromised operating system kernels. However, SVA provides no instructions for using hardware virtualization features such as Intel's Virtual Machine Extensions (VMX) and AMD's Secure Virtual Machine (SVM). Consequently, operating systems running on top of SVA cannot run guest operating systems using features such as Linux's Kernel Virtual Machine (KVM) and FreeBSD's bhyve.

This paper presents a set of new SVA instructions that allow an operating system kernel to configure and use the Intel VMX hardware features. Additionally, we use these new instructions to create *Shade*. Shade extends Apparition (an SVA-based system) to ensure that a compromised host operating system cannot use the new VMX virtual instructions to attack host applications (either directly or via page-fault and last-level-cache side-channel attacks).

CCS Concepts • **Security and privacy → Operating systems security**; **Trusted computing**; **Virtualization and security**;

Keywords hypervisors, hypervisor security, untrusted hypervisor, side channels, secure computer architectures, trusted execution environments, compiler-based virtual machines

ACM Reference Format:
Ethan Johnson, Komail Dharsee, and John Criswell. 2019. Secure Guest Virtual Machine Support in Apparition. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '19), April 14, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3313808.3313809

VEE '19, April 14, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6020-3/19/04…$15.00
https://doi.org/10.1145/3313808.3313809

1 Introduction

Modern operating systems such as Linux and FreeBSD are capable of running hardware-based virtual machines (Linux via KVM [22] and FreeBSD via bhyve [2]). The kernel code of these operating system (OS) kernels has been enhanced to use the virtualization features of modern x86 processors—Intel's *Virtual Machine Extensions* (VMX, also known as *VT-x*) [21] and AMD's *Secure Virtual Machine* (SVM, also known as *AMD-V*) [1]—to launch new instances of operating systems. This feature allows users to run applications designed for one operating system on top of another (e.g., a user can run a Windows virtual machine on a Linux laptop). It also allows users to run applications in an environment isolated from the host OS kernel, protecting the host kernel and applications running on it from programs running in the guest virtual machine (VM).

In parallel, recent research work has demonstrated how virtual instruction set computing (VISC) can be employed to enforce fine-grained security policies on application and OS kernel code. In VISC systems [4, 13, 14], software is compiled to a virtual instruction set; a trusted translator within a compiler-based virtual machine instruments the code with run-time checks to enforce a security policy before generating native code to execute on the processor. Secure Virtual Architecture (SVA) [13] is a VISC infrastructure that has been used to enforce memory safety [12, 13] and control-flow integrity (CFI) [10]. SVA has also been used to build systems such as Virtual Ghost [11] and Apparition [19] which protect applications from compromised OS kernels; current defenses mitigate both direct attacks and last-level-cache [20] and page-fault [39] side-channel attacks.

The SVA virtual instruction set currently lacks support for VMX hardware extensions which allow OS kernels to run whole-system virtual machines efficiently. This work extends the SVA virtual instruction set with new VMX-based instructions that allow an OS kernel running on an SVA-based system to execute virtual machines. Furthermore, we enhance the implementation of these instructions with run-time checks to ensure that the host OS kernel cannot use them to access the private memory of host applications. Specifically, we enhance Apparition [19] so that the host OS kernel cannot use hardware-based virtual machines to bypass Apparition's memory protections, side-channel protections, and control-flow protections. This new system, dubbed

Shade, protects host applications both from the host OS kernel and from guest software running atop the host OS.

The challenge in providing secure access to VMX hardware is that a compromised OS kernel could use it to bypass Apparition's [19] run-time checks. More specifically, Shade must solve the following challenges:

- Shade must enforce control flow integrity across VM entry and exit.
- Shade must prevent a compromised hypervisor from providing a guest VM with memory that the hypervisor may not read or write. Otherwise, a hypervisor could simply load its code into a guest VM to bypass Apparition's run-time memory protection checks.
- Shade must prevent a compromised hypervisor from giving a guest VM access to hardware features (e.g. model-specific registers) that could allow the guest to deactivate Apparition's security enforcement.
- Shade must ensure that Apparition's side-channel attack mitigations (e.g. cache partitioning) are applied to guest VMs.

To summarize, our contributions are as follows:

- We describe the design of new virtual instructions that allow OS kernels ported to SVA to use processor virtualization features.
- We describe the design and prototype implementation of *Shade*, a system that implements the new virtual instructions in a way that protects host applications from both the host OS and guest VM software.
- We evaluate the performance overhead of Shade's run-time checks.

The rest of the paper is organized as follows. Section 2 describes Apparition and the security guarantees that it provides, and Section 3 provides background on the Intel VMX processor features. Section 4 presents our threat model. Section 5 describes the design of our virtual instruction set extensions and how it protects host applications from compromised hypervisors and virtual machines under their control. Section 6 describes our prototype implementation, and Section 7 presents our experimental results. Section 8 presents related work, Section 9 describes directions for future work, and Section 10 presents concluding remarks.

2 Apparition

Apparition [19], shown in Figure 1, is a compiler-based virtual machine, based on SVA [13], that is placed between privileged software and hardware. Apparition protects application data from theft and corruption by a compromised OS kernel. Apparition divides the virtual address space of each process into four regions as Figure 2 shows:

- **User Memory**: User memory is the region of the virtual address space shared by an application and the OS kernel. Both the application and the OS kernel can read and write user memory. Standard operating systems such as Linux [6] and FreeBSD [26] provide user memory.
- **Kernel Memory**: Kernel memory is the region of the virtual address space belonging to the OS kernel; only the OS kernel and the Apparition VM are allowed to read and write kernel memory.
- **Ghost Memory**: Ghost memory is memory that application code is allowed to read and write but the OS kernel is prevented from reading and writing. Applications can store data in ghost memory to protect its confidentiality and integrity.
- **SVA VM Memory**: SVA VM memory is memory that only the Apparition VM can read and write. This memory is used to store Apparition's internal data structures.

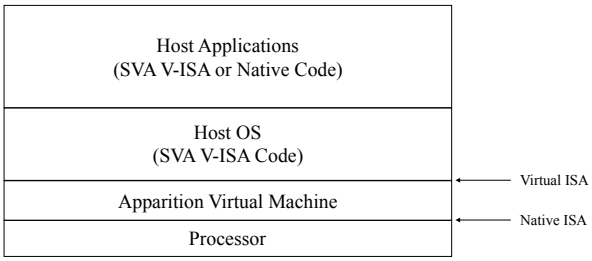

Figure 1. Apparition Architecture

Apparition [19] exposes a virtual instruction set based on LLVM IR [23] to which OS kernel software (and optionally, application software) is compiled; the processor implements a native instruction set. The Apparition VM translates virtual instruction set code to native code for execution, adding instructions to enforce Software Fault Isolation (SFI) [36] before every load and store in the OS kernel to ensure that the OS kernel cannot read or write ghost memory or Apparition VM memory. The Apparition VM also inserts instructions enforcing Control Flow Integrity (CFI) [3] into code during translation to prevent memory safety errors from bypassing the SFI mechanisms. Finally, the implementation of the SVA-OS instructions (a set of virtual instructions for operating systems to perform state manipulation and privileged hardware configuration) adds additional run-time checks to ensure that operations such as context switching, thread creation, and MMU configuration do not expose the contents

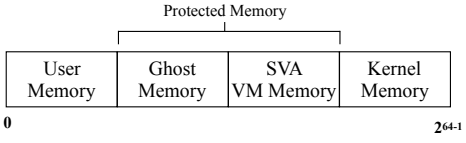

Figure 2. Virtual Address Space Layout

of ghost memory or SVA VM memory to theft or corruption [11].

As part of its confidentiality protections, Apparition [19] prevents a compromised OS kernel from using page-fault side-channel attacks [39] and last-level-cache (LLC) side-channel attacks [20] to steal application data residing in ghost memory. The SVA-OS MMU instructions in Apparition ensure that the OS kernel cannot read or write page table entries that map ghost memory. The Apparition VM also leverages Intel's Cache Allocation Technology (CAT) [21] to partition the last-level cache to prevent accesses to ghost memory from leaking information to the OS kernel via the last-level cache.

3 Intel Virtualization Extensions

To achieve optimal performance, a virtual machine should execute code directly on the system's real hardware as much as possible. However, because full-system virtual machines need to run privileged software (e.g., OS kernels) and yet need to be restrained from violating the confidentiality, integrity, and availability of the host system and other VMs, this is not always feasible. Virtualization systems must ensure that VMs trap to privileged host software, known as a *hypervisor* or *virtual machine monitor* (VMM), to safely emulate privileged operations without directly exposing them to potentially untrusted guest software.

To improve the performance of full-system virtual machine monitors and simplify their development, Intel added *Virtual Machine Extensions* (VMX) to their x86 processors [21]. VMX provides hardware support for implementing "trap-and-emulate" hypervisors sans binary recompilation or paravirtualization. AMD implements structurally similar, but incompatible extensions in their x86 processors known as *Secure Virtual Machine* (SVM) or *AMD-V* [1]. For simplicity, we focus on Intel hardware in this work; our approach should apply equally well to AMD systems. Privileged host software can instruct the CPU to enter a "guest mode" (known as *VMX non-root operation* in Intel's parlance), in which code continues to execute directly on the hardware, but the processor will stop and return to hypervisor code in "host mode" (*VMX root operation*) if selected privileged instructions are issued. Entry to guest mode is called *VM entry* and traps to the host *VM exits*. The hypervisor can flexibly configure which conditions trigger a VM exit to suit its needs.

Because VM exits are expensive, subsequent improvements to VMX have focused on providing mechanisms to safely virtualize an increasingly large subset of x86 functionality without trapping to the hypervisor, thus increasing performance. These improvements include *extended page tables* (EPT), the addition of an IOMMU to support efficient pass-through of physical devices to guests (*Intel VT-d*), and interrupt controller virtualization [21].

3.1 Basic VMX Operation

The VMX software interface is centered around the *Virtual Machine Control Structure* (VMCS), a roughly page-sized data structure which is managed by the hardware and accessible by software through special instructions [21]. Each VM that is active on the hardware has its own VMCS. The VMCS contains nearly all of the data relevant to running a VM on the physical hardware, including controls that define when VM exits occur, the current state of guest control registers, and fields that reflect which interrupts the guest is blocking, the power-management state of its virtual CPU, etc. The VMCS also stores key aspects of the suspended host state while a CPU is executing in guest mode, such as its instruction pointer and stack pointer. In general, the VMCS stores any guest and host state that must be saved/loaded atomically on VM entry and exit to ensure defined system operation. State that does not need to be saved/loaded atomically, such as the general-purpose registers, is not stored within the VMCS but left to the hypervisor to save/restore as it desires.

VMX requires that a VMCS be *loaded* onto a logical processor before it can be used [21]. This allows the hardware to optimize its use of the VMCS by caching some of its fields in internal, non-architecturally-visible registers across successive runs of the same VM. When the hypervisor wants to run a different VM, it must unload the active VM and load the new VM. Because portions of a loaded VMCS may be cached within the processor, Intel mandates the use of the special instructions `VMREAD` and `VMWRITE` to access VMCS fields instead of directly reading and writing them in memory. These instructions will only operate on a currently loaded VM, of which there can only be one at a time on a given logical processor. The actual in-memory format of the VMCS is implementation-dependent, and Intel warns that directly accessing it may cause undefined behavior.

VM entry is triggered by one of two privileged instructions, `VMLAUNCH` and `VMRESUME` (which are identical except that the optimized `VMRESUME` is used when the active VM has remained loaded on the processor since a previous run; the instructions are mutually exclusive) [21]. These instructions perform a bevy of complex functions in a single atomic step to transition the processor into guest mode before resuming execution from the guest's program counter. Prior to VM entry, the hypervisor writes appropriate values to VMCS fields that will be loaded into control registers, segment registers, model-specific registers (MSRs), and other processor state on VM entry, thus defining the operating environment for the guest system. It likewise writes to a corresponding set of fields in the same VMCS that will be loaded atomically on VM exit, thus defining the state to which the host will return when guest execution is suspended due to attempting to perform some privileged operation. When `VMLAUNCH`/`VMRESUME` is executed, the processor checks the guest and host state fields, as well as the control fields defining when VM exits

should occur, for consistency. If and only if all of these checks succeed, the processor enters guest mode and resumes execution from the guest's program counter. Otherwise, control is returned to the hypervisor with an error.

A VM exit occurs when guest code attempts to perform an operation that is defined as privileged by the VMCS controls (as configured by the hypervisor) [21]. The reverse of VM entry is performed, again as an atomic operation from a software perspective: guest state is saved to the VMCS, and host state is loaded. The hypervisor then resumes execution in the state it specified prior to VM entry. It can then read various VMCS fields that report the reason for the exit and any information that may be needed to handle it appropriately. How a VM exit is handled is up to the hypervisor; a common reason for an exit is an attempt by the guest to perform I/O, which is traditionally emulated by the hypervisor through virtual devices (disks, network interfaces, graphics cards, etc.) managed in software [30]. Once the hypervisor has modified the guest's state to reflect any emulated privileged operations, it may perform VM entry again to resume the guest.

3.2 Extended Paging

Extended paging [21] (called *Extended Page Tables* (EPT) by Intel) provides additional support beyond the basic VMX feature set for efficient virtualization of physical memory for VMs. Although guest memory can be virtualized without extended paging, strategies for doing so are cumbersome, may be inefficient, and can require extensive duplication of guest paging hierarchies [21]. EPT provides a straightforward mechanism for virtualizing what the guest sees as "physical memory" by mirroring the design of standard x86-64 paging.

A 4-level hierarchy of extended page tables, nearly identical in layout to the regular page tables used to virtualize the memory spaces of user processes, maps *guest-physical* addresses to *host-physical* addresses [21]. When a guest attempts to access what it sees as a physical address, the MMU translates it to a real physical address on the host and accesses the appropriate address. As long as the hypervisor has mapped a real physical page into the respective entry in the extended page table hierarchy, the processor will directly perform the memory access on behalf of the guest without leaving guest mode or otherwise involving the hypervisor. If the requested guest-physical address is unmapped in the VM's EPT hierarchy, a VM exit will occur, called an *EPT fault*, trapping to the hypervisor, which can map in a physical page and re-enter the VM so the guest can re-try the access. Alternatively, the hypervisor might manipulate the guest's execution state to emulate the attempted memory access without mapping in a real memory page, e.g. to provide the guest with memory-mapped I/O to virtual devices. In general, the mechanism of EPT faults is exactly parallel to the way operating systems handle page faults by user applications. The pointer to the root of the extended paging hierarchy, comparable to `CR3` in normal paging, is stored in the VMCS and set by the hypervisor [21].

Note that the VM guest, as a fully virtualized x86 system, has its *own* regular page tables which operate "on top of" extended paging [21]. Extended paging is invisible to the guest, which only sees the results of ordinary memory accesses within what it perceives to be physical memory, just as a user application only sees virtual memory in conventional OS paging. The guest OS may manage its memory however it chooses, just as if it were running on real hardware. It may elect to load `CR3` and set up its own page tables to virtualize its guest-physical address space to its own user applications, in which case the processor translates guest memory accesses through as many as *eight* levels of page tables in total (four EPT levels + four conventional page table levels for a 64-bit guest). Alternatively, the guest may access guest-physical memory directly by running in an unpaged mode, in which case its memory accesses are only translated through the four-level EPT hierarchy.

4 Threat Model

Our threat model, based on the threat model of Apparition [19], assumes that privileged system software (such as the OS kernel and hypervisor) is vulnerable and can be controlled by an attacker. (We assume that the hypervisor is integrated into the host OS kernel in this work, though our solutions can also be applied to standalone hypervisors that operate in lieu of a host OS.) This attacker seeks to read or write memory belonging to user-space applications whose data is protected by the *trusted execution environment* (TEE) provided by the Apparition compiler-based virtual machine [19]. Confidentiality attacks may attempt to directly read memory or may opt to use page-fault side channel attacks [39] or last-level cache side-channel attacks [20]; other side channels not mitigated by Apparition are outside the scope of our threat model. The attacker may also seek to steal or corrupt data indirectly by attempting to disclose or corrupt data within the Apparition virtual machine, which enforces the TEE protections by mediating the system software's interactions with privileged hardware state.

We assume that the Apparition VM, and Shade's extensions to it, are implemented correctly. The Apparition VM has a Trusted Computing Base (TCB) of 6,841 source lines of code (SLOC) [19]; Shade's TCB (a superset of Apparition's) is 10,111 SLOC (see Table 3 in Section 6). With these sizes, we believe the Apparition/Shade TCB is amenable to strong testing and verification efforts. We assume also that protected user-space applications running within the Apparition TEE, and libraries they use, are part of the TCB for that application's security policy (though not for other independently protected applications). Attacks leveraging vulnerabilities within applications and their libraries are out of scope for our

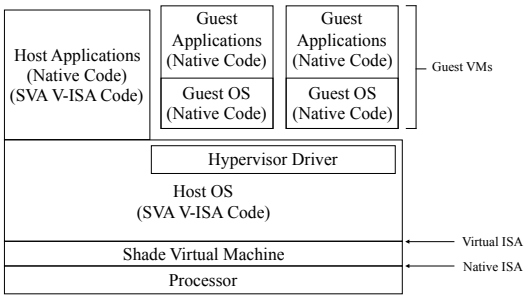

Figure 3. Shade Architecture

work; applications can defend themselves using hardening tools [17, 28, 29] or type-safe languages. Since the application is designed to take advantage of a TEE, we assume that it and its libraries distrust return values from the OS to protect against Iago attacks [7].

We explicitly *exclude* the OS kernel and hypervisor from the TCB. Apparition [19] provides applications the ability to maintain a secure operating environment notwithstanding kernel compromise.

Guest virtual machines are considered untrusted. Containment of vulnerable software is a primary use case for virtualization systems. Furthermore, because guest VMs are under control of the hypervisor (which is untrusted), even a benign VM could be co-opted by a compromised hypervisor to launch attacks against ghost memory and the Shade VM.

Attacks involving physical access to the machine are out of scope for our work (although SVA does provide protection against DMA attacks by requiring the use of an IOMMU [12]).

5 Design

Figure 3 shows Shade's architecture: it is a compiler-based virtual machine based on SVA that protects host applications from the host OS kernel. However, as Figure 3 depicts, Shade also permits the host OS kernel to create and execute hardware-based virtual machines via extensions to the virtual instruction set that expose the Intel VMX [21] processor features. These hardware-based virtual machines can execute native-code OS kernels and applications. Shade protects host applications from these hardware-based VMs and the hypervisor driver managing them just as Apparition [19] protects host applications from the OS kernel. Shade protects the confidentiality and integrity of data in ghost memory and of interrupted application state saved during interrupts, traps, system calls, and context switches. Shade prevents the OS kernel and guest VMs from using direct memory read attacks, page-fault side-channel attacks [39], and LLC side-channel attacks [20] to steal data stored in ghost memory. In this way, Shade prevents the host OS kernel from using the new virtual instructions supporting hardware VMs to bypass the security guarantees originally provided by Apparition.

In this section, we describe in detail the threats to Apparition's security policies posed by VMX features and the countermeasures Shade employs to mitigate them.

5.1 Control Flow Integrity

Like Apparition [19], Shade uses SFI to prevent the OS kernel from reading and writing ghost memory and SVA VM memory. Shade, like Apparition, also uses CFI enforcement to ensure that compromised kernel-mode software cannot bypass the SFI instrumentation. Thus, Shade must handle VM entry and exit in a way that enforces CFI.

As Section 3 describes, the hypervisor (in our case, the hypervisor driver within the OS kernel) configures within the VMCS the code address to which the processor should transfer control upon a VM exit. A compromised host could configure this code pointer within the VMCS to bypass CFI by setting this VMCS field to an arbitrary value. In addition to this code pointer, the VMCS contains various other fields storing sensitive saved host state fields—such as its stack pointer and segment registers—which could similarly be exploited to corrupt control flow integrity.

We address this threat by storing the VMCS within the SVA VM's memory and by controlling access to a guest VM via the interface of our new SVA-OS instructions. Table 1 summarizes the virtual instructions (intrinsics) that encapsulate the management of guest VMs. The `sva.allocvm()` intrinsic creates a new guest VM by allocating a VMCS within the SVA VM memory and initializing it with the values pointed to by the `initial_ctrls` and `initial_state` structures, which are vetted to ensure that they do not create a guest VM with permissions that could bypass Shade's security enforcement. The intrinsic also initializes the pointer to the guest VM's extended page tables in the VMCS via the `initial_eptable` argument; this pointer must point to a frame that has already been declared as a top-level EPT page. Since the VMCS is allocated within the SVA VM's memory, the OS kernel cannot read or write it due to Shade's SFI instrumentation. It therefore cannot modify the aforementioned sensitive host state fields within the VMCS during or after creation of the guest VM. The `sva.allocvm()` intrinsic returns an identifier denoting the new guest VM upon success.

With this design, Shade can prevent violations of control flow integrity. Because privileged system software is forced to use SVA-OS virtual instructions in lieu of assembly code, the VM entry/exit process is under control of the Shade VM. Shade disallows the system software from writing to the VMCS fields controlling host state on VM exit, and instead writes its own safe values there to ensure that VM exits transfer control to the Shade VM.

Before a guest VM can start execution, it must be loaded onto a processor. The `sva.loadvm()` and `sva.unloadvm()` intrinsics load and unload the specified guest VM onto and

Table 1. Shade SVA-OS VM Management Intrinsics

Name	Description
int sva.allocvm (sva_vmx_vm_ctrls *initial_ctrls, sva_vmx_guest_state *initial_state, pml4e_t *initial_eptable)	Allocate a descriptor and supporting data structures for a virtual machine and initialize them with the provided execution controls and guest state. Returns an integer ID that identifies the VM.
void sva.freevm(int vmid)	Deallocate a virtual machine and associated structures.
int sva.loadvm(int vmid)	Make a virtual machine active on the processor. Returns an error code.
int sva.unloadvm(void)	Unload the current virtual machine from the processor. Returns an error code.
int sva.runvm(void)	Enter guest-mode execution of the currently loaded VM, which suspends host operation until a VM exit occurs. Returns an error code.
uint64_t sva.getvmreg (size_t vmid, enum sva_vm_reg reg)	Gets the current value of a guest register whose state is saved and loaded by Shade on VM entry/exit (i.e., not managed by the processor in the VMCS).
void sva.setvmreg(size_t vmid, enum sva_vm_reg reg, uint64_t data)	Sets the value of a guest register which is saved/restored by Shade.
int sva.readvmcs (enum sva_vmcs_field field, uint64_t *data)	Reads a field in the currently loaded VM's Virtual Machine Control Structure, storing the value in `*data` and returning an error code.
int sva.writevmcs (enum sva_vmcs_field field, uint64_t data)	Writes the contents of `data` to a VMCS field in the currently loaded VM's VMCS. Returns an error code.

off of the currently executing processor, respectively. The `sva.freevm()` intrinsic deallocates a guest VM so long as it has been unloaded from the processor.

Guest registers that are not managed by the processor in the VMCS as part of the atomic save/restore operations performed during VM entry and exit, such as the general-purpose and floating-point registers, must be loaded immediately before VM entry and saved immediately after VM exit by the host. Since VM entry and exit take place within Shade VM code, Shade initializes a guest's registers with the values provided to `sva.allocvm()`, and stores their values in SVA VM memory while the guest is not running. The `sva.getvmreg()` and `sva.setvmreg()` intrinsics allow the hypervisor driver to observe and change these values while the guest is not running.

From the hypervisor driver's perspective, the virtual instruction for VM entry, `sva.runvm()` (corresponding to the native instructions `VMLAUNCH` and `VMRESUME`, whichever is situationally appropriate) has the control-flow semantics of a normal function call. When the hypervisor calls this virtual instruction, it relinquishes control to the Shade VM, which runs the active virtual machine until a VM exit occurs, then returns to the calling code in the hypervisor. CFI is therefore maintained in the untrusted hypervisor.

5.2 Protecting Ghost Memory and System Integrity

As Section 2 states, Apparition [19] utilizes run-time checks in its implementation of the SVA-OS MMU instructions to prevent a compromised OS kernel from reading and writing ghost memory and SVA VM memory. Specifically, it prevents the kernel from changing page table mappings for virtual addresses within the protected memory region. This prevents the OS kernel from making SVA VM memory accessible to user-space code, or mapping the physical frames corresponding to protected memory into virtual addresses to which the OS kernel can read and write (namely, the user and kernel memory regions in Figure 2). As Apparition stores all native code translations within the SVA VM memory, the OS kernel cannot modify its code pages to be writable and cannot map new memory frames with arbitrary contents into its code segment. This prevents the OS kernel from loading new native code or writing NOP instructions over the SFI and CFI instructions added by the Apparition VM when translating virtual instruction set code to native code. Page-table pages (PTPs) are also located within SVA VM memory to force the OS to use the SVA-OS MMU instructions to perform page-table updates. The OS must declare PTPs to Apparition before use so that Apparition can protect them; the SVA-OS MMU instructions will refuse to add a page to a page table if it has not previously been declared as a PTP.

Guest virtual machines running under VMX pose similar threats to protected memory. A compromised hypervisor could use extended paging (EPT, described in Section 3) to map protected memory frames into a guest VM's physical address space. The hypervisor could then load native code into the guest VM which accesses the protected memory, thereby circumventing Apparition's protections.

Shade provides SVA-OS virtual instructions (summarized in Table 2) for configuring extended paging, similar to the existing SVA-OS instructions for configuring regular page

Table 2. Shade SVA-OS Extended Paging Intrinsics

Name	Description
void sva.mm.declare.l<1-4>.eptpage (uintptr_t frameAddr)	Declare a frame to Shade for use as a page-table page in extended paging (EPT).
void sva.mm.update.ept.mapping (page_entry_t *eptePtr, page_entry_t val)	Update an entry (add, remove, or update a mapping) in an extended page table.
void sva.mm.load.eptable (int vmid, pml4e_t *empl4t)	Load the root extended-page-table pointer for a VM.
uintptr_t sva.mm.save.eptable (int vmid)	Save the current value of the extended-page-table pointer for a VM. Returns the current contents of the EPT pointer field in the specified VM's VMCS.
void sva.mm.remove.page (uintptr_t paddr)	Informs Shade that the system software will no longer be using a page as an extended page-table page. (This intrinsic is also used for non-extended paging.)

tables [9, 11, 12]. As with regular page tables, the Shade VM prevents the kernel/hypervisor from writing directly to extended page table pages, and it prevents guest VMs from mapping host PTPs (regular and extended) via EPT. This ensures that the OS kernel must use the SVA-OS instructions to configure the extended page tables. Shade's implementation of the SVA-OS virtual instructions performs run-time checks that ensure EPT mappings are not created that would violate Shade's security policies. Specifically:

1. Guests may not have EPT mappings to ghost memory. This prevents theft and corruption of sensitive data within host applications.
2. Guests may not have EPT mappings to SVA VM memory. This prevents theft or corruption of data held in Shade's data structures. It also prevents guests from reading or writing the native code of the kernel/hypervisor, host applications, and the Shade VM.
3. Guests may not have mappings to host page-table pages, whether regular or extended.

5.3 Controlling Privileged Hardware Configuration

Apparition [19] prevents a compromised OS kernel from reconfiguring privileged hardware in a way that would circumvent its security enforcement. As it forces the OS kernel to use SVA-OS virtual instructions to configure privileged hardware state, either run-time checks in the SVA-OS instructions prevent insecure configuration of privileged hardware state, or the SVA-OS instructions themselves provide no way of expressing an insecure configuration of privileged hardware state. For example, if the OS kernel could clear flags in the `CR0` control register, it could disable hardware memory protection [21] upon which the Apparition VM relies. However, as the SVA-OS interface provides no instruction for the kernel to set or clear flags in `CR0`, the OS kernel has no way of thus configuring the processor unsafely.

Because VMX allows guest virtual machines to run native code directly on the hardware in privileged modes, a compromised kernel could instruct its hypervisor driver to create a guest that is permitted to change privileged system state in ways that impact Shade's security enforcement. VMX allows hypervisors to configure the conditions that will trigger a VM exit through controls in VMCS fields (see Section 3) [21]. If, for instance, a VMCS is configured so that a guest VM does not perform a VM exit when it writes to a privileged model-specific register (MSR), the value written to that MSR will persist after VM exit. The Shade VM must ensure that all privileged state components which can be changed by a guest are restored to proper host values on VM exit.

VMX provides features which allow safe virtualization of some privileged state without affecting the corresponding host state. For example, the privileged `CR3` register is virtualized by extended paging and automatically switched between host and guest values configured in the VMCS on VM entry and exit. Other privileged state, such as kernel-mode MPX state used by Apparition to accelerate SFI checks [19], is not managed by VMX; Shade must save/load such state, as it does general purpose registers, on VM entry and VM exit.

For any remaining state that is managed neither by VMX nor Shade (e.g., because it is not incorporated into the design of a particular implementation of Shade), Shade configures the VMCS to force a VM exit when a guest VM attempts to change that state. This is achieved through run-time checks performed by the `sva.writevmcs()` intrinsic, which prevent control bits from being set unsafely in fields that determine whether the use of particular processor features will cause VM exits. Similar run-time checks are performed by `sva.readvmcs()` to ensure that the untrusted hypervisor may not read sensitive privileged state stored in the VMCS, such as saved host state fields (see Section 5.1).

5.4 Side-Channel Attacks

Apparition thwarts last-level-cache (LLC) and page-fault side channel attacks against ghost and SVA VM memory [19]. Shade maintains these protections in the presence of untrusted guest VMs.

5.4.1 Last-Level-Cache Side Channels

Apparition [19] uses Intel's *Cache Allocation Technology* (CAT) feature [21] to thwart last-level-cache (LLC) side-channel attacks. Using CAT, Apparition statically partitions the LLC so that the OS kernel and the Apparition VM have their own cache partitions. Apparition also gives each application using ghost memory (called a *ghosting application*) its own cache partition. Applications not using ghost memory use the OS kernel's partition. These protections prevent a compromised OS kernel from launching LLC side-channel attacks against ghosting applications and the Apparition VM. It also prevents both regular and ghosting applications from attacking other ghosting applications on the system.

Shade must ensure that guest VMs cannot use LLC side-channel attacks against host ghosting applications and the Shade VM. In Shade, the hypervisor driver, being part of the OS kernel, uses the OS kernel's cache partition. Shade also places guest VMs within the OS kernel's cache partition. This isolates guest VMs' cache lines from cache lines used by host ghosting applications and the Shade VM. Shade's `sva.runvm()` intrinsic switches between Shade's and the OS's cache partitions on VM entry and exit.

The Intel VMX extensions [21] permit a hypervisor to configure the VMCS so that code running in the guest OS can reconfigure the Intel CAT partitions. If the guest VM can reconfigure the cache partitions, it could reconfigure the hardware so that it and ghosting applications share the same cache lines, allowing cache side-channel attacks. To mitigate this, Shade initializes all new VMCS structures that it allocates so that writing to any of the MSRs that configure Intel CAT will cause a VM exit. Furthermore, the `sva.writevmcs()` intrinsic prevents the hypervisor driver from subsequently reconfiguring the VMCS to allow guests to write to those MSRs. With these two protections, guest VMs cannot give themselves access to the cache lines belonging to ghosting applications.

5.4.2 Page-Fault Side Channels

Apparition [19] prevents the OS kernel from launching page-fault side-channel attacks [35, 39] against ghost memory and SVA VM memory by moving page table pages into the SVA VM memory. This allows Apparition to prevent reads and writes to page table pages using its existing SFI instrumentation. Additionally, the SVA-OS MMU instructions do not allow the OS kernel to read or modify page table entries that map memory frames into either ghost memory or SVA VM memory. This prevents the OS kernel from forcing an application to page fault when accessing its ghost memory [39] or from inferring ghost memory page accesses by reading the accessed and dirty bits in page table entries [35].

Shade likewise prevents code within guest VMs from launching page-fault side-channel attacks against applications using ghost memory. Since Shade prevents the OS kernel from mapping host page tables into the guest VM's address space, the guest VM cannot read or write host page table entries and therefore cannot launch page-fault side-channel attacks [35, 39] against host applications.

As an additional line of defense against page-fault side-channel attacks, Apparition [19] disables on-demand allocation of physical memory for ghost memory to prevent the OS from inferring a ghosting application's memory access patterns by observing when the Apparition VM requests physical memory from the OS. Instead, physical memory is mapped when an application requests that virtual address space within ghost memory be allocated. Shade does not affect this enhancement.

Table 3. Shade TCB Size

Component	Apparition SLOC [19]	Shade SLOC
Compiler Passes	1,018	1,018
SVA-OS Instructions	5,823	9,093
Total	**6,841**	**10,111**

6 Implementation

We implemented Shade by extending the existing source code for Apparition [19] which is built for 64-bit x86 systems. Shade uses LLVM 3.1. We used the FreeBSD 9.0 kernel ported to the SVA virtual instruction set [10, 11, 19]. The Shade SVA-OS instructions are written in C and assembly code while the LLVM compiler passes are written in C++.

Table 3 shows the TCB size of Apparition and Shade; the Apparition sizes were reported by Dong *et al.* [19] while we measured the TCB size of Shade using sloccount [38]. The Apparition source code contains the combined source code of KCoFI [10], Virtual Ghost [11], Apparition [19], and the experimental Apparition system used to evaluate Spectre-resistant SFI [18]. Shade contains all of the code for these systems plus the additional code for VMX support.

Our own measurements confirm that the compiler passes for Apparition and Shade have the same size; this is expected as Shade does not modify the LLVM compiler passes used to implement SFI and CFI for Apparition. However, Shade increases the size of the SVA-OS instructions by 56%, increasing the overall TCB size by 48%. There are three reasons for the increase of TCB size despite the simplicity of Shade's code. First, Apparition's TCB was quite small in absolute terms. Second, there is a significant amount of state to save/restore on VM entry/exit; there are roughly 2-4 lines of code for each piece of CPU state. Third, the VMX extensions have numerous configurable options. This leads to large switch statements in Shade's code to implement the run-time checks and structure definitions with many field members.

To evaluate and guide the design of Shade's virtual instructions supporting VMX use by hypervisors, we developed a

minimalist "toy" hypervisor using the Shade API which allowed us to directly test basic VMX operations without the complexity of a full hypervisor. This hypervisor is capable of constructing simple guest VMs, allocating memory to them using EPT, and loading program code into them to be run in guest mode. It is able to run VMs and automatically handle a selection of VM exits, such as directing external interrupts to the host OS as appropriate and manipulating a guest's state to emulate privileged instructions. We employed this toy hypervisor to support the hypervisor benchmarks we performed in our evaluation (Section 7.2).

In addition to the state saved/restored by the processor on VM entry/exit (Section 3.1), some fields must be saved/restored by the hypervisor to support their use in guests. We support the subset of these fields necessary to support our benchmarks and some future work (Section 9). These include the general purpose registers, `FXSAVE`-managed FPU state, `CR2`, `XCR0`, MPX bounds registers, and `IA32_BNDCFGS`.

7 Evaluation

For our experiments, we measured the performance overhead that Shade's run-time checks induce. We specifically sought to quantify two potential areas of impact:

1. **Impact on host performance.** Shade's addition of VMX support left most of Apparition's SVA-OS instructions unchanged, but some enhancements were made to the existing MMU instructions to facilitate EPT support.
2. **Impact on hypervisor performance.** Ideally, the run-time checks performed by Shade's implementation of the new virtual instructions to support VMX (see Section 5) should not impose significant overhead on the basic operations performed by a hypervisor.

We used the LMBench [27] microbenchmarks to measure Shade's impact on host kernel performance (Section 7.1) and developed microbenchmarks for key VMX operations to measure the impact on hypervisor performance (Section 7.2).

For all of our experiments, we used a Dell Precision T3620 workstation. This machine has a sixth generation Intel® Core™ i7-6700 processor (four cores at 3.4 GHz with 8 MB of cache), 16 GB 2133 MHz DDR4 memory, a 256 GB SATA SSD, and a 500 GB SATA 7,200 RPM hard drive.

7.1 Host Kernel Microbenchmarks

To evaluate the impact of Shade's run-time checks on host kernel performance, we ran LMBench under five different configurations:

- Baseline FreeBSD 9.0 kernel
- Shade, no VMX checks, non-ghosting (**NC-NG**)
- Shade, no VMX checks, ghosting (**NC-G**)
- Shade, with VMX checks, non-ghosting (**C-NG**)
- Shade, with VMX checks, ghosting (**C-G**)

The baseline native x86-64 FreeBSD 9.0 kernel was compiled with the same Clang compiler and compiler optimizations used when compiling the Shade FreeBSD 9.0 kernel; the only difference in compiler settings is that Shade's SFI and CFI instrumentation are not applied to the native kernel.

The configurations labeled "*ghosting*" run LMBench under a modified `libc` that allocates heap objects in secure ghost memory instead of traditional user-space memory. This allows us to test the impact of security checks and changes to kernel behavior that only affect applications that utilize ghost memory. Notably, although the original Apparition paper [19] ran a similar suite of LMBench benchmarks, it did not run them in ghosting mode. Thus, our evaluation provides additional insight into Apparition's performance in areas unmodified by Shade.

We measured kernel performance both with all of Shade's security checks intact, and with checks specifically related to VMX disabled. From the perspective of applications like LMBench which do not utilize VMX, Shade sans VMX-specific checks is essentially just Apparition; this allows us to observe any performance impact Shade adds beyond Apparition, in addition to comparing to the native baseline FreeBSD kernel.

We used the version of LMBench available in the FreeBSD 9.0 ports tree. We selected the benchmarks that measure the latency of OS kernel features. We configured the benchmarks to run for 1,000 repetitions under their default configurations except as follows:

- We configured the `lat_mmap` benchmark to map the first 1 MB region of a randomly generated file located on the SSD; we collected data from the `/dev/urandom` device to create the file's contents.
- We configured the `lat_pagefault` benchmark to map a file of about 10 MB in size (located on the SSD) into memory and access it with a 256 KB stride.
- We ran the pipe benchmark, `lat_pipe`, for 10 repetitions. This benchmark takes longer to execute than most of the others; this change allows the benchmark to complete in a reasonable amount of time.
- We ran the `lat_pagefault` benchmark for 10,000 repetitions to reduce experimental error.
- We omitted the `lat_ctx` context switching benchmark because a bug in our Shade prototype sometimes causes the kernel to crash.

Table 4 shows the results of the LMBench benchmarks. We show the arithmetic mean of the execution time (in microseconds) of the baseline FreeBSD 9.0 kernel and the normalized runtimes for the aforementioned configurations of Shade. The standard deviation is expressed as a percentage of the arithmetic mean execution time.

Our results show that the VMX-specific run-time checks that Shade adds to Apparition have no significant impact on the kernel's performance for applications that do not utilize VMX (i.e., **Shade-NC-NG** is similar to **Shade-C-NG**,

Table 4. LMBench Performance Results

Test	FreeBSD 9.0		Shade-NC-NG		Shade-NC-G		Shade-C-NG		Shade-C-G	
	Mean (μs)	Std. Dev.	Normalized Runtime	Std. Dev.	Normalized Runtime	Std. Dev.	Normalized Runtime	Std. Dev.	Normalized Runtime	Std. Dev.
nullSyscall	0.1021	0.07%	6.69×	0.01%	6.69×	0.01%	6.69×	0.01%	6.69×	0.01%
write	0.1311	0.79%	5.63×	0.01%	6.06×	0.02%	5.63×	0.02%	6.06×	0.02%
read	0.1679	0.51%	6.03×	0.01%	6.36×	0.02%	6.03×	0.01%	6.36×	0.03%
sigInstall	0.1864	0.11%	6.63×	0.01%	6.63×	0.01%	6.63×	0.01%	6.63×	0.01%
pgFault	0.8778	10.28%	1.38×	4.20%	1.41×	7.02%	1.44×	3.98%	1.40×	5.29%
sigDeliver	1.2779	0.04%	1.80×	0.07%	1.80×	0.08%	1.81×	3.35%	1.79×	0.03%
stat	1.5153	0.10%	1.98×	0.05%	2.02×	0.08%	1.98×	0.06%	2.02×	0.04%
openClose	1.8528	0.23%	2.05×	0.33%	2.14×	5.17%	2.05×	0.19%	2.09×	0.14%
pipe	4.0589	0.93%	1.41×	0.19%	137.97×	0.10%	1.41×	0.26%	136.93×	0.10%
fcntl	4.9550	1.08%	1.61×	1.17%	111.93×	0.04%	1.61×	0.33%	112.04×	0.05%
mmap	8.8076	0.26%	2.61×	0.00%	2.61×	0.00%	2.66×	2.21%	2.63×	1.82%
select	14.8543	0.53%	1.37×	0.02%	1.37×	0.02%	1.37×	0.10%	1.37×	0.01%
forkExit	71.0459	0.17%	3.14×	1.11%	58.17×	0.03%	3.13×	0.66%	58.55×	0.06%
forkExec	79.3289	0.25%	3.04×	0.67%	51.83×	0.05%	3.03×	0.67%	52.19×	0.08%
forkShell	663.5778	0.12%	2.27×	0.33%	20.23×	0.16%	2.29×	0.31%	20.31×	0.12%

and **Shade-NC-G** is similar to **Shade-C-G**). This was in line with our expectations since the VMX-related code added to existing instructions is small and computationally simple. The majority of the code added by Shade supports virtual instructions specifically geared to hypervisors.

In general, our results show that system calls with small execution times (such as the null system call test and read/write) suffer a larger relative performance overhead compared to longer-running operations. However, the most notable finding is that tests that incorporate the `fork()` system call (*forkExit*, *forkExec*, *forkShell*, *pipe*, and *fcntl*) suffer dramatic overheads when they use ghost memory for heap allocations (**Shade-NC-G** and **Shade-C-G**).

Our investigation of this issue indicates that Apparition's side-channel defenses are responsible for these large overheads. When we disabled page-fault side-channel defenses (which maps physical frames to ghost memory virtual addresses at allocation time to prevent the OS from inferring application access patterns by observing page faults), the majority of the overhead compared to the non-ghosting configurations (**Shade-NC-NG** and **Shade-C-NG**) disappears. Furthermore, disabling Shade's last-level-cache side-channel defenses (i.e., cache partitioning using Intel CAT) eliminates the remainder of the discernible overhead, leaving performance comparable to the non-ghosting configurations.

We emphasize that these overheads are part of the original Apparition [19] system and not due to the enhancements made by Shade; this is clear from the fact that the **C** configurations (with Shade's VMX-related checks) perform comparably to the **NC** configurations (representing the latest version of Apparition). The original paper on Apparition [19] did not evaluate LMBench [27] using ghost memory.

7.2 Hypervisor Microbenchmarks

To measure the impact of the security checks performed by the new SVA-OS instructions that we have added to create Shade, we created microbenchmarks for the basic VMX operations our instructions provide.

For our baseline, we recompiled the SVA-OS library that implements the SVA-OS instructions with all VMX-related security checks disabled at compile time. We examined the open-source code of three real-world hypervisors (FreeBSD's bhyve [2], Linux's KVM [22], and the FreeBSD edition of VirtualBox [31]) and observed that they all incorporate low-level abstraction layers that encapsulate the assembly code performing raw VMX operations to provide a more convenient interface to the rest of the hypervisor. We believe that Shade's SVA-OS runtime library compiled without VMX-related security checks provides an interface comparable in abstraction level and overhead to a typical non-Shade-based hypervisor's low-level hardware abstraction layer and therefore serves as a viable baseline for microbenchmarks. Because our baseline presents the same API as Shade's intrinsics, we can perform a head-to-head comparison between Shade and the baseline without capturing unrelated differences in performance due to abstraction layer design.

Our baseline thus reflects a hypothetical Apparition system that unsafely uses native VMX operations directly (circumventing SVA's security model), in the manner that a typical hypervisor would use them (via calls to assembly code in its low-level abstraction layer). This is equivalent

Table 5. Hypervisor Microbenchmark Results

Test	Baseline		Shade	
	Arithmetic Mean	Std. Dev.	Normalized Runtime	Std. Dev.
Create/destroy VM	0.50 μs	0.03%	1.00×	0.03%
VM entry/exit	6.5 μs	0.01%	1.01×	0.00%
VMCS read	0.12 μs	0.02%	1.02×	0.01%
VMCS write	0.12 μs	0.01%	1.03×	0.01%

to the **Shade-NC-*** configurations evaluated in Section 7.1. We compare this against the overheads exhibited by Shade, wherein our safe (checked) virtual instructions are used in place of those unchecked low-level routines.

These microbenchmarks measure the overheads of individual operations provided by our VMX virtual instructions in isolation; the actual runtimes reported (in μs) should be interpreted with this in mind. They provide a useful measure of the impact of our checks, but may not be comparable to the runtimes of similar high-level operations in commodity hypervisors, which perform those operations in the context of larger software systems and may need to perform other tasks that would dominate their runtime (bookkeeping, configuring virtual devices, etc.).

7.2.1 Benchmark Design and Environment

We created four hypervisor microbenchmarks and ran them under both Shade and the aforedescribed baseline system. Table 5 summarizes the results. Each experiment was run 10 times for an experiment-specific number of iterations (specified below), and its runtime, in clock cycles, was measured using the CPU's timestamp counter (TSC). The Intel® Core™ i7-6700 processor provides a constant-rate TSC which measures real time (although we additionally disabled the CPU's clock frequency scaling in the BIOS to reduce experimental noise); we observed that our processor's TSC increments by approximately $3.4115 * 10^9$ ticks in one second, i.e., 3.4115 GHz, from which we can convert from TSC ticks to microseconds.

To further reduce experimental noise, we ran 10 untimed iterations of each benchmark loop prior to running the timed iterations to "warm up" any relevant CPU or OS caches that might otherwise yield nondeterministic performance. We ran each experiment immediately after booting the system, first running the full benchmark suite once as a warm-up and then a second time for the recorded experiments.

Create/destroy VM This benchmark tests the Shade intrinsics `sva.allocvm()` and `sva.freevm()`. Each round of the benchmark sets up a new VM and then immediately frees it. VM setup performed by `sva.allocvm()` includes allocating a physical page for the VM's VMCS, initializing it using the hardware instruction `VMCLEAR`, and setting up data structures in the Shade VM to track the guest system's saved registers between runs. (These same steps are performed, sans security checks, in the baseline version, as they correspond to steps a typical non-Shade-based hypervisor's low-level hardware abstraction layer would likewise perform to set up a new VM.) We performed 10,000,000 iterations of this test in each run of the experiment (which, as noted above, was performed 10 times overall in the full suite).

VM entry/exit VM entry and exit were tested using our toy hypervisor implementation described in Section 6. The timed portion of the benchmark measures entry/exit round-trip latency using `sva.runvm()` on an already-created and previously-launched VM (i.e., using the hardware instruction `VMRESUME`). The guest used in this scenario runs an infinite loop consisting of two instructions: `VMCALL`, which performs an intentional VM exit (hypercall), and an unconditional jump back to the `VMCALL` to implement the infinite loop. The toy hypervisor handles `VMCALL` exits by using `sva.readvmcs()` to determine the exit reason and read the guest's current instruction pointer, then `sva.writevmcs()` to increment the instruction pointer to skip over the `VMCALL`. Each run of this test performed 1,000,000 iterations. The toy hypervisor distinguishes between VM exits due to external interrupts (which we handled by allowing the host OS to service the interrupt, then immediately resuming the VM) and exits incident to guest events; we counted only guest-initiated exits (i.e., those due to `VMCALL`) in the number of test iterations performed.

VMCS read We tested the `sva.readvmcs()` intrinsic's performance by repeatedly reading from the *Guest RIP* VMCS field of a VM which had been previously created and loaded onto the processor. Each run performed 10,000,000 iterations.

VMCS write We tested the `sva.writevmcs()` intrinsic's performance by repeatedly writing to the *Guest RIP* VMCS field of a VM which had been previously created and loaded onto the processor. Each run performed 10,000,000 iterations.

7.2.2 Observations

As shown in Table 5, we did not observe significant overheads when checks were enabled on the VMX operations provided by Shade's intrinsics. Where we observed statistically significant overheads, they were within a range of 1–3%. We do not expect overheads at these levels to have a significant impact on overall real-world hypervisor performance.

8 Related Work

The Low Level Virtual Architecture (LLVA) [4] was the first to explore the use of LLVM IR as the interface between software and the processor; it primarily focused on user-space computation. Later work [5] added support for vector instructions to the virtual instruction set. Extensions to LLVA (dubbed LLVA-OS) added instructions to support an entire

commodity operating system kernel (initially Linux [14] and later FreeBSD [10, 11]). SVA [13] built upon prior LLVA work to enforce security policies across the entire system stack; Criswell *et al.* [12] studied the security implications of the LLVA-OS (now dubbed SVA-OS) instructions in the context of enforcing memory safety. Later work on KCoFI [10] and Virtual Ghost [11] explored how the SVA-OS instructions could be misused to violate control flow integrity or to attack applications running on top of the OS kernel. Apparition [19] protects applications from page-fault and last-level-cache side-channel attacks launched by compromised OS kernels.

The early work on SVA-OS [10–12] demonstrates two key methods of preventing attacks via abuse of the SVA-OS instructions: preventing expression of malicious behavior via the interface exposed by the SVA-OS instructions and adding run-time checks to the implementation of the SVA-OS instructions to detect malicious behaviors. Our work applies these same principles to providing VMX support to software.

Other defenses for page-fault side-channel attacks can mitigate attacks launched by guest VMs, but they make different tradeoffs than Shade. T-SGX [34] encapsulates software running within an Intel SGX enclave within hardware transactions. If the protected software triggers a page fault or other trap, the transaction aborts, preventing the OS kernel from observing the trap. Apparition [19] and Shade induce lower overhead than T-SGX. Déjà Vu [8] enables applications running within Intel SGX to detect whether their execution time is longer than expected; such increases in execution time are assumed to be due to page-fault and similar attacks from the OS kernel. Shade prevents page-fault side-channel attacks whereas Déjà Vu only detects them. Unlike T-SGX [34] and Déjà Vu [8], Apparition [19] and Shade fix the root cause of page-fault side channels by preventing the OS kernel from reading and writing PTPs that map sensitive information.

Several approaches exist for mitigating LLC side channels. One approach is to partition the cache. Shade currently partitions the LLC statically to mitigate LLC side-channel attacks. The processor could provide support for dynamic cache partitioning algorithms like SecDCP [37] that could provide the same security but with better performance for workloads with changing cache needs. Another approach is to change the cache's refill and eviction policies. SHARP [40] modifies the processor's cache line eviction policy to minimize evictions within private caches that are caused by memory behavior on other cores that cause evictions of the victim's cache line in the shared caches. The Random Fill Cache Architecture [24] fetches random lines into the cache that are near to the cache line requested by software. These approaches could be used but, to the best of our knowledge, are not available on commodity processors. A third approach is to permit LLC side channels but to prevent software from measuring time accurately enough to use them. TimeWarp [25] modifies the processor so that the timestamp counter, performance counters, and interrupts do not yield enough timing information. However, it allows privileged software, such as the OS kernel, to read accurate time information and would therefore not work under Shade's threat model.

9 Future Work

Several directions exist for future work. First, we plan to port existing kernel-level hypervisor software to our SVA virtual instruction set. Candidates include FreeBSD's bhyve kernel driver [2] and VirtualBox [31]. Such a porting effort will help us study the performance effects of using a virtual instruction set on hardware VM performance when running real virtual machine workloads.

Second, we will investigate whether it is possible to protect guest VMs from the host OS just as Shade protects host applications from the host OS and guest VMs. Supporting features such as host virtualization of I/O devices will provide challenges as such features inherently give the host OS some control over guest VMs.

Third, Dharsee *et al.* [15, 16] studied security-critical processor bugs and devised a design that utilizes SVA's control over native code generation to prevent software from triggering known processor bugs. With our new SVA-OS instructions, we can build a system that prevents software from triggering bugs in Intel processors' VMX extensions [32, 33].

10 Conclusions

The Apparition [19] compiler-based virtual machine provides an efficient way to protect applications running under commodity operating systems from side-channel attacks by a compromised kernel, but cannot maintain those protections in the presence of a hypervisor driver within the OS to support hardware-based guest virtual machines. We present Shade, an extension of Apparition that adds support for hypervisors operating within the untrusted OS kernel without compromising Apparition's protection of user-space applications. Shade provides a virtual instruction set interface that allows hypervisor drivers to access the hardware virtualization support afforded by Intel's VMX processor extensions. We evaluated the performance impacts of our extensions and confirmed that the security checks performed by the new virtual instructions should not have a significant impact on hypervisor performance, nor on the performance of the existing Apparition system on the host. Our performance evaluation also uncovers and explores surprising and interesting performance impacts of the original Apparition system on kernel performance for protected host applications which were not discovered in prior work.

Acknowledgements

We thank the anonymous reviewers and Isaac Richter for their insightful feedback. This work was supported by NSF Awards CNS-1629770 and CNS-1618213.

References

[1] 2017. *AMD64 Architecture Programmer's Manual.* Advanced Micro Devices.

[2] 2018. *FreeBSD Handbook.* https://www.freebsd.org/doc/handbook/index.html Revision 52666.

[3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information Systems Security* 13, Article 4 (November 2009), 40 pages. Issue 1.

[4] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke. 2003. LLVA: A Low-level Virtual Instruction Set Architecture. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36).* IEEE Computer Society, Washington, DC, USA, 205–216. http://dl.acm.org/citation.cfm?id=956417.956545

[5] Robert L. Bocchino, Jr. and Vikram S. Adve. 2006. Vector LLVA: A Virtual Vector Instruction Set for Media Processing. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE '06).* ACM, New York, NY, USA, 46–56. https://doi.org/10.1145/1134760.1134769

[6] D. P. Bovet and Marco Cesati. 2003. *Understanding the LINUX Kernel* (2^{nd} ed.). O'Reilly, Sebastopol, CA.

[7] Stephen Checkoway and Hovav Shacham. 2013. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13).* ACM, New York, NY, USA, 253–264. https://doi.org/10.1145/2451116.2451145

[8] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. 2017. Detecting Privileged Side-Channel Attacks in Shielded Execution with DÉjÀ Vu. In *Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security (ASIA CCS'17).* 7–18. https://doi.org/10.1145/3052973.3053007

[9] John Criswell. 2014. *Secure Virtual Architecture: Security for Commodity Software Systems.* Ph.D. Dissertation. Computer Science Department, University of Illinois at Urbana-Champaign, Urbana, IL.

[10] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (SP'14).* San Jose, CA, 292–307. https://doi.org/10.1109/SP.2014.26

[11] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. Virtual Ghost: Protecting Applications from Hostile Operating Systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14).* 81–96. https://doi.org/10.1145/2541940.2541986

[12] John Criswell, Nicolas Geoffray, and Vikram Adve. 2009. Memory Safety for Low-level Software/Hardware Interactions. In *Proceedings of the 18th USENIX Security Symposium (Security'09).* 83–100. http://dl.acm.org/citation.cfm?id=1855768.1855774

[13] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. 2007. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07).* Stevenson, WA, 351–366. https://doi.org/10.1145/1294261.1294295

[14] John Criswell, Brent Monroe, and Vikram Adve. 2006. A Virtual Instruction Set Interface for Operating System Kernels. In *Workshop on the Interaction between Operating Systems and Computer Architecture.* Boston, MA, USA, 26–33.

[15] Komail Dharsee, Ethan Johnson, and John Criswell. 2017. *Hardware Vulnerability and Mitigation Survey.* Technical Report TR 1000. http://hdl.handle.net/1802/32871

[16] Komail Dharsee, Ethan Johnson, and John Criswell. 2017. A Software Solution for Hardware Vulnerabilities. In *2017 IEEE Cybersecurity Development (SecDev).* 27–33. https://doi.org/10.1109/SecDev.2017.18

[17] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. 2006. SAFECode: Enforcing Alias Analysis for Weakly Typed Languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation.* Ottawa, Canada.

[18] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan Cox, and Sandhya Dwarkadas. 2018. Spectres, Virtual Ghosts, and Hardware Support. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP'18).* ACM, New York, NY, USA, Article 5, 9 pages. https://doi.org/10.1145/3214292.3214297

[19] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L. Cox, and Sandhya Dwarkadas. 2018. Shielding Software From Privileged Side-Channel Attacks. In *Proceedings of the 27th USENIX Security Symposium (Security'18).* 1441–1458. https://www.usenix.org/conference/usenixsecurity18/presentation/dong

[20] Marcus Hähnel, Weidong Cui, and Marcus Peinado. 2017. High-Resolution Side Channels for Untrusted Operating Systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17).* USENIX Association, Santa Clara, CA, 299–312. https://www.usenix.org/conference/atc17/technical-sessions/presentation/hahnel

[21] Intel Corporation 2016. *Intel 64 and IA-32 Architectures Software Developer's Manual.* Intel Corporation.

[22] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, Vol. 1. Ottawa, Ontario, Canada, 225–230. https://www.kernel.org/doc/mirror/ols2007v1.pdf#page=225

[23] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO'04).* Palo Alto, CA, 75–86. http://dl.acm.org/citation.cfm?id=977395.977673

[24] Fangfei Liu and Ruby B. Lee. 2014. Random Fill Cache Architecture. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'14).* 203–215. https://doi.org/10.1109/MICRO.2014.28

[25] Robert Martin, John Demme, and Simha Sethumadhavan. 2012. TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-channel Attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12).* 118–129. http://dl.acm.org/citation.cfm?id=2337159.2337173

[26] Marshall Kirk McKusick, George V. Neville-Neil, and Robert N. M. Watson. 2015. *The Design and Implementation of the FreeBSD Operating System* (second ed.). Pearson Education.

[27] Larry McVoy and Carl Staelin. 1996. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the USENIX Annual Technical Conference (ATC).* San Diego, CA, 23–23. http://dl.acm.org/citation.cfm?id=1268299.1268322

[28] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09).* ACM, New York, NY, USA, 245–258. https://doi.org/10.1145/1542476.1542504

[29] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory management (ISMM '10).* ACM, New York, NY, USA, 31–40.

[30] Oracle Corporation. 2018. Oracle® VM VirtualBox® User Manual. https://www.virtualbox.org/manual/UserManual.html

[31] Oracle Corporation. 2018. VirtualBox. https://www.virtualbox.org

[32] Intel Corporation. 2017. 6th Generation Intel® Processor Family Specification Update. http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/desktop-6th-gen-core-family-spec-update.pdf

[33] Intel Corporation. 2017. 7th Generation Intel® Processor Family Specification Update. https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/7th-gen-core-family-spec-update.pdf

[34] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[35] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *Proceedings of the 26th USENIX Security Symposium (SEC'17)*. USENIX Association, 1041–1056. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck

[36] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP'93)*. Asheville, NC, 203–216. https://doi.org/10.1145/168619.168635

[37] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. 2016. SecDCP: Secure Dynamic Cache Partitioning for Efficient Timing Channel Protection. In *Proceedings of the 53rd Annual Design Automation Conference (DAC '16)*. ACM, New York, NY, USA, Article 74, 6 pages. https://doi.org/10.1145/2897937.2898086

[38] David A. Wheeler. 2014. SLOCCount. http://www.dwheeler.com/sloccount/

[39] Y. Xu, W. Cui, and M. Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy*. 640–656. https://doi.org/10.1109/SP.2015.45

[40] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. 2017. Secure Hierarchy-Aware Cache Replacement Policy (SHARP): Defending Against Cache-Based Side Channel Atacks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. 347–360. https://doi.org/10.1145/3079856.3080222