Shielding Software From Privileged Side-Channel Attacks

Xiaowan Dong, Zhuojia Shen, and John Criswell, *University of Rochester;*
Alan L. Cox, *Rice University;* Sandhya Dwarkadas, *University of Rochester*

https://www.usenix.org/conference/usenixsecurity18/presentation/dong

This paper is included in the Proceedings of the
27th USENIX Security Symposium.

August 15–17, 2018 • Baltimore, MD, USA

ISBN 978-1-931971-46-1

Shielding Software From Privileged Side-Channel Attacks

Xiaowan Dong
University of Rochester

Zhuojia Shen
University of Rochester

John Criswell
University of Rochester

Alan L. Cox
Rice University

Sandhya Dwarkadas
University of Rochester

Abstract

Commodity operating system (OS) kernels, such as Windows, Mac OS X, Linux, and FreeBSD, are susceptible to numerous security vulnerabilities. Their monolithic design gives successful attackers complete access to all application data and system resources. Shielding systems such as InkTag, Haven, and Virtual Ghost protect sensitive application data from compromised OS kernels. However, such systems are still vulnerable to side-channel attacks. Worse yet, compromised OS kernels can leverage their control over privileged hardware state to exacerbate existing side channels; recent work has shown that a compromised OS kernel can steal entire documents via side channels.

This paper presents defenses against page table and last-level cache (LLC) side-channel attacks launched by a compromised OS kernel. Our page table defenses restrict the OS kernel's ability to read and write page table pages and defend against page allocation attacks, and our LLC defenses utilize the Intel Cache Allocation Technology along with memory isolation primitives. We prototype our solution in a system we call Apparition, building on an optimized version of Virtual Ghost. Our evaluation shows that our side-channel defenses add 1% to 18% (with up to 86% for one application) overhead to the optimized Virtual Ghost (relative to the native kernel) on real-world applications.

1 Introduction

Bugs in commodity operating system (OS) kernels, such as Windows [60], Mac OS X [64], Linux [15], and FreeBSD [54], render them vulnerable to security attacks such as buffer overflows and information leaks. Furthermore, their monolithic architecture provides high performance but poor protection: a single vulnerability may give an attacker control over the entire OS kernel, allowing the attacker to steal and corrupt *any* data on the system. To reduce the size of the trusted computing base (TCB) on commodity systems, software solutions (such as InkTag [40] and Virtual Ghost [26]) and hardware solutions (such as Intel SGX [42], ARM TrustZone [11], and Haven [12]) prevent the OS kernel from reading and corrupting application data.

Despite these protections, attackers can steal application data using side-channel attacks that exploit shared hardware resources [38] or interactions between application code and the OS kernel [73]. Worse yet, a compromised OS kernel can exacerbate these side channels by manipulating software state, e.g., via CPU scheduling, and by configuring privileged hardware resources, e.g., the processor's interrupt timer and memory management unit (MMU) [38, 73]. Shielding systems must mitigate side-channel attacks if they are to protect the confidentiality of application data.

In this paper, we present methods to defend against page table and last-level cache (LLC) side-channel attacks launched by a compromised OS kernel. Our methods require no changes to existing processors. A malicious OS kernel may infer victims' memory access patterns and in turn recover secret information via tracing page table updates or page faults, or measuring the victims' cache usage patterns [43, 52, 63, 73]. To eliminate page table side channels, our key insight is that trusted software should prevent the OS kernel from reading or manipulating page table entries (PTEs) for memory holding application secrets. To thwart LLC side-channel attacks, we leverage Intel's Cache Allocation Technology (CAT) [4] in concert with techniques that prevent physical memory sharing.

Since our solution must prevent physical memory sharing, control configuration of the Intel CAT feature, and prevent reading and writing of page table pages, we implement our solution by enhancing Virtual Ghost. Virtual Ghost [26] already controls an OS kernel's access to page tables and to privileged hardware registers. It also provides private memory in which an application can store sensitive information and prevents sharing of physical memory containing application secrets. As Virtual Ghost is based on Secure Virtual Architecture (SVA) [28], we can combine our solution with other security policies enforced by SVA (such as memory safety [27,28]). Our solution does not change the Virtual Ghost paravirtualization interface and therefore requires no changes to existing SVA software and hardware.

We prototype our changes in a new version of Virtual Ghost dubbed *Apparition*. *Apparition* is optimized relative to the original Virtual Ghost by using Intel Memory Protection Extensions (MPX) [4] to reduce software fault isolation (SFI) overheads and by eliminating serializing instructions (which reduce instruction-level parallelism) added by the original Virtual Ghost to control page table access.

To summarize, our contributions are as follows:

- We show that using MPX for SFI and eliminating serializing instructions when accessing page table pages improves performance by up to 2× relative to the original Virtual Ghost.
- We design, implement, and evaluate a defense against page table side-channel attacks in Apparition that leverages Apparition's control over the page table pages.
- We show how Apparition's control over privileged hardware state can partition the LLC to defeat cache side-channel attacks. Our defense *combines* Intel's CAT feature [4] (which cannot securely partition the cache by itself) with existing memory protections from Virtual Ghost [26] to prevent applications from sharing cache lines with other applications or the OS kernel.
- We present a design that eliminates side-channel attacks that infer code memory accesses by controlling interrupt, trap, and system call dispatch, context switching, and native code generation.
- We evaluate the performance of Apparition, study the sources of its overheads, and compare it to the performance of Virtual Ghost enhanced with our new optimizations. Using native FreeBSD as the baseline, we find that Apparition adds 1% to 18% overhead to this version of Virtual Ghost on the real-world applications we tested except for one real-world program that experiences up to 86% additional overhead.

The rest of the paper is organized as follows. Section 2 describes our attack model. Section 3 provides background on memory management side channels along with potential/possible attacks. Section 4 provides background on Virtual Ghost and explains how we improved its performance. Section 5 describes the design of our mitigations against page table and cache-based side-channel attacks, and Section 6 discusses how our work mitigates some of the recent speculative execution side-channel attacks. Section 7 describes our prototype implementation. Section 8 presents the results of our experimental evaluation. Section 9 discusses related work, and Section 10 summarizes our contributions.

2 Attack Model

Our attack model assumes a strong attacker that controls the OS kernel and wishes to steal application data. Due to defenses like Virtual Ghost [26], this attacker cannot directly read application memory. We assume that the application and the libraries that it uses are part of the TCB for that application's security policy; that the application author has taken measures to ensure that the application and its libraries are safe from direct attack, e.g., by using security hardening tools [33, 56] or type-safe programming languages, and that the application and its libraries protect themselves from Iago attacks [17] by distrusting return values from the OS. We also assume that the attacker cannot gain physical access to the machine. Under such conditions, side-channel attacks become attractive.

We assume that the attacker will attempt to use side channels, either via a malicious user-space process or via malicious code within the OS kernel itself. We focus on page table side-channel [63, 73] and LLC side channel [13, 43, 52, 76, 79] attacks launched by software because of their practicality. These side channels may leak information on the program's accesses to data and/or code memory. Speculative execution side channels are outside our attack model's scope, but we discuss how our system can mitigate some of the Meltdown [49] and Spectre [46] side channels in Section 6. Side-channel attacks launched by hardware are outside the scope of our attack model.

3 Side-Channel Attacks

Side-channel attacks exploit implicit information flows within modern processors [36–38, 43, 52, 58, 63, 69, 73] to steal sensitive application data. The memory management side channels fall into two categories: ones resulting from shared architectural states and ones due to the OS's control of memory management.

Modern systems share architectural states across processes, including translation lookaside buffers (TLBs), translation caches, CPU caches, memory controllers, memory channels, DIMMs, and DRAM ranks and banks. The shared state allows one process to indirectly infer another process's behavior without direct access to the victim process's data. Observing which code or data a victim process accesses allows attackers to infer protected application data [37, 38, 58, 69].

A compromised OS can leverage its complete control over privileged processor state to create additional side channels. For example, the OS can steal a victim process's secret information by tracing page faults, page table updates, and cache activities [38, 73]. It can control system events to alleviate noise and use a side channel to steal an application's secret data with a single execution of the victim's code [38, 63, 73].

Systems that protect applications from the OS kernel like Virtual Ghost [26], Overshadow [20], InkTag [40], and Haven [12] do not mitigate these side channels; the architectural states are still shared among processes, and the OS kernel has access to or even controls the page table on these systems. In this section, we explain the page table [63, 73], LLC [43, 52], and instruction tracing [73] side-channel attacks that Apparition mitigates.

3.1 Page Table Side Channels

Commodity OS kernels can configure page tables, intercept and process page faults, and query the virtual address causing a page fault [15, 54, 60, 64]. With these abilities, a compromised OS can monitor which virtual addresses a victim process accesses and, with knowledge of the application's source code, infer its secret information [73]. Recent research [63, 73] shows that a compromised OS can use its ability to configure the page table to launch page fault side-channel attacks to acquire sensitive application data protected by Intel SGX [23, 42]. The attack is powerful enough to steal a document and outlines of JPEG images from a single execution of applications protected by InkTag [40] and Haven [12].

More specifically, the OS kernel can use the methods below to infer information about an application's memory access patterns via the virtual-to-physical address translation mechanism:

Swapping If the OS kernel cannot directly modify the PTEs for pages containing private application data, it can indirectly mark the pages inaccessible if the shielding system provides the OS with a mechanism to swap pages out and back in. The OS can use the mechanism to swap a page out and then infer the memory access patterns of applications by monitoring when the shielding system requests the OS to swap the page back in. Systems such as InkTag [40] and Virtual Ghost [26] provide mechanisms for swapping that prevent direct data theft via encryption but do not mitigate swapping side channels.

Reading PTEs If the OS kernel cannot modify PTEs and cannot swap out pages, it can still infer an application's memory access patterns by reading PTEs as the application executes. Many processors set a dirty bit in the PTE when they write to a page. Processors may also set an accessed bit when they read from or write to a page. By continually examining PTEs, the OS can learn when an application first reads from and writes to various memory locations [67]. On multi-processor and multi-core systems, the compromised OS can scan the page tables (which reside in memory) on one core while the application executes on another core.

Inferring Caching of Translations A compromised OS can potentially infer a victim's memory access patterns using PRIME+PROBE [8–10, 38, 58, 66, 78] and FLUSH+RELOAD [13, 76, 79] cache side-channel attacks on caches holding virtual-to-physical address translations. Processors cache virtual-to-physical address translations in TLBs [3, 4], on-chip translation caches [4, 14], and CPU caches in the memory hierarchy [2, 3]. If a compromised OS can use the same virtual-to-physical translation caches as the application or determine if a PTE is already cached in the processor's memory caches, it can infer information on whether the application has used that page.

We observe that successfully mitigating page table side channels requires protecting both the *confidentiality* and *integrity* of virtual-to-physical address translations.

3.2 Cache Side Channels

Cache side-channel attacks infer secret data by measuring the cache usage patterns of the victim [36–38, 43, 52, 58, 76, 79]. Two common cache side-channel attacks are PRIME+PROBE [58] and FLUSH+RELOAD [76], both of which can be applied on private caches [58] and shared LLC [43, 52].

The PRIME+PROBE attack [58] fills the monitored cache set with its own cache lines, busy-waits for a set time, and measures the time it takes to access its cache lines again. A longer access time indicates that the attacker's cache line has been evicted by a victim's access to data mapping to the same cache set. The FLUSH+RELOAD attack [76] is a variant of the PRIME+PROBE attack that relies on the victim and the attacker sharing pages containing target cache lines. Page sharing is common for shared libraries. The attacker first flushes the target cache line e.g., with the `clflush` instruction, busy-waits for a set time, and measures the time it takes to access the target cache line. A shorter access time indicates that the victim has already reloaded this target cache line.

LLC side-channel attacks can achieve a high attack resolution without requiring the attacker and the victim to share the same core [52]. Cache partitioning [35, 44, 50, 61, 70, 71, 80] can mitigate cache side channels by preventing the attacker from evicting the victim's cache lines. However, existing work assumes an unprivileged user-space attacker [70, 71, 80] or a virtual machine attacking its neighbors [35,44,50,61,80] and relies on privileged code to configure and manage the partitioning.

These defenses are ineffective against a compromised OS kernel. A compromised OS kernel can assign the same page color to the attacker and the victim or configure the hardware so that the attacker and the victim share the same cache partition. The OS kernel could even launch cache side-channel attacks itself. Therefore, our cache partitioning defenses must prevent malicious privileged code from manipulating cache partitions as well as from sharing partitions with protected applications.

3.3 Instruction Tracing Side Channels

We have so far presented side-channel attacks that attempt to infer data memory accesses. However, the instruction sequence executed by a program may also leak information about application secrets if there is a control dependence on data that the application wishes to keep secret i.e., an implicit flow [32]. A compromised OS could exploit side channels to trace instruction execution in a number of ways. If the shielding system neglects to hide an application's saved program counter when an interrupt, trap, or system call occurs, the OS could configure the processor timer to mimic single-step execution [38] and read the program counter as each instruction is executed. If that is not possible, the OS could use a page fault or cache side-channel attack on application code memory instead of (or in addition to) application data memory. Previous work has used page fault side channels [73] to infer when instructions are executed and, from that, to infer secret data from an application.

4 Virtual Ghost Improvements

Apparition extends Virtual Ghost. As Figure 1 shows, Virtual Ghost [26] is a compiler-based virtual machine, built from SVA [28], interposed between the software stack and the hardware. We present Virtual Ghost's design and then describe two performance improvements we made to Virtual Ghost that are present in Apparition.

4.1 Design

The OS kernel on a Virtual Ghost system is compiled to a virtual instruction set (V-ISA) [26]. The Virtual Ghost Virtual Machine translates virtual instructions to the native instruction set (N-ISA) for execution. Virtual Ghost can sign and cache native code translations to provide ahead-of-time compilation, or it can translate code at system install time, boot time, or just-in-time. Virtual Ghost forces all OS kernel code to be in V-ISA form. Application code can be in either V-ISA or N-ISA form.

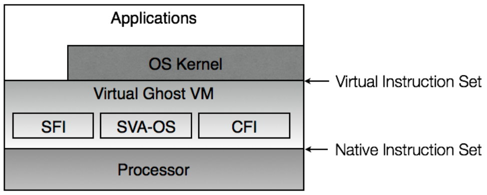

Figure 1: Virtual Ghost Architecture

The V-ISA consists of two sets of instructions [26]. The SVA-Core instructions are based on the LLVM Intermediate Representation (IR) [47], which uses static single assignment (SSA) form [30] to enable efficient static analysis of code. However, the original LLVM IR cannot support a complete OS kernel, so SVA provides a second set of instructions, SVA-OS [29], which allows the OS kernel to configure privileged hardware state, e.g., the MMU, and manipulate program state, e.g., context switching. The SVA V-ISA enables Virtual Ghost [26] to use compiler techniques to enforce security policies. Virtual Ghost can add run-time checks while translating code from the V-ISA to the N-ISA; the SVA-OS instructions can help enforce security policies by restricting hardware configuration and state manipulation.

Via compiler instrumentation and run-time checks, Virtual Ghost can provide applications with the functionality they need to protect themselves from a compromised OS kernel [26]. One such feature is *ghost memory*. For each process, Virtual Ghost divides the virtual address space into four regions as Figure 2 depicts. There is user-space memory that an application and the OS kernel can use to communicate; both can read and modify it. There is also kernel memory, which the OS kernel can read and write. Unlike existing systems, Virtual Ghost prevents user-space memory and kernel memory from being executable; they do not contain executable native code. Virtual Ghost adds a new ghost memory region that only the application can read and modify and can therefore use to hold sensitive data. Finally, there is the Virtual Ghost VM memory region in which Virtual Ghost stores its own data structures, the native code translations it creates for V-ISA code, and the code segments of N-ISA application code. Pages containing native code are mapped as execute-only while all other Virtual Ghost VM memory regions are inaccessible to applications and the kernel.

With these features, programmers can write *ghosting applications* for Virtual Ghost systems that actively protect themselves from the OS kernel: applications can

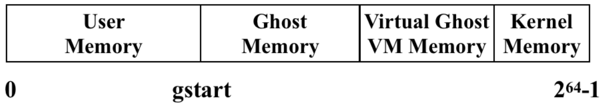

Figure 2: Virtual Ghost Address Space Layout

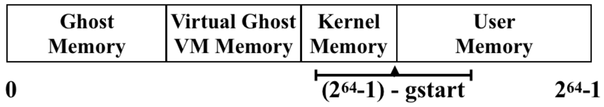

Figure 3: Address Space Layout Seen by Intel MPX

store all their data and encryption keys inside ghost memory to prevent theft and tampering, and they can use encryption and digital signatures to maintain data confidentiality and integrity when sending data into or receiving data from the operating system's I/O systems [26]. Since Virtual Ghost generates all the native code that is executed on the system [26], it can place that code into the Virtual Ghost VM memory and protect its integrity from both the OS kernel and errant applications.

Virtual Ghost employs SFI [68] to protect the confidentiality and integrity of ghost memory and Virtual Ghost VM memory [26]. It adds a set of bit-masking and predicated instructions before every load and store within the OS code to ensure that every pointer used in a load or store operation points into either user- or kernel-space memory. Additionally, by placing interrupted program state in the Virtual Ghost VM memory during interrupt, trap, and system call dispatch, Virtual Ghost can protect saved processor state using SFI. However, as Virtual Ghost allows the OS kernel to read page tables, it does not place them in Virtual Ghost VM memory. Instead, it maps page table pages as read-only memory by the OS and makes the OS use SVA-OS instructions to modify them, thereby preserving the integrity of the page table pages. Finally, Virtual Ghost employs control flow integrity (CFI) [7] to ensure that the SFI instrumentation is not bypassed.

We have enhanced the performance of Virtual Ghost with two new optimizations, which we include in Apparition. First, our prototype uses the Intel MPX bounds checking instructions [4] to implement faster SFI. Second, we refactored how Virtual Ghost protects page table pages to reduce the number of serializing instructions.

4.2 Intel Memory Protection Extensions

Intel's MPX [4] was originally designed to accelerate memory safety enforcement via hardware support. MPX enhances the processor with four bounds registers, each of which maintains the lower and upper bounds of a single memory object. Bounds checking instructions check a virtual address against either the lower or upper bound of the specified bounds register and generate a trap if the virtual address does not reside within the bounds.

Virtual Ghost uses SFI to ensure that the kernel does not access ghost memory and VM memory regions while allowing access to user- and kernel-memory regions. To implement SFI using MPX, we treat the combined user- and kernel-space regions as a single large memory object; the Virtual Ghost VM can then replace SFI's bit-masking and predicated instructions before every load and store within the kernel with MPX bounds checking instructions.

One challenge with efficiently using MPX is that the user- and kernel-memory regions are not contiguous. Furthermore, since their current placement enables the compiler to use more efficient addressing modes on x86-64, moving them to make them contiguous could negatively impact performance.

To address this issue, each run-time check before a load or store first subtracts the length of user-space memory (denoted *gstart*) from the address that is to be checked. This makes the user- and kernel-space regions appear contiguous (as Figure 3 shows). MPX bounds checks can then be used by setting the base and bound registers to the remapped values of the start of kernel-space and the end of user-space memory. If the access is outside of kernel and user space, the processor generates a trap into the Virtual Ghost VM which handles the out-of-bounds error.

4.3 SVA Internal Direct Map

A direct map is a range of virtual pages that are mapped to consecutive physical addresses, i.e., the first page to the first physical frame of memory, the second page to the second physical frame, and so forth. With a strategically placed direct map, an OS kernel can quickly find a virtual address mapped to a specific physical address by applying a simple bitwise OR operation to the physical address [15]. Operating systems such as Linux and FreeBSD use the direct map to write to page table pages. Since Virtual Ghost must control how the processor's MMU is configured [26], it originally mapped page table pages in the OS kernel's direct map for read-only access, and when an SVA-OS instruction needed to update the page tables, it temporarily cleared the x86 CR0.WP bit to disable the MMU's enforcement of write protection, thereby allowing the Virtual Ghost VM to modify the page table.

We have found that this method incurs significant overhead as flipping CR0.WP is a serializing operation that interferes with instruction-level parallelism [4]. This caused Virtual Ghost's page table updates to be much slower than those of a conventional OS kernel, decreasing the speed of process creation and termination, demand paging, and the execution of new programs.

Apparition eliminates the need for modifying CR0.WP by placing a direct map of physical memory within the Virtual Ghost VM memory that provides write access to all physical frames, including page table pages. When Virtual Ghost needs to update a PTE, it simply modifies the entry via its internal direct map instead of flipping CR0.WP to toggle the write protection on the OS kernel's direct map. Since this internal direct map is within Virtual Ghost VM memory, the existing SFI mechanism prevents the OS kernel from altering it.

5 Side-Channel Mitigations

We now present our design for mitigating page table, LLC, and instruction tracing side-channel attacks.

5.1 Page Table Side Channels

To mitigate the page table side-channel attacks described in Section 3.1, a system must protect both the confidentiality and integrity of the page table pages. Apparition must therefore enforce several restrictions.

Page Table Restrictions Apparition must prevent the OS from modifying PTEs that map ghost memory. Otherwise, the OS can unmap ghost memory to track the program's memory accesses via page faults. Likewise, Apparition must ensure that page frames used for ghost memory are not mapped into virtual memory regions that the OS can access; Virtual Ghost already enforces these constraints [26].

Apparition must additionally prevent the OS from reading PTEs (and therefore the corresponding page table pages) that map ghost memory. This prevents the OS from observing updates to PTEs caused by ghost memory allocation, deallocation, and swapping and from inferring information when the processor sets the accessed or dirty bits in PTEs for ghost memory.

To enforce these restrictions, we exploit the hierarchical, tree-like structure of x86 page tables. Virtual Ghost allows the OS kernel to directly read all PTEs but forces the kernel to modify PTEs with the `sva_update_mapping()` SVA-OS instruction [26]. This ensures that the OS does not gain access to ghost memory by altering the page table. Apparition disables *all* OS accesses to the subtree of the page table that maps ghost memory by removing read/write permission to the page table pages in this subtree from the OS's direct map; only the Apparition MMU instructions can read and write PTEs mapping ghost memory via the new SVA internal direct map described in Section 4.3. This ensures the integrity and confidentiality of ghost memory.

Swapping Apparition's ghost memory swapping instructions must prevent the OS from selecting which ghost memory pages to swap out and in. Instead, the secure swap-out instruction should randomly select a page to encrypt and swap out. The secure swap-in instruction should swap in *all* the pages that have been swapped out for that process (as opposed to swapping in a single page). This prevents the OS from learning which pages the process accesses. However, it also restricts the size of any single application's ghost memory to a fraction of physical memory; otherwise, it may be impossible to swap in all swapped-out ghost pages, causing the process to fail to make forward progress. Since the OS retains control over user-space memory, it should swap that memory out first before swapping out ghost memory; swapping out user-space memory imposes no restrictions on the OS.

5.2 Page Allocation Side Channels

By protecting the confidentiality and integrity of page table pages, our Apparition design protects applications from side channels that flow through the page table pages. However, in addition to these protections, our Apparition design must ensure that the application does not leak information through its ghost memory allocation behavior. Otherwise, a compromised OS can use this new side channel in lieu of existing page table side channels.

Virtual Ghost [26] requires the OS to provide a callback function that the Virtual Ghost VM can use to request physical frames from the OS kernel. This design decouples resource management from protection: the OS decides how much physical memory each process uses while Virtual Ghost protects the integrity and confidentiality of the memory. However, Virtual Ghost imposes no restrictions on when the Virtual Ghost VM requests physical memory from the OS. As a result, a compromised OS kernel can use the physical memory callback like a paging side channel. For example, if the Virtual Ghost VM lazily maps physical memory to ghost virtual addresses on demand and requests a single memory frame from the OS when it needs to map a ghost page, then the OS can infer the application's paging behavior.

To mitigate this side channel, in Apparition we disable demand paging on ghost memory. By doing so, we convert this side channel into a memory allocation side channel from which the OS can only infer memory allocation size; this leaks much less information about an application's secret data. To the best of our knowledge, no existing work exploits such memory allocation side

Name	Description
`void allocmem(int num, uintptr_t frames[])`	Allocate `num` physical memory frames and store the addresses to them in the specified array.
`void freemem(int num, uintptr_t frames[])`	Free `num` physical memory frames whose addresses are stored within the specified array.

Table 1: Physical Memory Allocation Callbacks

channels. To obfuscate the memory allocation size information, we redesign the physical memory allocation callback and impose new restrictions on how Apparition uses it. Table 1 shows the new design. The Apparition VM calls `allocmem()` to request a specified number of frames and `freemem()` to free frames. In our design, the Apparition VM will request a random number of frames from the OS when it needs more physical memory; these frames will be stored within an internal cache of free frames that it can use to fulfill ghost memory requests. When the internal cache of free frames becomes sufficiently large, the Apparition VM will return frames to the OS so that they can be used for other purposes. This design obscures ghost memory allocation patterns from the OS while still giving the OS some control over how much physical memory is used for ghost memory across all processes running on the system. We can create Aparition VM APIs for applications to disable these two protections if the application is not concerned about page allocation side-channel attacks.

5.3 Code Translation Side Channels

As Section 3.3 explains, attackers can use side channels on code memory accesses in addition to data memory accesses. Since Virtual Ghost places native code translations and N-ISA application code into Virtual Ghost VM memory [26], Apparition's page table (Section 5.1) and page allocation (Section 5.2) defenses eliminate code memory side channels. However, for V-ISA applications, Apparition must translate V-ISA code to N-ISA code without creating new side channels. When the OS loads an application in memory for execution, it loads the V-ISA code into either user-space or kernel-space memory and then asks Apparition to verify the integrity of the code and to create the native code for the application in Virtual Ghost VM memory. Apparition must ensure that its accesses to the V-ISA code do not leak information about the application's execution.

Two simple methods can eliminate this side channel. If the Apparition implementation does not employ run-time optimizations (such as lazy code translation), it must simply ensure that it translates all the V-ISA code of an application to native code when the OS requests translation via the `sva_translate()` SVA-OS instruction; so long as it does not read V-ISA code on demand as the program executes e.g., for lazy compilation, then no side channel exists.

If the Apparition VM performs run-time optimizations such as lazy code translation, it must copy the entire V-ISA code into Apparition VM memory first and use that copy to perform these run-time optimizations. In this way, both the V-ISA code and N-ISA code are protected from side channels.

5.4 LLC Side Channels

Our LLC side-channel defenses must prevent an application from sharing ghost memory with a compromised OS and other applications and ensure that cache lines for physical memory mapped to ghost memory will not be read or evicted by the OS or other applications.

Preventing Page Sharing Virtual Ghost [26] already ensures that an application's ghost memory cannot be accessed by the OS or other applications. As Sections 4.1 and 5.1 describe, the SFI instrumentation prevents the OS kernel from accessing ghost memory and from mapping ghost memory into regions that the OS kernel can access. Likewise, Virtual Ghost ensures that applications have their own private ghost memory that is not shared with other applications. This not only prevents data theft by applications and compromised OS kernels, but, as we discuss next, allows our Apparition design to utilize Intel CAT [4] to defend against LLC side-channel attacks.

Cache Partitioning Our defense against LLC side-channel attacks combines Virtual Ghost's existing memory protection mechanisms [26] with static cache partitioning implemented using Intel's CAT processor feature [4]. Intel CAT enables way-partitioning of the LLC into several subsets of smaller associativities [4]. A processor can switch among multiple classes of service (COS, or resource control tag with associated resource capacity bitmap indicating the subset of LLC ways assigned to the COS) at runtime. Privileged code can switch the COS and configure the bitmaps of each COS by writing to model-specific registers. The number of COSs supported depends on the processor type. In addition, Intel imposes two constraints [50]: the bitmap must contain at least 2 ways, and the ways allocated must be contiguous. Once CAT is configured, the processor can only load cache lines into its subset of the cache; code running in one COS cannot evict cache lines in another COS. However, software in one COS can read data from all cache lines in the LLC, allowing software running in different COSs to read the same cache lines if they are sharing physical memory e.g., read-only mapped shared library code.

Our design requires one partition for kernel code and non-ghosting applications not using ghost memory, one for Apparition VM code, and one for each ghosting application. The processor in our experiments (Section 8) has four partitions. If there are more ghosting applications executing than partitions available, then the Apparition VM will need to multiplex one or more partitions between ghosting applications and flush the cache on context switches. Partitioning ghosting applications from both the kernel and non-ghosting applications eliminates side channels between these two domains, preventing the kernel from inferring information by measuring cache access time. Partitioning also eliminates costly cache flushes when control flow moves between ghosting application, Apparition VM, and OS kernel/untrusted application code. Additionally, partitioning the Apparition VM from the kernel and from ghosting applications ensures that any secrets held within Apparition VM memory (such as page tables) do not leak to either applications or the OS kernel.

Unfortunately, Intel CAT allows data reads from cache lines outside of the current COS [4]. However, since Apparition ensures that there is no sharing of ghost memory or native code between a ghosting application and the OS kernel (or other applications), and since the MPX SFI protections prevent the OS kernel from accessing ghost memory and Apparition VM memory, *such cross-COS reads will never occur*. Hence, the memory protections in Virtual Ghost *coupled* with Intel CAT can defend against LLC side-channel attacks.

Cache Partitioning Configuration The Apparition VM configures the cache partitions on boot and uses several mechanisms which, together, ensure that the OS kernel cannot reconfigure or disable the cache partitioning. First, the SVA virtual instruction set has no instructions for changing the cache partitions. Second, Virtual Ghost's MMU protections prevent the OS kernel from loading new native code into memory that was not translated and instrumented by the Virtual Ghost VM [26]. Third, Virtual Ghost enforces CFI on kernel code, ensuring that the OS kernel can only execute its own code and cannot jump into the middle of variable-length x86 instructions within the kernel [26] that might reconfigure cache partitioning.

On an interrupt, trap, or system call, the processor transfers control to the Apparition VM which switches the cache partition in use to the Apparition VM's partition. After saving the interrupted processor state in Apparition VM memory, the Apparition VM switches to the kernel's cache partition before calling the kernel's interrupt, trap or system call handler. Likewise, SVA-OS instructions switch to the Apparition VM's partition on entry and back to the kernel's partition on exit.

Our design also protects distrusting applications from each other by giving each application needing protection from LLC side channels its own cache partition. Initially, the Apparition VM assigns one cache partition to the first application using ghost memory. This cache partition will be divided into more cache partitions when more applications needing protection are scheduled. Apparition can either divide the cache space evenly between applications or employ quality-of-service policies based on the applications' LLC working sets. The only restriction is that each application's partition must have at least two ways. On current Intel processors, the Apparition VM must flush the entire cache when dividing a cache partition. Similarly, the Apparition VM will need to flush the cache on context switches if the number of distrusting ghosting applications exceeds the number of COSs provided by the processor.

If a process wants to create a cooperating thread with which to share its ghost memory or a child process which it trusts to use the same cache partition, the process can provide an option to the `fork()` system call indicating that the new process or thread should use the same cache partition as the parent process. Virtual Ghost (and hence Apparition) dispatches all system calls and creates all new processes and threads [26]. It can therefore determine whether the new process or thread that it creates should use the same cache partition as its parent.

5.5 Instruction Tracing Side Channels

As Section 3.3 discusses, inferring the dynamic order in which a program executes its instructions can leak information about data if the program counter depends upon secret data [32]. Existing attacks exploit such implicit flows within programs by tracing code memory page faults [73] or via timer-based interrupts [38].

Virtual Ghost [26] saves interrupted program state within the Virtual Ghost VM memory, forcing the OS kernel to use SVA-OS instructions to read or modify interrupted program state. The SVA-OS instruction set does not provide an instruction for retrieving the program counter stored within interrupted program state [25, 26]. As a result, while a compromised OS can interrupt an application as frequently as it wants, it cannot infer the program counter from interrupted program state. Combined with the virtual instruction set code and native code memory mitigations described in Section 5.3, Apparition mitigates attacks that infer a ghosting application's program counter.

6 Impact on Speculation Side Channels

Recently, there has been much press about two classes of attacks, Meltdown [49] and Spectre [46], in which user-space code leverages speculative execution side channels in the processor to steal data and then exfiltrates the stolen data via existing side channels. While speculation side channels are outside the scope of our attack model in Section 2, our defenses mitigate some variants of these attacks that use cache side channels.

Spectre [46] is an attack in which one user-space process attempts to infer information about another user-space process. It utilizes the existence of shared branch prediction tables and branch target buffers to force the victim to speculatively execute code that loads sensitive data into the cache. Since our defenses partition the LLC and prevent the sharing of ghost memory, values in ghost memory will not become visible to attackers in the LLC. However, in order to mitigate speculation side-channel attacks, Apparition will need to prevent the sharing of all physical memory between untrusted processes, including native code pages and traditional user-space memory. Failure to do so would allow a Spectre attack to communicate information across the Intel CAT partitions through shared physical memory.

With several enhancements, Apparition could mitigate other forms of these attacks. To mitigate Meltdown [49] and Spectre [46] attacks that speculatively access out-of-bounds memory, Apparition could use speculation-resistant SFI instrumentation on both application and kernel code [34] to protect large memory regions; in particular, we show in [34] that SFI instrumentation using instruction sequences to stall speculative execution using a data dependence so that the SFI instructions must complete before the protected memory read instruction begins execution. To provide finer granularity protection, e.g., at the granularity of individual memory objects, Apparition could place `lfence` instructions before memory read instructions that have a control dependence on a branch to ensure that all instructions performing array bounds checks have committed before the load commences execution [6].

To mitigate Meltdown attacks [49], Apparition could transparently use a different set of page tables and PCIDs for user-space code, OS kernel code, and Apparition VM code [34], building off the suggestions from Intel [6].

Since Apparition uses a virtual instruction set to abstract away hardware details and controls native code generation, it can employ any or all of these mitigations *without changing application or OS kernel source code*.

Component	Source Lines of Code
SVA-OS	5,823
SFI Pass	292
CFI Pass	726
Total	**6,841**

Table 2: Apparition Physical Source Lines of Code

The virtual instruction set remains unchanged; Apparition can employ these solutions by enhancing its compiler transformations and native code generation.

7 Implementation

We implemented Apparition by modifying the Virtual Ghost prototype for 64-bit x86 systems [26]. Apparition uses the FreeBSD 9.0 kernel ported to the SVA-OS virtual instruction set and is compiled with the LLVM 3.1 compiler. The Apparition prototype only supports single-processor execution, so our evaluation focuses on single-core overheads.

We used `sloccount` [72] to measure the source lines of code (which excludes whitespace and comments) of the SVA-OS instructions, the SFI compiler pass, and the CFI compiler pass comprising Apparition; Table 2 shows the results. Apparition's TCB contains 6,841 source lines of code which includes all of Virtual Ghost's old functionality [26], Apparition's functionality, and configuration options to enable and disable the new Apparition features. The original Virtual Ghost prototype contained 5,344 source lines of code [26] in comparison.

We implemented the MPX SFI optimization in Apparition by changing the existing LLVM IR-level SFI pass in Virtual Ghost [26] to insert inline assembly code utilizing MPX instructions instead of LLVM IR bit-masking instructions. We also implemented the SVA direct map by enhancing the SVA-OS instructions within Apparition. While Virtual Ghost is designed to restrict Direct Memory Access (DMA) operations to memory with an I/O MMU [26], neither the original Virtual Ghost prototype nor our prototype implements this feature.

To implement our paging protections in Sections 5.1 and 5.2, we modified the ghost memory allocator within the Apparition VM so that it requests all physical memory frames from the OS when the application uses the hypercall to request ghost memory. The previous implementation [26] would delay allocation of physical memory until the application read or wrote the ghost memory; the Virtual Ghost VM would then request a frame from the OS and map it on demand. Our ghost memory allocator also implements randomization; it maintains a set of memory frames within the Apparition VM and requests a random number of frames from the OS kernel when this reserve becomes empty. Additionally, the FreeBSD 9.0 malloc() implementation always requests ghost memory in constant-sized chunks from the Apparition VM, further obscuring the application's actual memory allocation information from the OS kernel. As neither the Virtual Ghost prototype [26] nor our new prototype implement virtual-to-native code translation, we did not implement the mitigations in Section 5.3. Additionally, neither prototype supports swapping out of ghost memory to persistent storage.

Our prototype also implements the LLC side-channel mitigation features in Section 5.4. As our test machines support 4 cache partitions, we reserved one for the Apparition VM (dubbed VM COS), one for the OS kernel and non-ghosting applications (dubbed kernel COS), and one for a ghosting application (dubbed ghosting COS). We modified all of the SVA-OS instructions to switch between the kernel COS and the VM COS upon entry and exit. Our prototype switches between the ghosting COS and the kernel COS on context switches between ghosting and non-ghosting applications. It also multiplexes the ghosting COS by flushing the cache on context switches between two ghosting applications.

8 Evaluation

We first evaluate the performance optimizations described in Section 4. We then evaluate the performance overheads of our page table and LLC side-channel defenses.

8.1 Methodology

For our experiments, we used a Dell Precision T3620 workstation with an Intel® Core^{TM} i7-6700 hyper-threading quad-core processor at 3.40 GHz with an 8 MB 16-way LLC, 16 GB of RAM, and an Intel E1000 network card. The machine has both a 256 GB Solid State Drive (SSD) and a 7,200 RPM 500 GB hard disk. We stored all the files for our experiments on the SSD. For the network experiments, we used a dedicated Gigabit Ethernet network and a Dell T1700 Precision workstation as the remote system. The T1700 runs FreeBSD 9.3 and has an Intel® Core^{TM} i7-4770 hyper-threading quad-core processor at 3.40 GHz and 16 GB of RAM. We perform our experiments with the OS running in single-user mode to reduce noise from other processes on the system. We use a high-resolution timer (reading rdtsc directly) to measure time, and we report the average (arithmetic mean of) execution time of multiple runs.

Our evaluation needed benchmarks and applications that rely heavily on OS kernel services e.g., the file system and network stack. Our evaluation therefore used the following programs:

LMBench: We used the LMBench benchmark suite [55] to measure the latency of various system calls on Virtual Ghost with and without the new optimizations. For the benchmarks for which we can specify the number of repetitions to run, we used 1,000 repetitions. LMBench reports the median result of the number of repetitions specified. We configured lat_select to use local files. In lat_ctx, we measured context switch time between two processes; each process does nothing but passes a token to the other process via a pipe. For all the other workloads, we used the default configurations.

OpenSSH Client: We used the preinstalled OpenSSH [65] Secure Shell client and server to evaluate the Virtual Ghost optimizations. We ran the OpenSSH client on our FreeBSD 9.0 machine and the server on the FreeBSD 9.3 machine to measure bandwidth. We generated the contents of each file by collecting random numbers from the /dev/random device on our FreeBSD 9.0 machine and transferred the files to the FreeBSD 9.3 machine.

Ghosting OpenSSH Client: We evaluated our defenses on the ssh and ssh-keygen programs of the OpenSSH 6.2p1 application suite modified by Criswell et al. to use ghost memory to store heap objects [26]: ssh-keygen generates public and private key pairs for ssh to use for password-less authentication. Criswell et al. enhanced these two programs to share a hard-coded AES private application key that they use to encrypt private authentication keys. The ssh-keygen program encrypts all the private authentication key files it generates with this private application key. The ssh client decrypts these keys and puts them, as well as all other heap objects, into ghost memory. For these experiments, we ran the ghosting OpenSSH client on the Virtual Ghost and Apparition machine and the server on the machine running native FreeBSD 9.3. We collected the bandwidth reported in the ssh client's debug output when transferring 1 KB to 512 MB files using the modified ssh client. We transferred the files by having the ssh client run the cat command on the files on the server.

Ghosting Bzip2: We compiled Bzip2 1.0.6, a data compression program [16], with a new C library that can, at run-time, be configured to allocate heap objects in either traditional user-space memory or in ghost memory. We measure the time for Bzip2 to compress the 32 MB file we used in the OpenSSH experiments.

Ghosting GnuPG: We compiled GnuPG 2.0.18, a cryptography program [45], with our C library that can, at run-time, be configured to allocate heap objects in either traditional user-space memory or in ghost memory. We evaluate encrypting, decrypting, signing, and verifying signatures of files ranging from 1 KB to 32 MB in size. Due to space, we only report overheads for signing files. Encryption, decryption, and verification have

Test	Native (μs)	Std. Dev.	VG Overhead	Opt-VG Overhead
null syscall	0.1	0.0	2.9×	2.6×
open/close	1.8	0.0	2.3×	1.8×
mmap	5.6	0.1	5.1×	3.4×
page fault	36.3	1.3	1.0×	1.0×
fork + exit	49.2	0.1	4.1×	2.0×
fork + exec	54.4	0.1	3.9×	1.9×
fork + /bin/sh -c	515.4	1.0	2.2×	1.5×
signal handler install	0.2	0.0	2.3×	2.1×
signal handler delivery	1.1	0.0	0.9×	0.8×
read	0.1	0.0	2.7×	2.3×
write	0.1	0.0	2.9×	2.5×
stat	1.2	0.0	2.1×	1.8×
select	2.8	0.0	1.9×	1.6×
fcntl lock	2.8	0.0	1.9×	1.6×
context switch	0.5	0.0	1.2×	1.0×
pipe	1.6	0.0	1.7×	1.5×

Table 3: LMBench Latency Results

similar overheads.

Ghosting RandomAccess: We created a microbenchmark named RandomAccess which modifies an 8 MB array of 64 B elements in the heap in random order 20,000 times. Specifically, it first generates a random order in which to access all the array elements, ensuring that every element in the array is accessed once. It then iterates over the array in the random order, replacing the contents of the current element with the index of the previously accessed element. The first iteration warms up the cache and is not used in measuring performance; RandomAccess records the execution time of the next 20,000 iterations and reports the average latency of an iteration. By seeding the pseudo-random number generator with the same seed, RandomAccess can exhibit deterministic results. We link RandomAccess with our C library so that we can configure it to allocate heap objects in traditional user-space memory or in ghost memory as needed.

Ghosting Clang: We compiled Clang 3.0, a C/C++ compiler [1], with our C library that can, at run-time, be configured to allocate heap objects in either traditional user-space memory or in ghost memory. We measured the time to compile a C source file named `gcc-smaller.c` from SPEC CPU 2017 [5] into assembly code by using Clang. We used the `-O3` and `-pipe` command-line options.

Besides the native FreeBSD 9.0 kernel, we have conducted our experiments on the FreeBSD SVA kernels with the following configurations of Virtual Ghost/Apparition:

1. **VG**: Virtual Ghost without the new optimizations described in Section 4 and without our new defenses. This version of Virtual Ghost is a faster and more robust implementation of the original prototype [26].

2. **Opt-VG**: Virtual Ghost with the optimizations described in Section 4.

Test	Native (MB/s)	Std. Dev.	VG Overhead	Opt-VG Overhead
pipe	14,865.2	29.7	1.3×	1.2×

Table 4: LMBench Bandwidth Results

3. **Opt-VG-PG**: The optimized Virtual Ghost enhanced with only our defenses to the page table side-channel attacks.

4. **Opt-VG-LLCPart**: The optimized Virtual Ghost enhanced with only our mitigations to the LLC side-channel attacks.

5. **Apparition**: The optimized Virtual Ghost enhanced with the defenses to both the page table and LLC side-channel attacks (in other words, the full Apparition system).

8.2 Virtual Ghost Optimizations

We evaluate the overheads of the optimized version of Virtual Ghost's SFI enforcement and SVA-OS MMU instructions (described in Section 4) relative to the original Virtual Ghost and to native x86-64 FreeBSD. For the baseline kernel, we used a native x86-64 FreeBSD 9.0 kernel configured with the same options as the Virtual Ghost FreeBSD kernels and compiled with the same compiler and compilation options. We focus here on evaluating the overheads of Virtual Ghost on traditional non-ghosting applications, i.e., applications that do not use ghost memory but still need to run on the Virtual Ghost system. Our microbenchmarks and benchmark applications therefore do not use ghost memory when running on Virtual Ghost.

As shown below, our optimizations always improve performance for the benchmarks we tested.

Microbenchmarks: We used the LMBench benchmark suite [55] to measure the latency of various system calls on Virtual Ghost with and without the new optimizations. Tables 3 and 4 show the performance of the

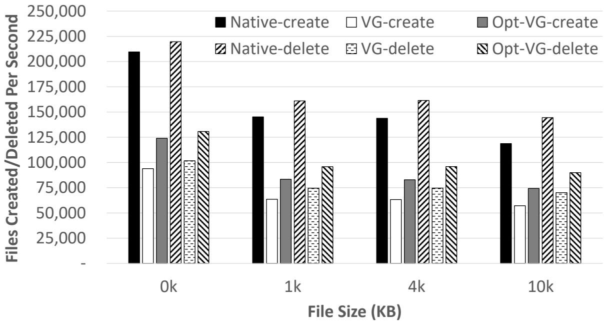

Figure 4: LMBench File Creation/Deletion Rate

native FreeBSD 9.0 kernel and the overheads of Virtual Ghost, with and without the optimizations, normalized to the native FreeBSD 9.0 kernel. While the overheads in Table 3 may seem high, we note that the performance of real-world applications (shown subsequently) are much better as applications only spend a portion of their time executing kernel code.

As Tables 3 and 4 show, Virtual Ghost incurs 2.4× overhead on average while our optimizations reduce the overhead to 1.8× on average. In particular, elimination of serializing instructions improves system calls that perform many page table updates. For example, `fork + exit` overhead drops from 4.1× to 2.0×, and `fork + exec` drops from 3.9× to 1.9×. On FreeBSD, the `mmap()` system call premaps some amount of physical memory to the newly mapped region, so our optimizations also improve its overhead from 5.1× to 3.4×.

Signal handler function dispatch shows a slight performance improvement on Virtual Ghost compared to native FreeBSD. The FreeBSD kernel on Virtual Ghost cannot read the register state saved on interrupts, traps, and system calls [26] and therefore does not copy this information into the user-space stack for signal handlers to inspect like the FreeBSD kernel does. We believe this is why Virtual Ghost shows a slight performance benefit for signal handler dispatch.

Figure 4 reports the performance of the file creation/deletion workload of LMBench on native FreeBSD and Virtual Ghost with and without the new optimizations. Virtual Ghost slows down the file creation and deletion rates by 2.2× and 2.1×, respectively, on average across all file sizes, and the optimizations reduce both of the overheads to 1.7×. The standard deviation is 0% for all file sizes tested.

Applications: Table 5 lists the average CPU time spent for OpenSSH client file transfers on the native FreeBSD kernel over 20 rounds of execution. We measured the CPU time by recording the number of unhalted clock cycles used while executing the `ssh` client with the `pmcstat` utility and then converted this number into milliseconds based on the CPU's clock speed. We made the same measurements for the OpenSSH client on Virtual Ghost with and without optimizations; the VG and Opt-VG lines in Figure 6 show the results. For files from 1 KB to 8 MB, the original Virtual Ghost incurs overheads of 3% to 12% with a 1% average standard deviation. The optimizations reduce the overhead to 2% to 10%. For files larger than 8 MB, the overheads of Virtual Ghost with or without the optimizations are negligible. Additionally, the differences between the results of 128 KB, 256 KB and 512 KB are within the standard deviation.

Size	CPU Time	Std. Dev.	Size	CPU Time	Std. Dev.
1	13.7	0.3	1,024	26.9	0.4
2	13.8	0.2	2,048	37.1	0.4
4	13.9	0.2	4,096	57.3	0.3
8	14.5	0.3	8,192	97.8	0.4
16	15.2	0.3	16,384	178.4	0.4
32	16.8	0.3	32,768	339.9	0.5
64	17.1	0.4	65,536	662.2	0.3
128	18.1	0.3	131,072	1,306.8	0.6
256	19.0	0.5	262,144	2,596.0	1.2
512	21.5	0.4	524,288	5,171.1	2.5

Table 5: OpenSSH Client Average File Transfer CPU Time. Time in miliseconds. Size in KB.

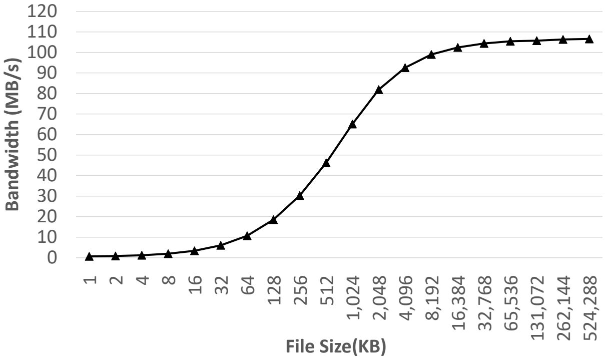

Figure 5: OpenSSH Client Average File Transfer Rate on Native FreeBSD

Figure 5 shows the average OpenSSH client file transfer bandwidth on the native FreeBSD kernel over 10 rounds. For files between 1 KB and 2 MB in size, the original Virtual Ghost incurs negligible overheads ranging from 1% to 3% with up to 1% standard deviations. With the optimizations, the overheads on bandwidth remain similar.

Table 6 shows the overhead of Virtual Ghost with and without the new optimizations on Bzip2 compression and GnuPG when signing 2 MB files. For this experiment, ghost memory is disabled, so heap objects are allocated in traditional user-space memory, and physical memory is mapped on demand. We use a small file size here as

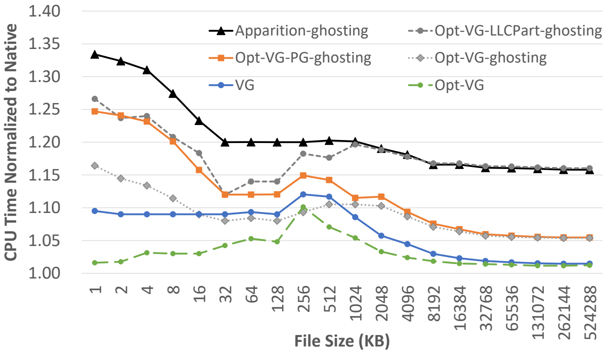

Figure 6: OpenSSH Client Average File Transfer CPU Time Normalized to Native FreeBSD

	Bzip2	GnuPG Signing
Native (ms)	183.20	54.71
VG Overhead (×)	1.05	1.06
Opt-VG Overhead (×)	1.04	1.03

Table 6: Bzip2 and GnuPG Results for 2 MB Files

	RandomAccess	Bzip2	Clang
Native FreeBSD	643.23 μs	2.89 s	28.36 s
std. dev.	0.64 μs	0.00 s	0.63 s
Opt-VG Overhead (×)	1.28	1.04	1.03
Opt-VG-PG Overhead (×)	1.32	1.04	1.03
Opt-VG-LLCPart Overhead (×)	2.09	1.04	1.03
Apparition Overhead (×)	2.11	1.05	1.05

Table 7: RandomAccess, Bzip2 and Clang Results

Virtual Ghost has higher overhead on GnuPG when compressing 2 MB files than when compressing larger files. Virtual Ghost adds 5% overhead to Bzip2, which is reduced to 4% with the optimizations. It incurs a 6% overhead to the overall performance for GnuPG signing; the optimizations reduce the overhead to 3%. The standard deviations for both Bzip2 and GnuPG is 0%.

8.3 Page Table Side-Channel Defenses

We now evaluate the performance of our page table side-channel defenses in Sections 5.1 and 5.2.

Ghosting RandomAccess: The second column of Table 7 reports the average latency of each iteration over 20 rounds of execution for the RandomAccess microbenchmark. The overheads on Virtual Ghost with our new optimizations without (Opt-VG) and with our page table side-channel defenses enabled (Opt-VG-PG) show that the page table side-channel defenses add no additional overhead to Opt-VG (when accounting for the standard deviation of 4%). This is because the only OS kernel operations incurred during the loop in RandomAccess are context switches, and our page table defenses add no overhead to context switching. We believe that Opt-VG and Opt-VG-PG add overhead to native FreeBSD because Opt-VG and Opt-VG-PG map ghost memory with 4 KB pages while native FreeBSD maps traditional user-space memory using super pages whenever possible [57].

Ghosting Bzip2: We enabled ghost memory for Bzip2 for all systems except the native FreeBSD kernel. The third column of Table 7 reports the average of 10 rounds of this experiment and shows that our page table defenses do not affect the overall performance of Bzip2 compression relative to Opt-VG. The standard deviation is 0%. Since Bzip2 accesses all the heap memory that it allocates when compressing the 32 MB file, our page table defenses do not incur any overhead by disabling demand paging of ghost memory.

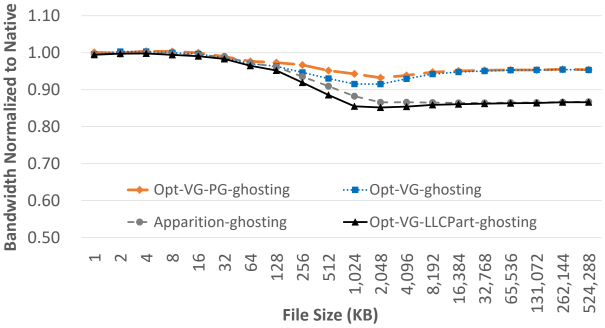

Figure 7: Ghosting OpenSSH Client File Transfer Bandwidth Normalized to Native FreeBSD

Ghosting OpenSSH Client: The Opt-VG-PG-Ghosting line in Figure 6 shows the overhead of our page table defenses on the unhalted CPU clock cycles (converted into time using the processor's clock frequency) of the ssh client transferring files. Each data point is the average of 20 rounds of execution. For 1 KB to 4 MB files, page table defenses increase the overhead of Opt-VG (denoted by the Opt-VG-ghosting line in Figure 6) by 1% to 10% with a 2% standard deviation. For large files, page table defenses add no overhead to the CPU time.

Figure 7 shows the overheads of our page table defenses on the client file transfer bandwidth. Page table defenses add no overhead to the optimized Virtual Ghost across all file sizes (differences are within the range of standard deviation).

Ghosting GnuPG: We enabled ghost memory for GnuPG for all systems except the native FreeBSD kernel. Table 8 shows the performance of signing files with GnuPG. The page table defenses incur a constant overhead of around 14 ms across all file sizes. This overhead

File Size (KB)	Native	Std. Dev.	Opt-VG	Opt-VG-PG	Opt-VG-LLCPart	Apparition
1	8.6	0.1	9.5	23.7	12.1	25.2
2	8.6	0.1	9.5	23.8	12.1	24.9
4	8.6	0.1	9.5	23.9	12.2	25.5
8	8.7	0.2	9.6	23.9	12.1	25.1
16	8.9	0.1	9.8	23.9	12.5	25.4
32	9.2	0.1	10.1	24.4	12.9	25.6
64	9.9	0.1	10.9	25.4	13.6	27.0
128	11.4	0.1	12.4	26.8	15.2	28.4
256	14.3	0.1	15.4	29.7	18.3	31.5
512	20.1	0.1	21.3	35.6	24.4	37.4
1024	31.6	0.1	33.2	47.7	36.4	49.4
2048	54.8	0.0	56.8	71.2	60.5	73.6
4096	100.9	0.1	103.9	118.2	108.0	121.1
8192	193.3	0.1	198.6	212.9	203.6	217.0
16384	377.8	0.2	386.2	400.1	394.6	407.3
32768	746.6	0.5	761.8	776.1	776.6	789.2

Table 8: GnuPG Signing Results. Time in milliseconds.

occurs because our page allocation defenses disable demand paging of ghost memory. `malloc()` attempts to fulfill allocation requests by allocating memory chunks with 4 MB alignment from the OS. This alignment constraint may cause `malloc()` to map a larger virtual memory region for the heap and return a pointer to an aligned 4 MB block within it. Although GnuPG only uses the aligned portion of memory, the page table defenses still allocate and map physical memory for the remaining unaligned 8 MB portion, incurring the 14 ms overhead. The overhead becomes negligible as the file size increases, as Table 8 shows. The standard deviation is 3% on average.

Ghosting Clang: As the fourth column of Table 7 shows, the page table defenses do not add any overhead to Clang relative to Opt-VG. This indicates that Clang uses most of the heap memory it allocates. Therefore, allocating and mapping physical memory at allocation time as opposed to on demand incurs no overhead.

8.4 LLC Side-Channel Defenses

We have compared the performance of various cache partition sizes with the baseline where the ghosting application, the kernel and the Apparition VM can all use the entire LLC. Our results indicate that the Apparition VM needs only 2 LLC ways to avoid performance degradation. We also experimentally determined that assigning 12, 2, and 2 LLC ways to the ghosting application, the kernel, and the Apparition VM, respectively, best achieves performance similar to the baseline. This provides ghosting applications the maximum number of LLC ways possible. While we use static partitions, we could leverage dynamic cache partitioning techniques e.g., SecDCP [70], to improve performance.

Ghosting RandomAccess: We use the RandomAccess microbenchmark in Section 8.1 to evaluate the impact of LLC partitioning when an application's working set is small enough to fit in the LLC but exceeds the capacity of the assigned partition. Since the 8 MB array is larger than the capacity of the 12-way partition of the 16-way 8 MB LLC, LLC partitioning increases the overhead of Opt-VG from 1.28× to 2.09× with a 3% standard deviation.

Ghosting Bzip2: We enabled ghost memory for Bzip2 for all systems except the native FreeBSD kernel. Table 7 shows the overhead of LLC partitioning on Bzip2 compressing a 32 MB file as Section 8.1 describes. LLC partitioning does not affect the performance of Bzip2, which indicates the capacity of the 12-way LLC partition is sufficient for the cache lines frequently accessed by Bzip2. The standard deviation is 0%.

Ghosting OpenSSH Client: We evaluate the overhead of LLC partitioning on OpenSSH client CPU time and bandwidth when transferring files of varying sizes; Figure 6 shows the file transfer CPU time normalized to the native FreeBSD 9.0 averaged over 20 rounds of execution. Opt-VG-LLCPart-ghosting (Opt-VG with LLC partitioning enabled) is 1.18× (on average with a worst case of 1.27×) across all file sizes (where Opt-VG is 1.09× on average) when normalized to FreeBSD. The overhead of LLC partitioning mainly comes from the LLC partition switches among the ghosting application, the kernel and the Apparition VM in the runtime, which slows down the performance by 1.16× on average across all file sizes. The standard deviation is 1% on average across all file sizes.

Figure 7 illustrates the performance impact of LLC partitioning on client file transfer bandwidth. The results are averaged over 20 rounds of execution. Opt-VG-LLCPart-ghosting reduces bandwidth to 0.91 that of native FreeBSD on average across all file sizes with a worst case of 0.85 (compared to 0.92 for Opt-VG). The standard deviation ranges from 0% to 1% across all file sizes.

Ghosting GnuPG: We enabled ghost memory for GnuPG for all systems except the native FreeBSD kernel. Table 8 shows the performance impact of LLC partitioning on GnuPG as Section 8.1 describes. For 1 KB to 4 MB files, LLC partitioning incurs a 3 ms to 4 ms overhead which is the overhead for maintaining i.e., switching among, different LLC partitions. For 8 MB to 32 MB files, although their sizes exceed the capacity of the 6 MB ghost memory LLC partition and the absolute additional execution time incurred by LLC partitioning is longer, the overhead to the overall performance is negligible. The execution time of Opt-VG-LLCPart for signing 8 MB to 32 MB files is 1.05× (Opt-VG is 1.02×) that for native FreeBSD on average. The standard deviation is 1.2% on average across all file sizes.

Ghosting Clang: Tables 7 and 9 show that our LLC side-channel defenses incur a negligible 3% overhead when assigning 12, 2 and 2 LLC ways to the ghosting Clang, the kernel, and the Apparition VM, respectively. However, when we shrink the number of LLC ways assigned to the ghosting Clang to 6, 4, and 2 while the LLC partition sizes of the kernel and the Apparition VM remain the same, we observe that the execution time for Opt-VG-LLCPart is as much as 1.1×, 1.3×, and 1.6× that of native FreeBSD. This is because the working set of Clang exceeds the capacity of the cache partition.

We also evaluated the overhead of LLC partitioning when executing more ghosting applications than the processor has partitions. As Section 7 describes, our pro-

# of LLC Ways	Overhead (×)	# of LLC Ways	Overhead (×)
2	1.64	8	1.08
4	1.30	10	1.05
6	1.14	12	1.03

Table 9: Overhead of Opt-VG with Varying Sizes of LLC partition for Ghosting Clang. Normalized to Native FreeBSD.

totype shares a single partition among multiple ghosting applications and flushes the cache on context switches between two ghosting applications. We run two ghosting Clang processes in parallel in the background, where each compiles either `gcc-smaller.c` or `gcc-pp.c` from SPEC CPU 2017 [5]. On native FreeBSD, it takes 57.3 seconds to compile `gcc-smaller.c` in this scenario; Compilation on Opt-VG-LLCPart takes 1.06× (1.03× for Opt-VG) the time on native FreeBSD, with a 0.4% standard deviation.

8.5 Evaluation of Combined Defenses

We now evaluate the combined overheads of our page table and LLC side-channel defenses using RandomAccess, Bzip2, the OpenSSH client, GnuPG, and Clang.

RandomAccess executes in 2.11× the time taken by native FreeBSD when executing on Apparition, as Table 7 shows; the standard deviation is 2%. The overhead mainly comes from the mitigations to LLC side-channel attacks. Table 7 also shows that Apparition with all defenses enabled on Bzip2 only adds 5% overhead (compared to Opt-VG's 4%) relative to native FreeBSD with 0% standard deviation.

Figure 6 shows the performance impact of all defenses on the OpenSSH client file transfer CPU time. The overhead of Apparition ranges from 16% to 33% relative to native FreeBSD, with a 1% standard deviation across all file sizes, which is a combination of the slow down incurred by page table and LLC side-channel defenses in addition to the overhead of Opt-VG. Figure 7 illustrates the performance impact of all defenses on the client file transfer rate. Apparition reduces the file transfer rate to 0.91 that of native FreeBSD on average across all file sizes with a worst case of 0.85 (compared to 0.92 for Opt-VG).

Table 8 shows that Apparition incurs a constant overhead of around 16 ms relative to Opt-VG on GnuPG across 1 KB to 4 MB files, 14 ms of which comes from the page table side-channel with the remaining from the LLC partitioning defenses. As Table 8 shows, the overhead of both defenses becomes negligible as the file size increases. The standard deviation is 3.0% on average across all file sizes.

Table 7 shows that the ghosting Clang compiler incurs 5% overhead relative to native FreeBSD with a standard deviation of 2% when running on Apparition.

9 Related Work

Recent work removes commodity OS kernels from the TCB. SP^3 [75], Overshadow [20], InkTag [40], CHAOS [18], and AppShield [21] build on commercial hypervisors and protect entire applications by providing an encrypted view of application memory to the OS and detect corruption of physical memory frames by the OS using digital signatures. Virtual Ghost [26] uses compiler instrumentation to insert run-time checks and can also protect entire applications. Hardware such as Intel SGX [23, 42] and AMD SEV [31, 39] protect unprivileged applications and virtual machines from malicious privileged code such as the OS and hypervisors. Haven [12] uses Intel SGX [23, 42] to isolate entire unmodified legacy applications from the OS. All of these shielding systems are vulnerable to side-channel attacks.

Page table side-channel attacks can steal secret application data on Intel SGX and InkTag [63, 67, 73]. T-SGX [62] transforms SGX applications to thwart page fault side channels by executing computations within Intel TSX transactions. TSX aborts transactions upon exceptions and interrupts, ensuring no page fault sequence leaks to the OS. However, its overhead ranges from 4% to 118% with a geometric mean of 50%. DÉJÀ VU [19] builds a software reference clock protected by Intel TSX transactions within SGX enclaves. It detects privileged side-channel attacks that trigger frequent traps and interrupts and aborts the application if an attack is detected.

Cache side-channel attacks are a known problem [36–38,43,52,58,76,79]. Several defenses partition the cache but generally assume an unprivileged attacker e.g., an unprivileged process [70, 71, 80] or a virtual machine attacking its neighbors [35, 44, 50, 61, 80]. These defenses cannot mitigate attacks by privileged code. Still, we can leverage techniques such as dynamic partitioning in SecDCP [70] to improve the performance of our cache partitioning scheme but, unlike SecDCP, ensure that the OS does not reconfigure or disable the partitioning.

Other mechanisms can mitigate cache side-channel attacks, but they also assume unprivileged attackers. SHARP [74] alters a shared cache's replacement policy to prevent the attacker from learning the victim's memory access patterns by cache evictions. It prioritizes evicting LLC cache lines that are not in any private L1 cache and the LLC cache lines of the current process. However, a compromised OS can still evict the cache lines of the victim as it can run on the victim's behalf. The Random Fill Cache Architecture [51] breaks the correlation between demand memory access and L1 cache fills to defend against reuse-based side-channel attacks. Wang and Lee [71] proposed that memory-to-cache mappings in L1 cache be dynamically randomized. Both approaches focus on L1 cache and may incur high performance overhead on much larger LLCs. Additionally, all three approaches require hardware modifications. FuzzyTime [41] and TimeWarp [53] introduce noise to the system clock to disrupt attackers' time measurements but hurt programs needing a high-precision clock.

Some approaches detect, rather than prevent, cache side-channel attacks. Chiappetta et al. [22] detect cache side channels by finding correlations between the LLC accesses of the attacker and the victim. HexPADS [59] detects cache side channels based on the frequent cache misses of the attacker. However, both approaches tend to suffer from high false positives and false negatives.

A final approach is to design hardware without side channels and formally verify that they are correct. SecVerilog [77] and Sapper [48] present new hardware description languages with information flow tracking that processor designers can use to design processors without timing-channel exploits. Sanctum [24] is an isolation framework similar to Intel SGX that mitigates page table and cache side-channel attacks by maintaining a per-enclave page table in addition to the traditional page table managed by the OS with extra registers and logic. It also isolates the enclaves in both DRAM and cache using page coloring maintained by the TCB. However, these defenses require hardware modifications.

10 Conclusions

Despite defenses such as InkTag [40], Virtual Ghost [26], and Haven [12], compromised OS kernels can steal application data via side-channel attacks. We present Apparition, an enhanced Virtual Ghost system that protects applications from page table and LLC side-channel attacks. Apparition improves the performance of the original Virtual Ghost by up to 2× by eliminating unnecessary serializing instructions and by utilizing Intel MPX. Apparition also enhances Virtual Ghost's memory protection features to thwart page table side-channel attacks and combines its memory protection features with Intel's CAT hardware to defeat LLC side-channel attacks. Apparition requires no changes to the processor or OS kernels running on SVA. We compared Apparition's performance to Virtual Ghost enhanced with our optimizations; it adds 1% to 18% overhead (relative to native FreeBSD) to most of the real-world applications we tested but adds up to 86% additional overhead to GnuPG.

Acknowledgements

The authors thank the anonymous reviewers for their insightful feedback. This work was supported by NSF Awards CNS-1319353, CNS-1618497, CNS-1618588, CNS-1629770, and CNS-1652280.

References

[1] clang: a C language family frontend for LLVM. https://clang.llvm.org.

[2] *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R Edition*. 2011.

[3] *ARM Architecture Reference Manual: ARMv8, for ARMv8-A Architecture Profile*. 2014.

[4] *Intel 64 and IA-32 Architectures Software Developer's Manual*, vol. 3. Intel, September 2016.

[5] SPEC CPU® 2017. https://www.spec.org/cpu2017, 2017.

[6] Intel analysis of speculative execution side channels. Tech. Rep. 336983-003, May 2018.

[7] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information Systems Security 13* (November 2009), 4:1–4:40.

[8] ACIIÇMEZ, O. Yet another microarchitectural attack: Exploiting I-cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture* (2007), CSAW'07, pp. 11–18.

[9] ACIIÇMEZ, O., BRUMLEY, B. B., AND GRABHER, P. New results on instruction cache attacks. In *Proceedings of the 12th International Conference on Cryptographic Hardware and Embedded Systems* (2010), CHES'10, pp. 110–124.

[10] ACIIÇMEZ, O., AND SCHINDLER, W. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In *Proceedings of the 2008 The Cryptographers' Track at the RSA Conference on Topics in Cryptology* (2008), CT-RSA'08, pp. 256–273.

[11] ARM LIMITED. ARM security technology: Building a secure system using TrustZone technology, 2009.

[12] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with Haven. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014), OSDI'14, pp. 267–283.

[13] BENGER, N., POL, J., SMART, N. P., AND YAROM, Y. "ooh aah... just a little bit": A small amount of side channel can go a long way. In *Proceedings of the 16th International Workshop on Cryptographic Hardware and Embedded Systems* (2014), CHES'14, pp. 75–92.

[14] BHARGAVA, R., SEREBRIN, B., SPADINI, F., AND MANNE, S. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (2008), ASPLOS'08, pp. 26–35.

[15] BOVET, D. P., AND CESATI, M. *Understanding the LINUX Kernel*, 3^{rd} ed. O'Reilly, Sebastopol, CA, 2006.

[16] BZIP2. bzip2 and libbzip2, 1996. http://www.bzip.org.

[17] CHECKOWAY, S., AND SHACHAM, H. 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* (2013), ASPLOS'13, pp. 253–264.

[18] CHEN, H., ZHANG, F., CHEN, C., YANG, Z., CHEN, R., ZANG, B., AND MAO, W. Tamper-resistant execution in an untrusted operating system using a virtual machine monitor. Tech. rep., Fudan University, Parallel Processing Insitute, 2007.

[19] CHEN, S., ZHANG, X., REITER, M. K., AND ZHANG, Y. Detecting privileged side-channel attacks in shielded execution with DÉJÀ VU. In *Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security* (2017), ASIA CCS'17, pp. 7–18.

[20] CHEN, X., GARFINKEL, T., LEWIS, E. C., SUBRAHMANYAM, P., WALDSPURGER, C. A., BONEH, D., DWOSKIN, J., AND PORTS, D. R. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (2008), ASPLOS'08, pp. 2–13.

[21] CHENG, Y., DING, X., AND DENG, R. H. Efficient virtualization-based application protection against untrusted operating system. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (2015), ASIA CCS'15, pp. 345–356.

[22] CHIAPPETTA, M., SAVAS, E., AND YILMAZ, C. Real time detection of cache-based side-channel attacks using hardware performance counters. *Appl. Soft Comput. 49*, C (Dec. 2016), 1162–1174.

[23] COSTAN, V., AND DEVADAS, S. Intel SGX explained. *IACR Cryptology ePrint Archive 2016* (2016), 86.

[24] COSTAN, V., LEBEDEV, I., AND DEVADAS, S. Sanctum: Minimal hardware extensions for strong software isolation. In *Proceedings of the 25th USENIX Security Symposium* (2016), SEC'16, pp. 857–874.

[25] CRISWELL, J. *Secure Virtual Architecture: Security for Commodity Software Systems*. PhD thesis, Computer Science Department, University of Illinois at Urbana-Champaign, Urbana, IL, August 2014.

[26] CRISWELL, J., DAUTENHAHN, N., AND ADVE, V. Virtual Ghost: Protecting applications from hostile operating systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (2014), ASPLOS'14.

[27] CRISWELL, J., GEOFFRAY, N., AND ADVE, V. Memory safety for low-level software/hardware interactions. In *Proceedings of the 18th Usenix Security Symposium* (2009), SEC'09.

[28] CRISWELL, J., LENHARTH, A., DHURJATI, D., AND ADVE, V. Secure Virtual Architecture: A safe execution environment for commodity operating systems. In *Proceedings of the ACM Symposium on Operating System Principles* (2007), SOSP'07.

[29] CRISWELL, J., MONROE, B., AND ADVE, V. A virtual instruction set interface for operating system kernels. In *Workshop on the Interaction between Operating Systems and Computer Architecture* (2006), WIOSCA'06, pp. 26–33.

[30] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems* (October 1991), 13(4):451–490.

[31] D. KAPLAN, J. P., AND WOLLER, T. *White Paper AMD Memory Encryption*. AMD, 4 2016.

[32] DENNING, D. E. A lattice model of secure information flow. *Commun. ACM 19*, 5 (May 1976), 236–243.

[33] DHURJATI, D., KOWSHIK, S., AND ADVE, V. SAFECode: Enforcing alias analysis for weakly typed languages. In *ACM Conference on Programming Language Design and Implementation* (2006), PLDI'06.

[34] DONG, X., SHEN, Z., CRISWELL, J., COX, A., AND DWARKADAS, S. Spectres, Virtual Ghosts, and hardware support. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy* (2018), HASP'18, pp. 5:1–5:9.

[35] GODFREY, M. On the prevention of cache-based side-channel attacks in a cloud environment. Master's thesis, School of Computing, Queen's University, Kingston, Ontario, Canada, Sept 2013.

[36] GRUSS, D., SPREITZER, R., AND MANGARD, S. Cache template attacks: Automating attacks on inclusive last-level caches. In *Proceedings of the 24th USENIX Security Symposium* (2015), SEC'15, pp. 897–912.

[37] GULLASCH, D., BANGERTER, E., AND KRENN, S. Cache games – bringing access-based cache attacks on AES to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy* (2011), SP '11, pp. 490–505.

[38] HÄHNEL, M., CUI, W., AND PEINADO, M. High-resolution side channels for untrusted operating systems. In *Proceedings of the 2017 USENIX Annual Technical Conference* (2017), pp. 299–312.

[39] HETZELT, F., AND BUHREN, R. Security analysis of encrypted virtual machines. In *Proceedings of the 13th ACM International Conference on Virtual Execution Environments* (2017), VEE'17, pp. 129–142.

[40] HOFMANN, O. S., KIM, S., DUNN, A. M., LEE, M. Z., AND WITCHEL, E. InkTag: secure applications on an untrusted operating system. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems* (2013), ASPLOS'13, pp. 265–278.

[41] HU, W.-M. Reducing timing channels with fuzzy time. *J. Comput. Secur. 1*, 3-4 (May 1992), 233–254.

[42] INTEL. *Software Guard Extensions Programming Reference*, October 2014. Document Number: 3329298-002.

[43] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. S$A: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (May 2015), SP'15, pp. 591–604.

[44] KIM, T., PEINADO, M., AND MAINAR-RUIZ, G. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *Presented as part of the 21st USENIX Security Symposium* (2012), pp. 189–204.

[45] KOCH, W. GnuPG, 2017. https://gnupg.org.

[46] KOCHER, P., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre attacks: Exploiting speculative execution.

[47] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the Conference on Code Generation and Optimization* (2004), CGO'04, pp. 75–88.

[48] LI, X., KASHYAP, V., OBERG, J. K., TIWARI, M., RAJARATHINAM, V. R., KASTNER, R., SHERWOOD, T., HARDEKOPF, B., AND CHONG, F. T. Sapper: A language for hardware-level security policy enforcement. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (2014), ASPLOS'14, pp. 97–112.

[49] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., MANGARD, S., KOCHER, P., GENKIN, D., YAROMN, Y., AND HAMBURG, M. Meltdown.

[50] Liu, F., Ge, Q., Yarom, Y., Mckeen, F., Rozas, C., Heiser, G., and Lee, R. B. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture* (2016), HPCA'16, pp. 406–418.

[51] Liu, F., and Lee, R. B. Random fill cache architecture. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (2014), MICRO'14, pp. 203–215.

[52] Liu, F., Yarom, Y., Ge, Q., Heiser, G., and Lee, R. B. Last-level cache side-channel attacks are practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (2015), SP'15, pp. 605–622.

[53] Martin, R., Demme, J., and Sethumadhavan, S. Time-Warp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (2012), ISCA'12, pp. 118–129.

[54] McKusick, M. K., Neville-Neil, G. V., and Watson, R. N. M. *The Design and Implementation of the FreeBSD Operating System*, second ed. Pearson Education, 2015.

[55] McVoy, L., and Staelin, C. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference* (1996), ATC'96, pp. 23–23.

[56] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 ACM Conference on Programming Language Design and Implementation* (2009), PLDI'09, pp. 245–258.

[57] Navarro, J., Iyer, S., Druschel, P., and Cox, A. Practical, transparent operating system support for superpages. *SIGOPS Oper. Syst. Rev. 36*, SI (Dec. 2002), 89–104.

[58] Osvik, D. A., Shamir, A., and Tromer, E. Cache attacks and countermeasures: The case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology* (2006), CT-RSA'06, pp. 1–20.

[59] Payer, M. HexPADS: A platform to detect "stealth" attacks. In *Proceedings of the 8th International Symposium on Engineering Secure Software and Systems - Volume 9639* (2016), ESSoS'16, pp. 138–154.

[60] Russinovich, M. E., and Solomon, D. A. *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000 (Pro-Developer)*. Microsoft Press, Redmond, WA, USA, 2004.

[61] Shi, J., Song, X., Chen, H., and Zang, B. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops* (2011), DSN-W'11, pp. 194–199.

[62] Shih, M.-W., Lee, S., Kim, T., and Peinado, M. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the Network Distributed Security Symposium*.

[63] Shinde, S., Chua, Z. L., Narayanan, V., and Saxena, P. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security* (2016), ASIA CCS'16, pp. 317–328.

[64] Singh, A. *Mac OS X Internals*. Addison-Wesley Professional, 2006.

[65] The OpenBSD Project. OpenSSH, 2014. https://www.openssh.com.

[66] Tromer, E., Osvik, D. A., and Shamir, A. Efficient cache attacks on AES, and countermeasures. *J. Cryptol. 23*, 1 (Jan. 2010), 37–71.

[67] Van Bulck, J., Weichbrodt, N., Kapitza, R., Piessens, F., and Strackx, R. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the 26th USENIX Security Symposium* (2017), SEC'17, pp. 1041–1056.

[68] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (1993), SOSP'93.

[69] Wang, W., Chen, G., Pan, X., Zhang, Y., Wang, X., Bindschaedler, V., Tang, H., and Gunter, C. A. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proceedings of the 2017 ACM Conference on Computer and Communications Security* (2017), CCS'17, pp. 2421–2434.

[70] Wang, Y., Ferraiuolo, A., Zhang, D., Myers, A. C., and Suh, G. E. SecDCP: Secure dynamic cache partitioning for efficient timing channel protection. In *Proceedings of the 53rd Annual Design Automation Conference* (2016), DAC'16, pp. 74:1–74:6.

[71] Wang, Z., and Lee, R. B. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (2007), ISCA'07, pp. 494–505.

[72] Wheeler, D. A. SLOCCount Version 2.26, 2004.

[73] Xu, Y., Cui, W., and Peinado, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (2015), pp. 640–656.

[74] Yan, M., Gopireddy, B., Shull, T., and Torrellas, J. 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* (2017), ISCA'17, pp. 347–360.

[75] Yang, J., and Shin, K. G. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of the 4th ACM International Conference on Virtual Execution Environments* (2008), VEE'08, pp. 71–80.

[76] Yarom, Y., and Falkner, K. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium* (2014), SEC'14, pp. 719–732.

[77] Zhang, D., Wang, Y., Suh, G. E., and Myers, A. C. A hardware design language for timing-sensitive information-flow security. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems* (2015), ASPLOS'15, pp. 503–516.

[78] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (2012), CCS'12, pp. 305–316.

[79] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-tenant side-channel attacks in PaaS clouds. In *Proceedings of the 2014 ACM Conference on Computer and Communications Security* (2014), CCS'14, pp. 990–1003.

[80] Zhou, Z., Reiter, M. K., and Zhang, Y. A software approach to defeating side channels in last-level caches. In *Proceedings of the 2016 ACM Conference on Computer and Communications Security* (2016), CCS'16, pp. 871–882.