Spectres, Virtual Ghosts, and Hardware Support

Xiaowan Dong
University of Rochester
Rochester, NY
xdong@cs.rochester.edu

Zhuojia Shen
University of Rochester
Rochester, NY
zshen10@cs.rochester.edu

John Criswell
University of Rochester
Rochester, NY
criswell@cs.rochester.edu

Alan Cox
Rice University
Houston, TX
alc@rice.edu

Sandhya Dwarkadas
University of Rochester
Rochester, NY
sandhya@cs.rochester.edu

ABSTRACT

Side-channel attacks, such as Spectre and Meltdown, that leverage speculative execution pose a serious threat to computing systems. Worse yet, such attacks can be perpetrated by compromised operating system (OS) kernels to bypass defenses that protect applications from the OS kernel.

This work evaluates the performance impact of three different defenses against in-kernel speculation side-channel attacks within the context of Virtual Ghost, a system that protects user data from compromised OS kernels: Intel MPX bounds checks, which require a memory fence; address bit-masking and testing, which creates a dependence between the bounds check and the load/store; and the use of separate virtual address spaces for applications, the OS kernel, and the Virtual Ghost virtual machine, forcing a speculation boundary. Our results indicate that an instrumentation-based bit-masking approach to protection incurs the least overhead by minimizing speculation boundaries. Our work also highlights possible improvements to Intel MPX that could help mitigate speculation side-channel attacks at a lower cost.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; **Operating systems security**;

KEYWORDS

speculation-based side channels, operating systems security, secure computer architectures, compiler-based virtual machines

ACM Reference Format:
Xiaowan Dong, Zhuojia Shen, John Criswell, Alan Cox, and Sandhya Dwarkadas. 2018. Spectres, Virtual Ghosts, and Hardware Support. In *HASP '18: Hardware and Architectural Support for Security and Privacy, June 2, 2018, Los Angeles, CA, USA.* ACM, New York, NY, USA, Article 5, 9 pages. https://doi.org/10.1145/3214292.3214297

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 the author(s) 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.
HASP '18, June 2, 2018, Los Angeles, CA, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6500-0/18/06…$15.00
https://doi.org/10.1145/3214292.3214297

1 INTRODUCTION

Side channels leveraging speculative execution in processors [16, 18] pose a serious threat to computing systems. The Spectre attack [16] leverages speculative execution to trick a victim process into loading data into the processor and then leaking that data to the attacker via cache side channels. Meltdown [18] exploits the fact that memory protection checks are performed late in the processor pipeline to bypass supervisor-only page protections. While these attacks are launched by malicious user-space applications to steal data that is supposedly protected by the processor's memory management unit (MMU), speculative execution attacks can also defeat other methods of memory isolation, such as software fault isolation (SFI) [25]. Worse yet, such attacks pose a threat to systems like Virtual Ghost [5] that are supposed to protect applications from compromised operating system (OS) kernels. If an application's physical memory remains mapped in the virtual address space while the kernel is executing, then a compromised kernel can use variants of the Spectre [16] and Meltdown [18] attacks to read the contents of the application's memory even if hardware protection mechanisms (e.g., the MMU) or software protection mechanisms (e.g., SFI) are employed to prevent the kernel from reading the memory.

Virtual Ghost [5] is a compiler-based virtual machine that uses a combination of program rewriting techniques and hardware support to protect applications from a compromised OS kernel. Consequently, it can use both SFI and hardware support to mitigate speculation-based side-channel attacks. We evaluate the performance impact of two different defenses against in-kernel speculation side-channel attacks. The first approach, suggested by Intel [13], transforms the SFI code so that the instructions performing memory protection checks commit before memory read instructions start. The second approach creates separate virtual address spaces for the application, the OS kernel, and the Virtual Ghost virtual machine (VM) and configures them such that memory private to one component is not mapped while the other components execute. This approach is similar to defenses adopted by the Linux kernel [8, 9].

Results on our application benchmarks show that the average increase in execution time from the use of SFI using bit masking, SFI using MPX and `lfence`, and separate address spaces ranges from 1.00× to 2.43×, 1.05× to 4.46×, and 1.00× to 1.59×, respectively. SFI using bit masking to rewrite pointer values always performs better than SFI using MPX and `lfence` because `lfence` stalls the execution of subsequent instructions until prior instructions are

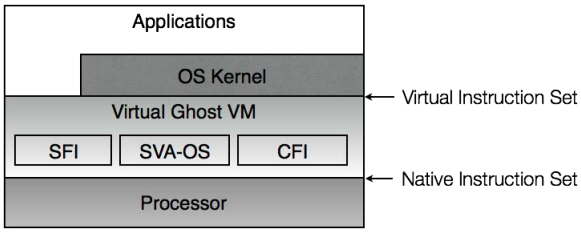

Figure 1: Virtual Ghost Architecture

completed locally. In contrast, SFI with bit masking alleviates the pipeline stall by converting the bounds check's control dependence into a data dependence. Specifically, the address targeted by the load has a data dependence on the bounds check, while the other speculative instructions can proceed.

To summarize, our contributions are as follows:

- We implement and evaluate an SFI approach that prevents the OS kernel from launching Spectre attacks [16]. We evaluate a variant that uses the Intel MPX features [12] with `lfence` and one that only uses bit-masking operations.
- We evaluate a method of using multiple page tables to prevent the OS kernel from using Spectre attacks [16].
- We find that on Virtual Ghost the SFI using bit-masking operations is generally the best performing way to defend against speculation side-channel attacks by the OS kernel. Both the separate-page-table approach, which was recently incorporated into commodity OS kernels [8, 9], and MPX with `lfence` suffer varying performance penalties due to serialization.
- We propose enhancements to the x86 ISA that would improve the performance of our defenses. We emulate these enhancements to demonstrate their benefits.

The rest of this paper is organized as follows. Section 2 presents background on Virtual Ghost [5], and Section 3 discusses Spectre attacks launched by compromised OS kernels. Section 4 describes our attack model. Section 5 presents the different methods of mitigating Spectre on Virtual Ghost. Section 6 describes our prototype implementation, Section 7 evaluates our defenses, and Section 8 discusses what hardware support would better aid the implementation of our defenses. Section 9 presents related work. Section 10 discusses future work and concludes.

2 VIRTUAL GHOST

Virtual Ghost [5] is a compiler-based virtual machine, built from Secure Virtual Architecture (SVA) [6], which protects applications from a compromised OS kernel. As Figure 1 shows, Virtual Ghost is interposed between the traditional software stack and the processor.

On Virtual Ghost systems [5], the OS kernel is compiled into a virtual instruction set (V-ISA) and then translated by the Virtual Ghost VM into the processor's native instruction set (N-ISA). The V-ISA extends the LLVM Intermediate Representation (IR) [17] with a set of instructions collectively called SVA-OS. The LLVM IR enables sophisticated static analysis on the OS kernel code while SVA-OS provides instructions for the OS kernel to use to configure hardware state (e.g., the MMU) and manipulate program state (e.g., context switching).

Virtual Ghost [5] provides three services to applications that want to protect themselves from the OS kernel:

- **Ghost Memory**: Virtual Ghost provides applications with memory that the OS kernel cannot read or write.
- **Protected State on Kernel Entry**: Virtual Ghost saves an application's CPU state into its own protected memory region when an interrupt, trap, or system call occurs. This feature protects both the confidentiality of the program's state as well as the integrity of its control flow.
- **Secure Encryption Key Delivery**: Virtual Ghost loads an application's private encryption and digital signing keys from persistent storage into the application's ghost memory to protect their confidentiality and integrity.

Virtual Ghost [5] partitions the address space of each process into four regions as Figure 2 depicts. There are the traditional user-space and kernel-space memory regions; both the application and the OS kernel can access the former but only the OS kernel can access the latter. Each process gets its own private user-space memory while kernel-space memory is shared among processes [19]. Ghost memory is memory that only user-space code can read or write; Virtual Ghost prevents the OS kernel from reading and writing ghost memory and from changing the page table entries that map ghost memory. Like user-space memory, each process has its own private ghost memory, which is only mapped when the process is running. Finally, there is Virtual Ghost VM memory: this is memory that only the Virtual Ghost VM, implementing the SVA-OS instructions, can read and write. The Virtual Ghost VM stores its own internal data structures, interrupted program state that it saves on interrupts, traps, and system calls, as well as kernel thread state saved on context switches, in the Virtual Ghost VM memory. Interrupts, traps, and system calls are first handled by the Virtual Ghost VM, which saves the interrupted program state and zeros out registers (except for those loaded with system call arguments) before handing control over to the OS kernel.

As Virtual Ghost [5] allows the OS kernel to read page tables, it does not place the page tables in Virtual Ghost VM memory. Instead, it maps page table pages as read-only memory in the OS page table and makes the OS use SVA-OS instructions to modify them, thereby preserving the integrity of the page table pages. However, enhancements to Virtual Ghost [7] protect the confidentiality of the page tables for *ghost memory* as Section 5.4 describes.

Unlike existing systems, Virtual Ghost [5] prevents user-space, kernel-space, and ghost memory from being configured as executable; they do not contain executable native code. Instead, Virtual Ghost puts the executable code, either code translations from V-ISA or code segments of N-ISA applications, in the Virtual Ghost VM memory region. Pages containing native code are mapped as execute-only while all other parts of the region are inaccessible to both applications and the OS kernel.

With these features, applications on Virtual Ghost systems can actively protect themselves from the OS kernel: they can store all their data inside ghost memory to prevent theft and tampering and can use encryption and digital signatures to maintain data

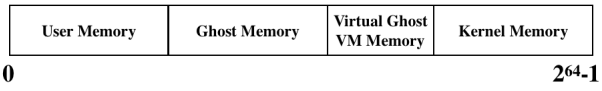

Figure 2: Virtual Ghost Address Space Layout

confidentiality and integrity when transmitting data through the OS kernel's I/O systems.

Virtual Ghost [5] employs compiler techniques to enforce security policies on the OS kernel, such as SFI [25] and control-flow integrity (CFI) [1]. SFI, as Sections 5.1 and 5.2 discuss, prevents the OS kernel from reading and writing ghost memory and Virtual Ghost VM memory while CFI guarantees that the kernel does not bypass the SFI instrumentation.

3 OS KERNEL SPECTRE ATTACKS

Speculation [10] is a technique in which a processor executes a set of instructions before knowing if those instructions should execute; the processor rolls back the effects of speculatively executed instructions if it determines that it should not have executed them. Speculation side-channel attacks [16, 18] cause the processor to speculatively read data (which the process may or may not have permission to access) whose value is exposed by further speculative execution of instructions that create a cache side channel. Spectre [16], for example, tricks a victim user-space process into speculatively assuming a branch is taken, resulting in an out-of-bounds array access that loads data into the cache. The address of this load can subsequently be observed by the attacker via a cache side channel, thereby exposing the value used to compute that address. Meltdown [18] exploits a flaw in which the processor checks MMU memory protections late in the processor pipeline. With Meltdown, a user-space process speculatively loads information from the OS kernel's memory before the processor performs the memory protection check and uses a cache side channel to leak the data to the attacking user-space process.

While the existing Spectre attack [16] is one user-space process attacking another user-space process, it is possible for a compromised OS kernel to use similar techniques to read ghost memory and Virtual Ghost VM memory. For efficiency, Virtual Ghost [5] keeps ghost memory and Virtual Ghost VM memory mapped into the virtual address space while the kernel is running and uses SFI to prevent the kernel from accessing ghost memory and Virtual Ghost VM memory. The SFI instrumentation adds code prior to every instruction that reads and writes memory to ensure that the instruction only accesses user-space or kernel-space memory. At the native code level, if the SFI instructions do not ensure that the SFI check has completed prior to a memory read instruction, then the kernel can speculatively load data from ghost memory or Virtual Ghost VM memory and leak it via a cache side channel. This variation of a Spectre attack is similar to Meltdown [18] in that it accesses memory that is currently mapped in the OS kernel's address space that it normally cannot access. Unlike Meltdown, this attack does not leverage an MMU check that is performed late in the pipeline. Rather, like existing Spectre attacks [16], it uses speculation to bypass instructions in the code segment that would otherwise restrict the program's access to memory.

4 ATTACK MODEL

Our attack model assumes that an attacker can compromise the OS kernel or replace it with intentionally malicious software. Since we assume that the system is using Virtual Ghost [5], a compromised OS kernel cannot read or write ghost memory or modify application control flow. We further assume that the attacker wishes to steal information; integrity and availability attacks are out of scope. Therefore, we assume that the OS kernel will want to use speculative execution side channels to steal private application information. Other side-channel attacks are possible but outside the scope of this work. We assume that processor protection features are implemented correctly from the perspective of the instruction set architecture (ISA); speculation side channels are the only flaw in the hardware.

Under these conditions, our defenses must prevent a compromised OS kernel from using the variations of Spectre attacks [16] presented in Section 3 to read Virtual Ghost VM memory and ghost memory. Additionally, since the OS kernel can create and execute new user-space programs, our defenses must prevent application code from using Spectre [16] and Meltdown [18] attacks to read Virtual Ghost VM memory. Since Virtual Ghost [5] and the applications executing thereon treat the OS kernel as untrusted code, applications wanting to keep their data private should store it in their own ghost memory and encrypt it when sending it through the OS kernel to perform I/O.

5 MEMORY PROTECTIONS

There are two broad approaches for preventing an OS kernel from launching Spectre attacks [16]. The first is to use an SFI technique that forces the processor pipeline to commit the SFI instructions before executing memory read instructions. This SFI protection must be enforced on both OS kernel and application code; this requires that OS kernel and application code be shipped as V-ISA code. The second is to unmap the ghost memory and Virtual Ghost VM memory from the virtual address space while the OS kernel is running and to unmap the Virtual Ghost VM memory while applications are running (we also unmap kernel-space memory like previous work [8, 9] as it mitigates user-level Meltdown attacks [18] and induces no additional overhead to our defense).

We explore three approaches of mitigating speculation side channels on Virtual Ghost:

(1) **Bit-Masking SFI**: We employ the SFI method in the original Virtual Ghost [5] on the kernel, which uses bit masking to ensure that addresses used in memory reads point into either user-space or kernel-space memory. This version, coincidentally, creates data dependencies that force the processor to delay memory loads until the SFI instructions commit.
(2) **MPX SFI with Lfence**: We employ a new SFI approach that uses the Intel MPX bounds checking instructions [12] to verify that pointers point into either user-space or kernel-space memory [7]. According to Intel [14], adding an `lfence` between an MPX check and a memory read instruction stalls the pipeline to prevent Spectre [16] attacks. Note that MPX SFI will need the `lfence` only before loads and not before stores since we are concerned with the speculative read exposing protected data values. Speculative writes do not load

```
movq   %r9, %rbx
shrq   $0x20, %rbx
andl   $0xfffffd00, %ebx
cmpq   $0xfffffd00, %rbx
sete   %bl
movzbl %bl, %ebx
shlq   $0x29, %rbx
orq    %r9, %rbx
movb   (%rbx), %bl
```

(a) SFI Using Bit-Masking Operations

```
movabsq $0x10000000000, %rax
addq    %rsi, %rax
bndcl   %rax, %bnd0
lfence
movq    (%rsi), %rdx
```

(b) SFI Using MPX

```
cmpq   $4095, %rdi
setbe  %r15b
movzbq %r15b, %r15
orq    %r15, %rdi
movb   (%rdi), %al
```

(c) SFI Using EMPX

Figure 3: SFI Code Snippets (AT&T Syntax)

any protected data into the processor registers; both Spectre and Meltdown attacks depend upon loading protected data into a processor register in order to leak it via a covert channel [16, 18].

(3) **Separate Virtual Address Spaces**: We employ a method that creates separate pages tables for the application, the OS kernel, and the Virtual Ghost VM.

Modern OS kernels maintain a data structure dubbed the *direct map*. The direct map is a one-to-one mapping of contiguous virtual addresses to contiguous physical memory addresses which the OS kernel uses to quickly find a virtual address mapped to a physical address to which it can read and write [4]. A compromised OS kernel could use a Spectre [16] or Meltdown [18] attack to read the contents of the direct map, bypassing our protections. Therefore, after describing our three defenses, we explain how we protect the direct map by moving it into Virtual Ghost VM memory.

5.1 Bit-Masking SFI

The SFI instrumentation in the original Virtual Ghost [5] uses simple bit masking to check whether a pointer points into the Virtual Ghost VM memory or ghost memory regions and to move that pointer into the kernel-space region if so; since the OS kernel should only read and write user-space and kernel-space memory, moving the pointer into kernel space causes an incorrect OS kernel to behave securely but still incorrectly. This SFI instrumentation uses data dependence instead of branches and bounds checking instructions to protect memory. Figure 3a shows an x86 assembly code snippet that we use to perform an SFI check in the latest Virtual Ghost implementation before each memory read. This code checks whether the input pointer points into either the Virtual Ghost VM memory or ghost memory (these are placed contiguously in the virtual address space). If the pointer is within these two memory regions, it sets a bit in the pointer to move it into the kernel-space memory region.

In Figure 3a, the pointer to check is stored in register `%r9`. The SFI instrumentation works as follows:

(1) It copies the original pointer value from `%r9` to `%rbx`.
(2) It extracts the high-order 24 bits of the pointer value and determines whether it equals `0xfffffd` (the value of the high order bits of any address within the ghost memory or Virtual Ghost VM memory region).
(3) The `sete` and `movzbl` instructions place `0x1` into `%ebx` if the pointer is within the ghost memory or Virtual Ghost VM memory region or `0x0` otherwise. The `shlq` instruction transforms a `0x1` value into `0x20000000000`.
(4) If the original pointer needs to be moved out of ghost memory or Virtual Ghost VM memory, the `orq` instruction will mask the pointer value with `0x20000000000`, generating a pointer into the kernel-space memory region. If the original pointer pointed into user-space or kernel-space memory, then it is masked with `0x0` and remains unchanged.

Intel states that using a `sete` instruction constrains speculation [14]. Additionally, the bit-masking instructions create a data dependence between the SFI code and the address of the protected load; the processor must compute the result of the SFI-protected pointer *before* the memory read instruction, thereby preventing Spectre [16] attacks.

5.2 MPX SFI with Lfence

We utilize Intel MPX [12], a hardware mechanism originally designed for efficient pointer bounds checking, to implement SFI. MPX provides four bounds registers; each register maintains the lower and upper bounds of a memory object. Software is instrumented to use bounds checking instructions that check whether a pointer is within the bounds stored within a specified bounds register; these instructions generate a trap if the pointer is out of bounds.

Apparition [7] is an enhanced version of Virtual Ghost [5] that uses MPX instructions for its SFI implementation. Instead of storing the bounds of individual memory objects within the bounds registers, Apparition [7] stores the bounds of an entire *memory region* within a bounds register. Specifically, Apparition treats user-space and kernel-space memory as a single large memory region and checks whether each OS kernel load and store is within the bounds of this memory region. As there is only one memory region, only one bounds register is needed. One difficulty is that the user-space memory and kernel-space memory are not contiguous; ghost memory and Virtual Ghost VM memory reside between them as Figure 2 shows. To address this issue, each run-time check before a load or store subtracts the starting address of ghost memory from the value of the pointer being dereferenced, thus making the user- and kernel-space memory appear contiguous. MPX can then be used by setting the lower and upper bounds in the bounds register to the

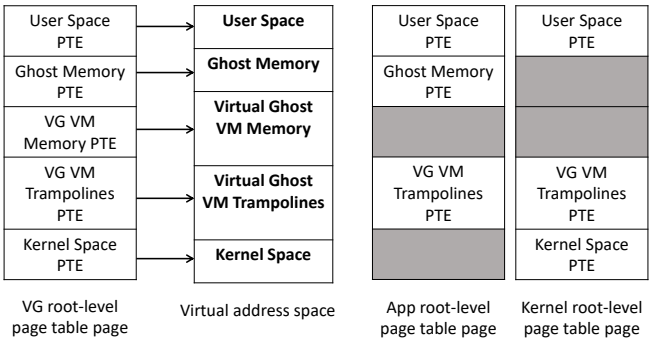

Figure 4: The separate root-level page table pages

remapped values of the start of kernel-space memory and the end of user-space memory, respectively.

Intel processors assume that most bounds checks will pass and will therefore speculatively execute instructions after an MPX bounds check instruction [14]. To prevent a Spectre attack from reading out-of-bounds data, we modify Virtual Ghost to insert an `lfence` instruction between the MPX bounds check instruction and the memory access as Intel suggests [13, 14]. The `lfence` instruction acts as a speculation barrier, preventing younger instructions, including memory reads, from executing speculatively before older instructions, including the MPX bounds check, retire [13, 14].

Figure 3b lists instructions in an MPX SFI check. The `movabsq` and `addq` instructions subtract a constant from the pointer value stored in `%rsi` to make user-space and kernel-space memory appear contiguous; the MPX bounds check instruction (`bndcl`) follows. The subsequent `lfence` instruction ensures that the processor pipeline stalls until the bounds check has completed and the processor knows that it will not generate a trap. At that point, the `movq` instruction that performs the actual memory read can commence.

5.3 Separate Virtual Address Spaces

Another defense for Spectre attacks [16] is to execute the application, the OS kernel, and the Virtual Ghost VM within their own separate virtual address spaces. This defense requires three address spaces that unmap memory needing protection as follows:

- **VG-AddrSpace**: The Virtual Ghost VM maps all four memory regions depicted in Figure 2.
- **App-AddrSpace**: Applications map only user-space memory and the application's ghost memory.
- **Kernel-AddrSpace**: The kernel maps only user-space and kernel-space memory.

While the OS kernel is executing, only the Kernel-AddrSpace can be used. The App-AddrSpace is used when the user-space application is running; the VG-AddrSpace translations are active only when the Virtual Ghost VM is executing.

We modified Virtual Ghost [5] to create three separate virtual address spaces for each process by maintaining separate page tables for the VG-AddrSpace, App-AddrSpace, and Kernel-AddrSpace as Figure 4 depicts. For architectures that use hierarchical page tables e.g., x86 [12], we strategically place the ghost memory and Virtual Ghost VM memory regions in the virtual address space so that modifying one entry in the root-level page table page can unmap the region. This approach reduces the number of additional page table pages required; the Virtual Ghost VM only needs two additional root-level PTPs for the App-AddrSpace and Kernel-AddrSpace in addition to the existing root PTP for the VG-AddrSpace. We add a new fifth memory region, named *Virtual Ghost VM trampolines*, which contains the minimal subset of Virtual Ghost VM code required to switch address spaces; it is mapped by a separate root-level page table entry in all three address spaces.

Virtual Ghost [5] prevents the OS kernel from modifying page tables directly; instead, the Virtual Ghost VM provides SVA-OS instructions that the OS kernel can use to insert, update, or remove page table entries (PTEs) at each level of a hierarchical page table. We modified the relevant SVA-OS instructions to propagate any changes to the root-level PTEs of a memory region to all the root-level PTPs mapping the memory region. These SVA-OS instructions also ensure that the root-level entries of the ghost memory regions can only be populated in the VG-AddrSpace and App-AddrSpace, and the root-level entries of the Virtual Ghost VM Memory can only be inserted in the VG-AddrSpace.

Whenever switching address spaces, the Virtual Ghost VM must load the page table base register (PTBR) with the physical address of the correct root-level PTP. For SVA-OS instructions, the Virtual Ghost VM loads the root-level PTP of the VG-AddrSpace into the PTBR and then restores the Kernel-AddrSpace root-level PTP into the PTBR before returning control to kernel code. For system calls, traps, and interrupts, the Virtual Ghost VM loads the root-level PTP of the VG-AddrSpace into the PTBR so that it can save the CPU state into Virtual Ghost VM memory where it will be protected from applications and the OS kernel. It then switches to the Kernel-AddrSpace and calls the OS kernel's handler for the system call, trap, or interrupt.

To alleviate Translation Look-aside Buffer (TLB) flushes when switching address spaces, we assign different address space identifiers (ASIDs) to the OS kernel, the application, and the Virtual Ghost VM. We also enhanced the SVA-OS MMU instructions to ensure that ghost memory and Virtual Ghost VM memory are never mapped with page table entries marked with the global bit [12]. On x86, both the PTBR and the ASID are in Control Register 3 (CR3) [12]. Modifying either the PTBR or ASID is a serializing instruction [12], resulting in additional latency on every switch among the App-AddrSpace, the Kernel-AddrSpace, and the VG-AddrSpace.

Two x86 features prevent fully correct implementation of multiple address spaces. First, the x86 saves several registers e.g., program counter, stack pointer, and processor status register, into memory before handing control over to the interrupt handling routine [12]. This creates a "chicken and egg" problem: software must switch to a different virtual address space in order to save interrupted program state in Virtual Ghost VM memory, but the processor attempts to save the state before software can switch to the virtual address space that maps Virtual Ghost VM memory. Second, the x86 requires that a general purpose register be free in order to change the ASID and page table pointer in CR3; the only instruction that can modify a control register reads its input from a general purpose register, and no general purpose registers are free when the interrupt handling code begins execution [12].

Two simple changes would alleviate these difficulties. First, the processor should save the registers that are currently saved into memory on an interrupt, trap, or system call into special registers on the processor (as ARM does [2]) instead of saving them to memory. This change would allow software to switch to a different virtual address space before saving interrupted program state into Virtual Ghost VM memory. Second, one or more general purpose registers should be reserved for trusted code to use to switch virtual address spaces, alleviating the need to save general purpose registers before performing the virtual address space switch.

5.4 Virtual Ghost Internal Direct Map

A direct map is a region of contiguous virtual memory that is mapped to a contiguous range of physical memory [4]. The OS kernel uses a direct map to compute a virtual address that is mapped to a specified physical address in constant time via bitwise OR operations on the physical address [4]. Apparition [7] extends Virtual Ghost [5] by creating a direct map within the Virtual Ghost VM Memory that only the Virtual Ghost VM can use. This new direct map provides write access to all the physical memory. Apparition unmaps physical memory frames used for ghost memory, Virtual Ghost VM memory, and page tables mapping ghost memory and Virtual Ghost VM memory from the OS kernel's direct map. Such frames only appear in Apparition's internal direct map, and the SVA-OS instructions maintain this restriction. Combined with our defenses, this internal direct map prevents the kernel and applications from gleaning confidential information from the original direct map via speculation side channels.

6 IMPLEMENTATION

Our work modifies the Virtual Ghost [5, 7] prototype for x86-64 systems and uses the FreeBSD 9.0 kernel ported to the SVA virtual instruction set. This prototype only supports single-processor execution. The kernel is translated from V-ISA to N-ISA code and instrumented during static compilation; online translation of V-ISA to N-ISA code and moving N-ISA code into the Virtual Ghost VM memory is not implemented. Virtual Ghost [5] instruments loads, stores, atomic operations, and calls to `memset()` and `memcpy()` at the LLVM IR level; it does not instrument calls to functions that copy data between user and kernel memory [19]. Since we modified our kernel to use constant-sized stack frames, the stack pointer is never loaded from memory, removing the need for bounds checks on reads from stack spill slots. We modified the SFI MPX pass so that it inserts `lfence` instructions before every load instruction, atomic instruction, and `memcpy()` call. We do not modify the bitmasking SFI instrumentation as it already inserts instructions e.g., `sete` [14], that constrain speculation prior to memory access instructions. The CFI instrumentation uses a bitwise OR instruction to ensure that control-flow targets are within kernel memory, and it folds the memory read that loads CFI labels into an x86 compare instruction. These two features prevent the kernel from using the memory reads added by CFI to launch Spectre attacks.

We only instrument the OS kernel with the SFI instructions to defend against speculation side-channel attacks performed directly by the OS kernel. We therefore evaluate OS kernel performance and leave application performance for future work. To make a fair comparison between the separate-address-spaces approach and the SFI approaches, using two address spaces, the VG-AddrSpace and Kernel-AddrSpace as described in Section 5.3, is sufficient to mitigate against kernel-side speculation side channels, where the application and the Virtual Ghost VM share the VG-AddrSpace.

Due to the x86 limitations described in Section 5.3, we mimic the behavior of using separate page tables by creating identical copies of the top-level page table page for VG-AddrSpace and Kernel-AddrSpace. We then modified the Virtual Ghost VM so that transitions between the VG-AddrSpace and Kernel-AddrSpace change page table pages and switch to separate address space identifiers (which are called *Process Context Identifiers*, or *PCID*s [12]). This causes the OS kernel and the Virtual Ghost VM to use separate page tables and TLB entries.

7 EVALUATION

7.1 Methodology

We ran the experiments on a Dell Precision T3620 workstation with a 3.40 GHz Intel® Core™ i7-6700 hyperthreading quad-core processor, 16 GB of RAM, an Intel® E1000 network card, a 256 GB Solid State Drive (SSD), and a 7,200 RPM 500 GB hard disk. For our network experiments, we used a dedicated Gigabit Ethernet network with a Dell T1700 Precision workstation running FreeBSD 9.3 as the client machine; this machine has a 3.40 GHz Intel® i7-4770 hyperthreading quad-core processor with 16 GB of RAM. We ran our experiments with the OS in single-user mode to alleviate noise from other processes on the system. We stored all files for our experiments on the SSD.

Besides the baseline FreeBSD 9.0 kernel, we conducted our experiments on the FreeBSD SVA kernels executing on the following configurations of Virtual Ghost [5] coupled with the SVA internal direct map (described in Section 5.4):

(1) **SFI-MPX-lfence**: Virtual Ghost implementing SFI with MPX and `lfence`. The `lfence` stops all younger instructions from executing, even speculatively, before the older instructions retire.
(2) **SFI-arith**: Virtual Ghost employing traditional SFI using bit-masking operations. The speculation boundary is only between the check and the use of the pointer; other speculative instructions can proceed.
(3) **AS**: Virtual Ghost using separate virtual address spaces. Address space switching occurs frequently since each kernel invocation of an SVA-OS instruction (as described in Section 2) triggers two address space switches: one from Kernel-AddrSpace to VG-AddrSpace and the other from VG-AddrSpace back to Kernel-AddrSpace. For traps, interrupts and system calls, the trap handling code within the Virtual Ghost VM must first execute in VG-AddrSpace before switching to Kernel-AddrSpace to execute the kernel trap handlers. In addition, modification of CR3.PTBR and CR3.PCID are expensive serializing instructions.
(4) **SFI-MPX**: Virtual Ghost implementing SFI with MPX but without `lfence`.

Test	Native (μs)	Std. dev. (μs)	Overhead (×)				
			SFI-MPX lfence	SFI-arith	AS	SFI-MPX	SFI-EMPX
null syscall	0.1	0.0	5.3	2.9	7.0	2.6	2.8
open/close	1.8	0.0	6.0	2.3	1.9	1.9	2.1
mmap	5.7	0.1	6.2	3.5	4.6	2.8	3.2
page fault	31.5	1.7	1.1	1.1	1.1	1.1	1.0
fork + exit	50.2	0.1	4.2	2.2	3.9	1.9	2.2
fork + exec	55.0	0.2	4.1	2.2	3.7	1.9	2.2
Average	-	-	4.5	2.4	3.7	2.0	2.3

Table 1: LMBench Results

(5) **SFI-EMPX**: Virtual Ghost implementing SFI to emulate the MPX improvements suggested in Section 8. We propose hardware improvements to MPX to gain the performance of SFI-MPX with the security of SFI-arith.

7.2 Microbenchmarks

We used LMBench [20] to measure the overhead of Virtual Ghost with the various mitigations on various system calls. We chose the process latency benchmarks that would impact application performance most and measure the performance of the OS kernel instead of the hardware. We used 1,000 repetitions for the benchmarks for which the number of repetitions is configurable. We report the averages of the elapsed time output by the benchmarks over 10 rounds of execution.

Table 1 reports the overhead of Virtual ghost with the various mitigations. The overheads of SFI-MPX-lfence, SFI-arith, AS and SFI-MPX are 4.5×, 2.4×, 3.7×, and 2.0× on average, respectively, across all the system calls tested. Our results show that SFI-arith outperforms SFI-MPX-lfence and AS for all the system calls except for `open/close` and `page fault`. AS is slower due to the frequent address space switches and the expensive serializing instructions. SFI-MPX-lfence has the worst performance of the three defenses for all the system calls except for `null syscall` due to the expense of using an `lfence` speculation barrier. SFI-MPX-lfence is more than 2 times slower than SFI-MPX on average. AS shows the worst overhead on the `null syscall` benchmark relative to SFI-Arith and SFI-MPX-lfence since no work is performed in the system call, making the overhead of address space switches a larger proportion of the execution time than in the other benchmarks. For `open/close`, AS is faster than SFI-arith, where the overhead of address space switches is smaller than the costs of instrumenting every kernel load and store with bit-masking instructions. The three defenses incur similar overhead on `page fault` when accounting for standard deviation. In addition, the overhead incurred by SFI-EMPX is similar to SFI-arith across all the system calls tested.

7.3 Libc Compilation Performance

We studied our defenses' overheads when compiling the FreeBSD 9.0 C library. To measure compilation time, we read the x86 Time Stamp Counter (TSC) right before and after the execution of the `make` command, took the difference between the two, and divided the result by the processor frequency to compute execution time. We ran each experiment 10 times and report the averages in Table 2. SFI-arith is the fastest option among the three mitigations, incurring an overhead of 1.17×. AS is slower than SFI-arith due to the high frequency of address space switches. SFI-MPX-lfence performs the worst due to the use of `lfence`; adding an `lfence` to the MPX check increases the overhead from 1.12× to 1.44×. SFI-EMPX incurs similar overhead as SFI-arith and SFI-MPX without `lfence`.

7.4 Postmark Performance

To analyze the performance impact of the defenses on the file system, we used `Postmark` [22], which mimics the behavior of a mail server and exercises the file system intensively. We configured `Postmark` to use 500 files ranging in size from 500 B to 9.77 KB. We also configured it to use Unix buffered file I/O and a 512 B block size for reads and writes. The read/append and create/delete biases were set to 5. We performed 500,000 transactions in each run.

Table 2 reports the averages over 20 rounds of execution. AS incurs the smallest overhead (1.58×), outperforming SFI-arith (2.43×) and SFI-MPX-lfence (4.46×). The cost of AS is smaller than the cost of SFI-arith, which indicates the overhead of address space switches is less expensive than instrumenting all the kernel load and stores. `Lfence` increases the overhead of SFI-MPX significantly, from 1.63× to 4.46×. SFI-EMPX incurs 13.17% less overhead compared to SFI-arith and 29.45% more overhead than SFI-MPX without `lfence`.

7.5 OpenSSH Performance

To analyze our defenses' effects on network I/O, we measured the bandwidth of the OpenSSH [24] server running on our defenses serving files to a FreeBSD client. We used the OpenSSH server on our test machine and the OpenSSH `scp` client on the FreeBSD 9.3 machine described in Section 7.1. We created a set of files of varying sizes before running the experiment; we filled the contents of each file with random numbers generated by the `/dev/random` device on our test machine.

We ran each experiment 20 times and report the averages. Figure 5 illustrates the average file transfer rates on the baseline FreeBSD 9.0, and Figure 6 shows the overhead of the mitigations. SFI-MPX-lfence has the worst performance, incurring overhead ranging from 1.05× to 1.97× across all file sizes due to the expensive `lfence`. In contrast, the SFI-MPX overhead without `lfence` is 1.00× to 1.34×. SFI-arith is a clear winner over the other two defenses with overhead ranging from 1.04× to 1.40× when transferring 1 KB to 4 MB files, whereas the overhead of AS is larger, from 1.05× to 1.59×. AS is slower than SFI-arith due to the frequent and expensive address space switches. For large files ranging from 8 MB to 512 MB, the overheads of SFI-arith, AS and SFI-MPX are negligible. SFI-EMPX performs similarly as SFI-arith and is slightly slower than SFI-MPX for files from 1 KB to 128 KB. For 256 KB

App	Native (s)	Std. Dev. (s)	Overhead (×)				
			SFI-MPX lfence	SFI-arith	AS	SFI-MPX	SFI-EMPX
Libc compilation	67.68	0.64	1.44	1.17	1.22	1.12	1.15
Postmark	8.00	0.55	4.46	2.43	1.58	1.63	2.11

Table 2: Postmark and Libc Compilation Results

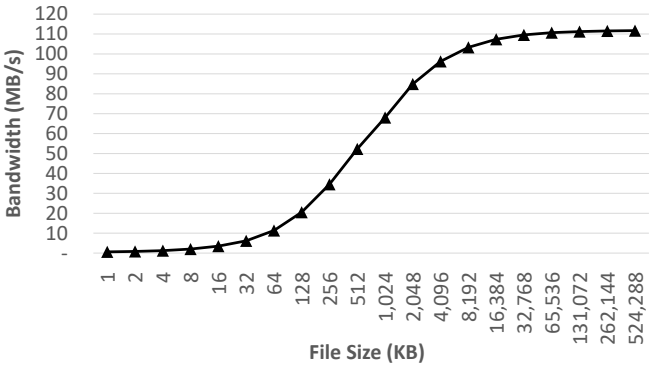

Figure 5: SSH Server Transfer Rate on Native FreeBSD

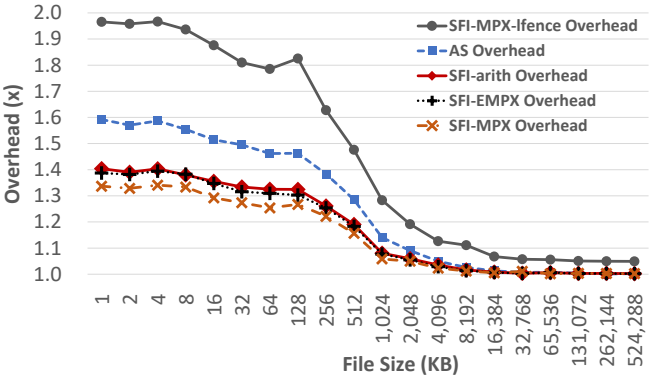

Figure 6: Overheads on SSH Server Transfer Rate

to 512 MB files, SFI-EMPX, SFI-arith and SFI-MPX incur similar overhead.

8 HARDWARE IMPROVEMENTS

As Section 7 shows, MPX provided the best performance prior to the addition of `lfence` to mitigate Spectre [16] attacks. We propose two improvements to MPX that could provide good performance and mitigate Spectre attacks.

First, adding a condition code dependency between the MPX bounds checking instruction and the memory access instruction would help eliminate the need for an `lfence`. For example, instead of triggering a fault, the MPX bounds checking instructions could simply set a bit in a register indicating that the bounds check failed; the subsequent memory access instructions could potentially be conditionally executed based on the value of this bit, generating a trap if the bit is set.

Second, to allow pointers to point into multiple regions of the virtual address space, the MPX bounds checking instructions could be enhanced to take an operand that specifies a set of bounds registers against which to check the pointer; an out-of-bounds result only occurs if the pointer is not within the bounds stored within any of the specified bounds check registers. This change would decrease register pressure and instruction count by alleviating the pointer arithmetic needed to make user-space and kernel-space appear contiguous.

Figure 3c shows x86 assembly code that mimics SFI that uses our proposed MPX improvements. The code checks the pointer stored in `%rdi` against a constant value using the `cmpq`. It then sets a bit in the pointer if it is out of bounds using the `setbe` instruction; this creates the data dependency between the check and the `movb` which reads memory. Our results show that this emulated SFI has similar performance to the fastest of the three defenses, as Section 7 shows.

9 RELATED WORK

Intel [14], Microsoft [21], and ARM [3] leverage speculation barriers (such as `cpuid`, `lfence`, `CMOVcc`, and `SETcc`) to mitigate Spectre [16] attacks that bypass bounds checking. Kernel page-table isolation (KPTI or KAISER) modifies the Linux kernel to mitigate Meltdown [8, 9, 18] by unmapping kernel-space memory when user-space code is running on the processor. Microsoft Windows 10 has similar features [15]. These defenses do not mitigate speculation side-channel attacks performed by privileged code.

Intel suggests that future processors should be able to use memory protection keys (MPK) and supervisor-mode access prevention (SMAP) [12] to restrict the memory that might be used to create cache side channels.

Red Hat found the kernel patches for mitigating Meltdown and Spectre launched by user-space attackers incurred 1% to 8% overhead on the wide range of applications tested, including HPC, JavaVM and database benchmarks [11]. Nikolay et al. evaluated the overhead of these kernel patches on HPC applications and found their overhead was 2% to 3% for single node jobs and 5% to 11% for two node jobs [23].

10 FUTURE WORK AND CONCLUSIONS

SFI with bit masking generally has the best performance of our three defenses; SFI with MPX and `lfence` has the worst performance. Average application execution time overheads for SFI with bit masking, MPX coupled with `lfence`, and separate address spaces range from 1.00× to 2.43×, 1.05× to 4.46×, and 1.00× to 1.59×, respectively. Our hypothetical SFI utilizing our MPX improvements has similar or better performance compared to bit-masking SFI.

There are several directions for future work. First, we can explore the use of type-safety optimizations [6] to remove unneeded speculation barriers and evaluate the development cost of such optimizations. Second, we will prototype and evaluate our proposed improvements to Intel MPX [12].

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments and suggestions. This work was funded by NSF Awards CNS-1319353, CNS-1618497, CNS-1618588, CNS-1629770, and CNS-1652280.

REFERENCES

[1] 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.
[2] ARM. 2014. *ARM Architecture Reference Manual: ARMv8, for ARMv8-A Architecture Profile.*
[3] ARM. 2018. ARM speculation barrier header. https://github.com/ARM-software/speculation-barrier.

[4] D. P. Bovet and Marco Cesati. 2006. *Understanding the LINUX Kernel* (3^{rd} ed.). O'Reilly, Sebastopol, CA.

[5] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. Virtual Ghost: Protecting Applications from Hostile Operating Systems. In *Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems*.

[6] 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 ACM SIGOPS Symposium on Operating System Principles*. Stevenson, WA, USA.

[7] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan Cox, and Sandhya Dwarkadas. 2018. Shielding Software From Privileged Side-Channel Attacks. To appear in the 27th USENIX Security Symposium.

[8] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR is Dead: Long Live KASLR. In *Engineering Secure Software and Systems*. Springer International Publishing, Cham.

[9] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 368–379. https://doi.org/10.1145/2976749.2978356

[10] John L. Hennessy and David A. Patterson. 2002. *Computer Architecture: A Quantitative Approach* (3^{rd} ed.). Morgan Kaufmann, San Francisco, CA.

[11] Red Hat Inc. 2018. Speculative Execution Exploit Performance Impacts - Describing the performance impacts to security patches for CVE-2017-5754 CVE-2017-5753 and CVE-2017-5715. https://access.redhat.com/articles/3307751.

[12] Intel. 2016. Intel 64 and IA-32 Architectures Software Developer's Manual. (September 2016).

[13] Intel. 2018. *Intel Analysis of Speculative Execution Side Channels*. Technical Report 336983-001.

[14] Intel. 2018. *Speculative Execution Side Channel Mitigations*. Technical Report 336996-001.

[15] Alex Ionescu. 2017. Windows 17035 Kernel ASLR/VA Isolation In Practice (like Linux KAISER). (2017).

[16] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. (2018).

[17] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proceedings of the Conference on Code Generation and Optimization*. San Jose, CA, USA, 75–88.

[18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. (2018).

[19] 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.

[20] Larry McVoy and Carl Staelin. 1996. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference (ATC'96)*. USENIX Association, Berkeley, CA, USA, 16. http://dl.acm.org/citation.cfm?id=1268299.1268322

[21] Andrew Pardoe. 2018. Spectre mitigations in MSVC. https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/.

[22] Postmark. 2013. Email delivery for web apps. https://postmarkapp.com/.

[23] Nikolay A. Simakov, Martins D. Innus, Matthew D. Jones, Joseph P. White, Steven M. Gallo, Robert L. DeLeon, and Thomas R. Furlani. 2018. Effect of Meltdown and Spectre Patches on the Performance of HPC Applications. (2018). https://arxiv.org/abs/1801.04329.

[24] The OpenBSD Project. 2014. OpenSSH. http://www.openssh.com http://www.openssh.com.

[25] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP '93)*. ACM, New York, NY, USA, 14. https://doi.org/10.1145/168619.168635