Hardening Hypervisors with Ombro

Ethan Johnson, Colin Pronovost, and John Criswell
Department of Computer Science, University of Rochester

Abstract

This paper presents Ombro, a low-level virtual instruction set architecture (vISA) which enforces compiler-based security policies on real-world commodity hypervisors. We extend the Secure Virtual Architecture (which itself extends the LLVM compiler's Intermediate Representation) to support the full set of hardware operations needed to run an x86 commodity hypervisor used in some of the world's largest public clouds, namely, the Xen 4.12 hypervisor, running in full hardware-accelerated mode using Intel's Virtual Machine Extensions (VMX). We have ported Xen 4.12 to the Ombro vISA and demonstrated that it can run unmodified guest VMs of real-world relevance (namely, Linux guests under Xen's HVM and PVH modes). Furthermore, to demonstrate Ombro's ability to harden hypervisors from attack, Ombro implements control flow integrity and the first protected shadow (split) stack for x86 hypervisors. Our performance results show that Ombro achieves this protection without imposing measurable overheads on most application benchmarks.

1 Introduction

Various ideas have been proposed and demonstrated that can improve hypervisor security against low-level attacks such as memory safety vulnerabilities. One such approach would be to re-implement the hypervisor in a safe language such as Rust [1], but this is considered prohibitively labor-intensive for real-world hypervisors such as Xen [3], VirtualBox [46], or Hyper-V [42]. Another is to provide a whole-VM trusted execution environment (TEE) that protects VMs from a compromised hypervisor by isolating and removing most of the hypervisor from the trusted computing base [32, 41, 55, 56]; this approach is powerful and promising but tends to impose heavyweight requirements (e.g. moving the non-trusted portion of the hypervisor into VMX/SVM's non-root (guest) mode, which imposes high overheads e.g. on VM exit handling) [41, 56] or substantial modifications to hardware [32, 55]. Enclave-based TEEs such as Intel's SGX [30] offer similar benefits but provide a more functionally limited operating environment compared to whole-VM TEEs. Designs such as HyperSafe [51], which adds lightweight control flow integrity (CFI) protection to the hypervisor, make hypervisors more resilient against attack but face an "arms race" of rapidly evolving attacks [7, 8, 27, 29], necessitating the addition of further defenses such as a shadow stack (HyperSafe [51] is vulnerable to attacks that corrupt return addresses) to remain viable.

When applied to kernel-mode software like hypervisors, hardening approaches such as CFI must account for the fact that the raw hardware/software interface provided by the native ISA is much "messier" than the execution environment user-mode applications can expect. Low-level operations which are typically thought of as transparent and orthogonal, such as context switches, VMX/SVM guest entry/exit, page table updates, and control register modifications, are fully exposed to host-kernel-mode software. These operations present numerous opportunities for security invariants such as CFI enforcement to be undermined when software logic has been corrupted by a memory safety exploit.

Prior work on the Secure Virtual Architecture (SVA) [15, 17] addresses this issue by extending the LLVM compiler's Intermediate Representation (IR) with *virtual instructions* (a.k.a. *intrinsics*) that encapsulate these low-level hardware/software interactions with principled, higher-level abstractions intended for use by kernel-mode code. Program state discontinuities that could break security enforcement are prevented, since operations such as context switches and paging updates are handled safely by a thin, trusted layer of code provided by the compiler to implement the virtual instructions, whose behavior cannot be compromised by bugs in a kernel or hypervisor built upon them. To date, SVA has been used to successfully enforce security policies such as memory safety [16] and CFI [13] on commodity Linux and FreeBSD OS kernels. Initial support for hardware-accelerated virtual machines via Intel VMX has been added to SVA [34], but it lacks several key features needed to support real production hypervisors such as Xen [3], VirtualBox [46] and Hyper-V [42], and also lacks SMP (multiprocessor) support, a necessity in the modern cloud.

In this paper, we present *Ombro*, a low-level virtual instruction set architecture (vISA) designed to support the efficient and complete implementation of compiler-based security mitigations in real-world commodity hypervisors. We extend SVA to support the full set of hardware features needed to support the Xen 4.12 hypervisor [3], including virtual APIC support, model-specific register (MSR) virtualization, and I/O port virtualization. We also add the first SMP support to SVA (benefiting non-hypervisor designs as well that are based on it) and make several key design improvements to SVA's existing VMX support [34] to address shortcomings in performance and its ability to integrate with the Xen codebase. We have ported Xen 4.12 to the Ombro vISA and demonstrate that it can run unmodified guest VMs of real-world relevance (Linux guests under Xen's HVM and PVH modes) with negligible performance impacts on most application benchmarks. Additionally, as a case study in the kinds of hypervisor security mitigations whose design and implementation Ombro sup-

ports, we demonstrate the implementation of control flow integrity with return address protection (shadow/split stack) using the tools provided by Ombro, and that these mitigations add no further performance impacts to guest operations.

To summarize, our contributions are as follows:

- We have enhanced the SVA vISA developed in Shade [34] to support a full-featured production-quality hypervisor (namely, Xen). We have also added symmetric multiprocessing (SMP) support to the SVA vISA.
- We have developed techniques that ensure that bugs in hypervisors cannot break return address and control flow integrity. We have created a prototype of a system, dubbed Ombro, that uses our enhanced SVA vISA to enforce these policies on a production-quality hypervisor.
- We have ported the Xen hypervisor to Ombro. This is the first full port of a full-featured production-quality hypervisor to the SVA virtual instruction set.
- We have evaluated the performance of Ombro and found that the vISA imposes negligible performance impacts on most guest application benchmarks. We have also found that the addition of CFI and return address protection imposes no measurable overheads.

2 Background

Ombro employs *virtual instruction set computing* (VISC) [5, 16] to ensure that its security guarantees (which mitigate control-flow hijacking) are not bypassed via low-level interactions between the hypervisor and the x86 hardware. Here, we present background information on VISC, the VISC-based Secure Virtual Architecture (SVA), and its features and limitations relevant to Ombro.

2.1 Virtual Instruction Set Computing

Virtual instruction set computing (VISC) [5] is a system design in which the instruction set to which software is compiled (the *virtual instruction set* or *vISA*) is decoupled from the instruction set implemented by the processor (the *native instruction set*). A trusted code generator translates code from the virtual instruction set to the native instruction set. This translation can occur at any time (at compile time, link time, install time, boot time, or just-in-time during program execution). The defining characteristic of VISC is that all software in the system must be translated from virtual instruction set code to native instruction set code. In an idealized theoretical VISC system, this includes applications as well as system software (e.g. operating system kernels and hypervisor executives), i.e., all code on the system must target the virtual ISA rather than the native ISA. Practical designs may elect to relax this requirement within specific domains (e.g. applications or guest VMs running in less-privileged hardware modes) to maintain support for existing native-code applications.

Secure Virtual Architecture (SVA) [15,16] is a VISC infrastructure that leverages the trusted code generator to enforce security policies on all software in the system stack, including the OS kernel and (optionally) library and application code. Because software must be translated by SVA's trusted code generator, SVA can instrument code during native code generation to enforce security policies. SVA's virtual instruction set is an extended version of the original LLVM Intermediate Representation (LLVM IR) [39], allowing SVA to use aggressive static analysis to optimize away provably unnecessary run-time security checks.

SVA extends LLVM IR with new virtual instructions (called *intrinsics*) to support low-level privileged operations such as I/O, MMU configuration, and context switching in kernel-mode software without the need for native assembly code or direct access to privileged in-memory hardware data structures (page tables, etc.) [15]. These intrinsics are implemented by a small library of trusted code (the "SVA-OS runtime") which is, architecturally, considered part of the compiler, and linked or inlined into the target program as necessary during translation from virtual to native code. The resultant vISA provided to kernel programmers is designed to make it impossible to express computations that would violate the security policies specified for the system. To the extent that goal cannot be ensured statically, the SVA-OS implementation vets inputs and sanitizes outputs at runtime to prevent raw hardware functions from being used in unsafe ways.

To prevent attacker-compromised kernel-mode software from simply bypassing the vISA by executing native code, newer versions of SVA [13–15, 22, 34] enforce code segment integrity on the kernel by using software fault isolation [50] to prevent it from utilizing any kernelspace page-table mappings that are both writable and executable. This ensures that all native code has been translated or validated by the SVA code generator (either ahead of time or by request at runtime) and contains any necessary instrumentation while not containing privileged native instructions that would bypass the vISA.

Because it is possible to fully implement kernel-mode system software (e.g., OS kernels or hypervisors) using the SVA vISA's virtual instructions, compiler-based security transformations such as control flow integrity (CFI) [4], software fault isolation (SFI) [50], or memory safety [20] enforcement can be performed on it at the LLVM IR level without "blind spots" arising from opaque native assembly or instructions with privileged side effects. SVA has been used to safely and efficiently support a variety of security policies and popular OS kernels over the years. The original SVA prototype enforced both spatial and temporal memory safety on the Linux 2.4 kernel [15, 16]. Subsequent iterations exchanged full memory safety for low-overhead CFI enforced on the FreeBSD 9.0 kernel [13] and explored a novel application of lightweight SFI instrumentation on the kernel to protect userspace applications from a compromised kernel [14]. Other SVA derivatives have explored adding protection against cache and speculative

side-channel threats against protected userspace applications in a Virtual Ghost system [21, 22, 34].

2.2 Kernel-Mode Memory Protection in SVA

Starting with the *Virtual Ghost* project [14], SVA has supported protecting designated sections of the host-virtual address space against tampering by kernel-mode (Ring 0) code using software fault isolation (SFI) [50]. This protection can be used as a foundation for multiple security policies and can optionally protect portions of userspace as well as kernelspace (lower and upper halves of the address space).

The SVA native code translator instruments all load and store instructions within host-kernel-mode code with SFI checks that determine whether the access references a virtual address within the protected region. If so, the check detects a violation and generates a trap, allowing the SVA VM to take corrective action, such as alerting the system administrator or terminating system execution. Otherwise, the load or store is allowed to proceed normally [14].

SVA's SFI checks can be implemented using traditional bitmasking instructions [50] or a fast scheme developed for Apparition [22] based on Intel's Memory Protection Extensions (MPX) [30]. The architecture is flexible and can be readily adapted to other hardware protection mechanisms such as segmentation [30] or memory protection keys [28, 30].

The memory region(s) protected by these SFI checks can be used to store user or kernel secrets where they cannot be seen or modified by the kernel/hypervisor. Depending on the needs of particular threat models, enforcement designs, and system performance, SFI checks can be omitted on loads so as to protect data against tampering even if it does not need to be secret [14].

SVA uses this memory protection in its design to make its other enforcement mechanisms complete while maintaining high performance. For instance, SVA maintains its own direct map (one-to-one mapping of all physical memory) within the kernelspace SFI-protected region, allowing intrinsics to write to page tables and code pages while leaving the kernel's mappings to them read-only; this avoids the need to expensively switch page tables or mappings on every such intrinsic call [22]. SVA likewise uses its SFI to ensure the integrity of its own metadata, such as tables tracking the permitted and current usage types of each physical memory frame [15, 22].

In Virtual Ghost [14], Apparition [22], and Shade [34], a userspace SFI-protected region is used to hide application secrets from a compromised OS kernel. In Ombro, we will use SVA's SFI to protect hypervisor control stacks and enforce return address integrity (Section 7).

2.3 VMX Support in SVA

Shade [34] added initial support for Intel VMX to SVA. Shade extended the SVA vISA with intrinsics and conceptual idioms for management of hardware-accelerated guest VMs—specifically, VM entry and exit (world switches), extended paging, and VMCS management—while ensuring that these newly introduced capabilities would not compromise SVA's ability to enforce the security policies introduced in prior work (specifically Virtual Ghost [14] and Apparition [22]).

VMX facilitates *world switches* (the host/guest context switches associated with VM entry and exit) by giving the hypervisor open-ended control over each state element—instruction pointer, control and segment registers, etc.—as an individual field within a special in-memory data structure called the Virtual Machine Control Structure (VMCS) [30]. Different state elements are handled differently according to their importance in maintaining consistent system operation, and this behavior can (to an extent) be controlled by the hypervisor via flags in the VMCS. Some fields can be bidirectionally saved/restored by the processor as part of the host/guest-mode transition; for others, the hypervisor is expected to load an arbitrary value into the VMCS to which the register will be reset on VM exit. Others, such as the general purpose registers, are untouched by entry and exit, requiring the hypervisor to save and restore them itself.

This makes security enforcement on hypervisors challenging because the architecture implicitly assumes the hypervisor can be trusted. The ability to set host-state fields and entry/exit control flags in the VMCS affords countless opportunities for a compromised hypervisor to exploit VM exit to escape CFI enforcement and other security measures. Besides the instruction pointer itself, fields such as the stack pointer, segment registers, and control registers (which can disable security features such as protected mode, No-Execute (NX) pages, and SMEP [30]) can completely redefine the hypervisor's environment, rendering many protections useless.

Shade addresses these issues in SVA by encapsulating the VM entry/exit process (`VMLAUNCH`/`VMRESUME`) into an SVA intrinsic, `runvm`, which handles switching of sensitive state during world switches [34]. `runvm` has the semantics of a self-contained function call, contrasting with the broad, open-ended modification of system state possible with the native interface. Shade keeps context-switched guest state for the host and each guest VM within SFI-protected SVA internal memory (Section 2.2), providing access to individual guest state fields only through targeted intrinsics. The VMCS itself is likewise stored in protected memory and accessible to the hypervisor through intrinsics (`read`/`writevmcs`) which only permit access to non-sensitive fields (or vet/sanitize input/output for partially sensitive fields). VMCS fields related to features that SVA needs to control at a higher level (e.g. extended paging) are blocked by `read`/`writevmcs`, forcing the hypervisor to use the appropriate higher-level intrinsics.

While Shade laid important high-level groundwork for safely supporting the use of VMX acceleration in an SVA-based system, it fell short of being able to support a full-scale commodity hypervisor like Xen (or even a lightweight virtualization support driver like KVM [36]). Shade was developed and evaluated with a minimalist "toy" hypervisor that exer-

cised core VMX operations without the complexity of a full hypervisor. This facilitated the initial design and debugging of complex intrinsics such as `runvm`, but precluded an end-to-end performance evaluation and left unclear the question of whether the design choices made in the vISA would truly be conducive to porting a real hypervisor without invasive code changes or performance impacts. Shade also lacked support for key VMX features important to real-world hypervisors such as accelerated interrupt controller (APIC), model-specific register (MSR), and I/O port virtualization [30], and only supported vetting of a small subset of VMCS fields. Our work remedies these shortcomings, allowing us to port Xen 4.12 to the SVA vISA and use it to enforce and evaluate a security policy of real-world interest (return address protection for CFI) on a real-world hypervisor.

3 Threat Model

Our threat model assumes that we have a system running a single bare-metal hypervisor, such as Xen, on the hardware. This hypervisor hosts one or more guest virtual machines running various operating systems. The hypervisor is benign but may have exploitable memory safety errors that permit control-flow hijacking attacks such as return-to-libc [49] and return-oriented programming (ROP) [47] attacks. As we want to mitigate advanced control-flow hijacking attacks [7, 8, 18, 27], our defense must protect the integrity (but not necessarily confidentiality) of return and return-from-interrupt addresses. Non-control data attacks [9], Data-Oriented Programming (DOP) attacks [29], and other memory safety attacks that do not corrupt return addresses, function pointers, and other control data are out of scope.

4 Design

In this work, we present three major design contributions:

1. We extend and improve upon the SVA virtual instruction set architecture (vISA) of the Shade [34] project to efficiently support the full set of hardware features needed to run the Xen 4.12 hypervisor [3] in support of guest VMs accelerated using the Virtual Machine Extensions (VMX) [30] feaures of modern Intel x86-64 processors.

2. We extend the SVA vISA to support symmetric multiprocessing (SMP), an essential feature of modern systems that historical SVA designs notably lacked.

3. We present a design for an efficient and straightforward scheme enforcing forward-edge control flow integrity (CFI) with return address integrity (i.e. backward-edge CFI). This serves as a case study in how the SVA vISA can be used to support sound and efficient enforcement of security policies on commodity hypervisors, and also represents an advancement in its own right on the state of the art [51] as the first efficient protected shadow stack design for a hypervisor.

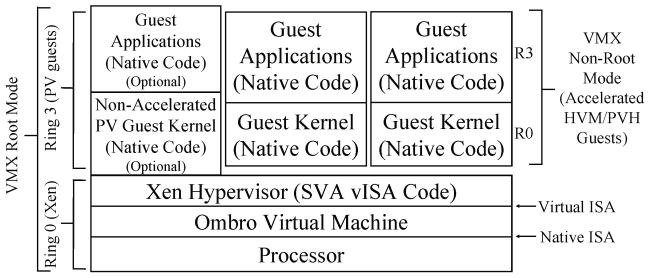

Figure 1: Ombro Architecture

Figure 1 depicts the overall architecture of Ombro as used to enforce security policies on the Xen 4.12 hypervisor.[1] Ombro permits the hypervisor to run in the processor's highest-privileged mode—Ring 0 of VMX root mode—without relying on hardware privilege isolation to enforce security policies on the hypervisor. The hypervisor is compiled to the SVA vISA rather than native code, preventing computations from being expressed that could violate security policy. The *Ombro Virtual Machine*'s[2] SVA-OS runtime support library—a thin layer of trusted native code provided by the SVA native code translator (i.e. the compiler; see Section 2.1) to implement the dynamic security checks required to safely implement the vISA on native hardware—runs alongside the hypervisor in this fully privileged mode, having been linked or inlined directly into the compiled hypervisor by the SVA native code translator. Software fault isolation (SFI) [50] (Section 2.2) and control flow integrity (CFI) [4] provided by SVA ensure that the hypervisor, despite running with full hardware privileges, cannot compromise the Ombro Virtual Machine or escape the vISA to run unrestricted native code.

Notably, Ombro does *not* require guest VM code (or host userspace application code for an OS-resident hypervisor) to be compiled to the SVA vISA. The hypervisor/host OS continues to employ standard x86 privilege isolation features (namely, privilege rings and VMX root/non-root modes) to isolate guest VMs and userspace applications from the hypervisor/kernel and each other, allowing guests and applications to run unmodified native code. Only the hypervisor/host kernel itself must be ported to the SVA vISA so that it can be safely controlled within the processor's most privileged mode.

Section 5 presents the extensions and improvements we make to the Shade [34] version of the SVA vISA to support a full commodity hypervisor (Xen 4.12). Section 6 describes specific enhancements to the SVA design that are necessary to support multi-processor systems, addressing a weakness in prior SVA work. Section 7 presents a design that utilizes the SVA vISA to efficiently and soundly enforce return address

[1] Our design supports any x86-64 hypervisor in principle, including those integrated with a host OS kernel; for concreteness, we focus on Xen in this work. In an OS-resident hypervisor, host userspace applications take the place of non-accelerated paravirtual guests in Ring 3 of VMX root mode.

[2] "Virtual machine" here refers to the language-theoretic sense of the term, not to the concept of a "guest virtual machine" provided by hypervisors. We aim to consistently use "guest VM" to refer to the latter throughout this work.

integrity on Xen to defend the hypervisor against advanced control-flow hijacking attacks.

Guest Virtualization Modes Supported. The Xen 4.12 hypervisor, for legacy reasons, supports a variety of operating modes for guest VMs on the x86-64 platform, which can be used simultaneously for different guests [54]. These represent a continuum of hardware acceleration usage and guest OS support, ranging from the traditional non-accelerated paravirtualization described in the original Xen paper [23] (referred to as "PV" mode in Xen's documentation [54]), to VMX-accelerated full virtualization supporting native guests ("HVM" mode), to a modern paravirtual approach utilizing VMX acceleration with Xen-aware guests ("PVH" mode).

Because classic PV mode is being de-emphasized by the Xen project and likely to be phased out in the future [54], Ombro's design only supports VMX-accelerated guests (i.e. HVM and PVH) and does not provide a complete set of intrinsics for the hypervisor to support PV guests running in usermode (host Ring 3). However, doing so would be a straightforward extension of the existing SVA vISA, which fully supports [14–16, 22, 34]) OS kernels with userspace applications in Ring 3. Our prototype (Section 8) does exactly this (in a limited way with some shortcuts) for implementation convenience. At times, we refer to PV concepts in design sections for the sake of clarity to readers familiar with Xen. They are not, however, relevant to our security design, as the design is intended for a modern Xen installation utilizing HVM/PVH for all guests including the control domain (*dom0*).

Terminology: Guest VMs, Domains, vCPUs, etc. SVA interfaces (following Intel's convention) use the term "guest virtual machine" to refer to a singular guest CPU virtualized by a hypervisor. Each such guest CPU exists in one-to-one correspondence with a Virtual Machine Control Structure (VMCS) [30]. From the physical CPU's and SVA's perspective, it does not matter how the hypervisor may choose to group those virtual CPUs together. However, this distinction is important to Xen, which refers to the overall VM (which may include multiple emulated CPUs) as a "domain" and individual virtualized CPUs as "vCPUs". Each vCPU thus corresponds to exactly one VMCS, and to one "guest VM" from SVA's perspective. Xen performs context switches on a per-vCPU basis, not per-domain (different vCPUs from the same or different domains are context switched as independent entities), which is in line with Intel's and SVA's perspective. In this paper, we sometimes refer to "vCPUs" rather than "guest VMs" when this distinction is important.

5 vISA Additions and Improvements

Our experience porting Xen to the SVA vISA developed for Shade [34] led us to extend and improve upon the vISA, adding missing support for hardware virtualization elements used by Xen and mitigating disruptions to Xen's performance and code structure. We describe these improvements below.

5.1 Securing Higher-Level VMX Features

The Intel VMX feature set is controlled primarily through control and state fields in a large (page-sized) in-memory, per-vCPU data structure called the Virtual Machine Control Structure (VMCS) [30]. Most VMX settings and subfeatures are controlled straightforwardly via individual VMCS fields or bits within a field consolidating similar controls. Some subfeatures, however, are more complex and are spread across multiple VMCS fields; some even utilize subsidiary control pages that the hypervisor must provide and link into the main VMCS by storing (host-)physical pointers into specific fields.

Shade's vISA support for the VMCS (Section 2.3) is insufficient to manage multi-field controls and substructures because its `read`/`writevmcs` intrinsics operate on individual fields and cannot readily account for behavioral dependencies between them (e.g. fields activated by bits in other fields, or structure pointers that must be set prior to enabling a feature in a different field). Thus, Shade must err on the side of caution wherever invalid field combinations could lead to security holes or undefined behavior: it categorically blocks the hypervisor from writing to such fields and forces the features they control to be disabled or utilized in a hardcoded fashion preconfigured by the SVA runtime.

While this achieved Shade's goal of ensuring security for the host system while supporting basic VMX functionality, it locks the hypervisor out of performance- and functionality-critical VMX features such as extended paging, interrupt controller (local APIC) virtualization, and MSR[3] and I/O port virtualization. Shade provided higher-level intrinsics supporting extended paging by extending SVA's existing support for vetting host page table updates [34] but did not attempt to design vISA support for APIC, MSR, or I/O port virtualization.

Ombro extends the design of the vISA to support APIC, MSR, and I/O port virtualization as first-class idioms; Table 1 summarizes the new intrinsics. As all three of these features entail both multi-VMCS-field dependencies and substructures linked into the VMCS (and APIC virtualization interacts with MSR virtualization), we address them via similar techniques.

MSR and I/O virtualization are the more straightforward of these to support. By default, VMX guests are blocked from accessing MSRs or performing port I/O (i.e., `rd`/`wrmsr` and port I/O instructions cause VM exits) [30]. When MSR or I/O virtualization is enabled in the VMCS, the processor will selectively allow guests to read or write particular MSRs or I/O ports based on whether their corresponding bits are set in the *MSR* and *I/O bitmaps*, which are substructures linked via host-physical pointers in the VMCS. This hardware design has multiple security implications in an SVA system.

Firstly, as MSRs control privileged processor features (including crucial security features like long mode and page-

[3] *Model-specific registers* (MSRs) are a class of indexed control and information registers used extensively in the x86 ISA to manage privileged processor features [30]. They have widely varying semantic and security implications and represent a substantial portion of the x86 ISA's complexity.

Table 1: APIC, MSR, and I/O Virtualization Intrinsics

Name (Arguments)	Description
`vlapic.enable (paddr virtual_apic_page, paddr apic_access_page)`	Enable APIC virtualization for the active VM using the xAPIC (MMIO) interface
`vlapic.enable_x2apic (paddr virtual_apic_page)`	Enable APIC virtualization for the active VM using the x2APIC (MSR) interface.
`vlapic.disable`	Disable the currently active VM's local APIC (or use exit-based virtualization).
`posted_interrupts_enable (u8 vector, paddr descriptor)`	Enable posted interrupt processing for the currently active VM.
`posted_interrupts_disable`	Disable posted interrupt processing for the currently active VM.
`msr_intercept.{get,set,clear} (int vmid, u32 msr, enum rw)`	Get, set, or clear an MSR intercept for the specified VM.
`io_intercept.{get,set,clear} (int vmid, u16 port)`	Get, set, or clear an I/O port intercept for the specified VM.

level execute permissions [30]), the vISA cannot allow untrusted host software (e.g. the hypervisor or host kernel) to access them arbitrarily. Thus, neither of the `rdmsr` or `wrmsr` instructions are present in the SVA vISA; relevant processor features are managed through higher-level vISA idioms or controlled directly by SVA. A guest VM, however, can execute native (non-vISA) kernel-mode code; its access to MSRs is therefore constrained only by the MSR bitmaps. Therefore, it is necessary for Ombro to constrain the settings of the MSR bitmaps on a per-MSR level.

Secondly, the bitmaps themselves, being VMCS substructures addressed via raw host-physical pointers, would represent a security hole if the hypervisor were allowed to control them directly. A compromised hypervisor could configure the processor to use bitmap addresses corresponding to arbitrary physical memory, including SVA-protected pages (Section 2.2), allowing it to trick SVA into overwriting protected memory (since SVA must write to the bitmaps to ensure guests exit when accessing security-sensitive MSRs) or infer the contents of protected memory based on a guest's behavior.[4]

Ombro addresses these issues by allocating and taking ownership of the MSR and I/O bitmaps itself in protected memory, as it does with the VMCS (Section 2.3). The `msr_intercept.clear` intrinsic (Table 1) checks the provided MSR index against a whitelist of known-safe MSRs that guests can access without compromising Ombro's security policies. `io_intercept.clear` does not need to impose any restrictions under Ombro's threat model, as the only need is to prevent abuse of the bitmap substructure (per-port filtering is available for potential use under other threat models).

APIC virtualization (*APICv*) poses similar challenges but is more complex. Modern processors support both legacy *xAPIC mode* (in which the APIC is controlled via a memory-mapped I/O (MMIO) interface) and the newer *x2APIC mode* (which is controlled via MSRs) [30]. Traditionally, hypervisors would configure VMX to force a VM exit on all APIC accesses, either via extended paging (for the xAPIC MMIO interface) or by configuring the MSR bitmaps to force exits for APIC-related MSRs (for x2APIC). This allows the hypervisor to fully emulate the APIC in software, but is slow.

APICv allows certain common APIC accesses by the guest to be virtualized in hardware without a VM exit. The hypervisor provides a *virtual-APIC page* in memory whose fields stand in for the real APIC's registers when a guest attempts to access them [30]. Guest reads to APIC registers via the MMIO (xAPIC) or MSR (x2APIC) interfaces see the values provided by the hypervisor in the virtual-APIC page. Guest APIC writes are virtualized by hardware without a VM exit in situations involving the task- and processor-priority registers, end-of-interrupt signaling, and self-IPIs; unvirtualizable writes are stored to the virtual-APIC page followed by a VM exit so the hypervisor can handle them conventionally.

Relatedly, *posted-interrupt processing* [30] allows a hypervisor to send interrupts to a guest running on a different processor without forcing that guest to VM exit. When a processor in guest mode receives a (real) inter-processor interrupt to a specified *notification vector*, it will (without exiting) check an in-memory *posted-interrupt descriptor* to see if one or more virtual interrupt records have been deposited within (e.g. by the hypervisor on the sending CPU), and if so, deliver them to the running guest through its virtual APIC.

Both features effectively map pages of host-physical memory into a guest's address space with (limited) write access, posing a clear security risk under our threat model, as this could be used to defeat SVA-protected memory (e.g. the return address stacks described in Section 7). Unlike with MSR and I/O port virtualization, it is not convenient to simply have Ombro take ownership of the virtual-APIC page and posted-interrupt descriptor in protected memory, as the hypervisor needs to frequently read and write to them for normal operation. Thus, to allow the hypervisor to safely control these pages, our APICv intrinsics (Table 1) check and record the VMCS's references to them in SVA's memory metadata tables (which track the usage of every 4-kB physical memory frame; see Sections 2.2 and 6.1) as if they were page mappings accessible to the hypervisor. The hypervisor is thus only allowed to use non-sensitive pages it "owns" as APICv VMCS substructures. Additionally, Ombro ensures that the vector used for posted-interrupt notifications does not overlap with any intercepted by SVA.

5.2 Guest Context Switching Optimizations

The x86 platform supports context switching of floating-point unit (FPU) state using the `XSAVE` and `XRSTOR` instructions [30]. These instructions save or load a processor's entire FPU state, as well as that of several non-FPU features such as vector and memory protection extensions, as a several-kB monolithic data structure; system software is thus architected to perform these slow operations as infrequently as possible.

[4] While Ombro's threat model (Section 3) does not require confidentiality of SVA-protected memory, other SVA-based systems such as Virtual Ghost [14] and Shade [34] do, making this a relevant design consideration.

Xen refrains from using the FPU during its own execution, leaving guest state untouched and allowing Xen to only perform `XSAVE/XRSTOR` when context switching from one guest vCPU to another.

Shade [34] took the straightforward but inefficient approach of context switching FPU state on every VM entry/exit. This provided a clean abstraction wherein no guest state is ever active on the processor in host mode or vice versa, but porting Xen to the SVA vISA for Ombro showed this to be a flawed design, yielding more than 200% overhead over baseline (non-SVA) Xen in our no-op hypercall (VM entry/exit) microbenchmark and unacceptably high overheads of up to 60% on VM-exit-heavy guest workloads (Section 10).

Ombro eliminates this source of overhead by extending the SVA vISA's existing *thread* abstraction, which can be context switched independently of processor privilege level transitions, to include guest VM (VMX non-root mode) state in addition to userspace host process (Ring 3) state as in prior SVA work [14, 16]. This better matches how real-world hypervisors like Xen model context switching and allows FPU state to be switched on vCPU context switches instead of requiring it on every VM entry/exit. Because slow-switching elements of the guest's state (the FPU, some MSRs, etc.) are thus active even while the system is in host mode, this provides a slightly less flexible programming model to the hypervisor than in Shade, but these limitations are non-issues in practice and can be worked around: Xen already avoids using the FPU in hypervisor context; an OS-integrated hypervisor would simply allocate separate SVA threads for guest VMs and host processes; and a hypervisor could free up the FPU for itself by context switching to a dummy SVA thread not associated with a guest VM or host process.

5.3 VMCS Management Optimizations

Ombro makes two additional improvements to the SVA vISA to eliminate overheads related to VMCS management induced in Xen by the Shade [34] design. The foremost of these is that, unlike the native ISA, Shade only permits a single VMCS to be loaded on a CPU at a time. The native ISA only permits one VMCS to be *active* at a time, but others need not be unloaded to load a new one [30], allowing them to remain cached by the hardware for future context switches. This distinction is crucial to performance; we found via an informal benchmark that modifying Xen to explicitly clear (flush) the outgoing VMCS in every context switch induces unacceptably high (over 4x) runtime overhead on guests when the machine's physical CPUs are oversubscribed (i.e., when vCPU context switches are frequent). Ombro addresses this limitation by loosening the vISA to allow multiple loaded VMCSes to coexist, relying on the mutex in the SVA thread structure (Sections 5.2 and 6.1) to ensure a VMCS cannot be loaded on multiple CPUs at once (which is undefined behavior in the native ISA [30]).

The second improvement changes VMCS initialization (the `allocvm` intrinsic) to provide benign defaults for all security-sensitive VMCS and guest state fields rather than requiring the hypervisor to specify them up-front. While VMCS construction is not performance-critical for Xen, it occurs early in vCPU creation before Xen has determined most of the guest's initial state, making it awkward to port Xen to use Shade's interface. Ombro's interface is more general and agnostic to hypervisor design choices.

6 SMP Support in SVA

Ombro adds symmetric multiprocessing (SMP) support to SVA that all previous SVA systems [13–16, 21, 22, 34] lacked. This required several changes to the internal design of the SVA runtime library to make it thread-safe and to add support for multi-propcessor TLB coherency. However, we made *no* changes to the outward-facing SVA-OS virtual instruction set; the original design [15–17] proved general enough to be applicable to both uni- and multi-processor systems.

6.1 Thread Safety and Reference Counting

SVA maintains several data structures in its internal protected memory (Section 2.2) to track system state that it maintains on behalf of the hypervisor/OS kernel and to ensure that its intrinsics are not used to configure the system in a way that could undermine other security protections. These include thread structures used for context switching host processes [16] and guest vCPUs (Section 5.2) and a table tracking typed references to each physical memory frame to prevent host-kernel-mode software from using its control over the MMU to evade SVA's memory protections or code integrity enforcement [15].

As these structures must be thread-safe in a multi-processor system, Ombro adds locking to SVA's thread structures and to each entry in the frame usage table. Intrinsic calls attempting to load or save a thread or to change a frame's usage type must obtain the relevant lock to prevent races. Incrementing/decrementing a frame's reference count in the table when a page mapping is updated is a lock-free operation utilizing an atomic compare-exchange loop to perform the update while checking for integer overflow/underflow.

Additionally, Ombro expands the frame reference counts themselves to separately count read-only and writable page mappings to each frame. Prior SVA work [15, 22, 34] did not allow system software to create *any* mappings to SVA-protected frames even when they only require tamper-protection and not confidentiality (e.g. kernel code or page tables). This required ad-hoc (and OS-specific) handling in SVA of special cases like the kernel's direct map and read access to page tables. Ombro allows these to be handled through ordinary intrinsic calls, making SVA more system-agnostic and allowing Xen to continue supporting non-VMX PV guests under Ombro.

6.2 TLB Shootdowns

In an SVA system, physical memory frames used for security-sensitive purposes such as page-table pages, host-kernel-mode code, or SFI-protected memory (e.g. return address stacks in Ombro—see Section 7) are tracked by SVA so it can prevent system software from mapping them into the virtual address space with inappropriate permissions or outside the SFI-protected region [14, 15, 22]. To prevent use-after-free attacks based on stale TLB entries, SVA flushes the TLB whenever a frame's usage type changes and one or both of the types involved are security-sensitive (because x86 does not support selective TLB flushes based on physical addresses [30], a full TLB flush must be used).

On multiprocessor systems, this TLB flush must include *all* processors, necessitating *TLB shootdowns*. Ombro implements this by broadcasting an inter-processor interrupt (IPI) [30] to all processors at a reserved interrupt vector, which is received by an SVA handler that performs each local TLB flush. The processor initiating the shootdown will not release its lock on the frame's usage type until all other processors have acknowledged completion of the flush, ensuring that software cannot create a conflicting mapping based on the new type that would violate security policy.

7 Return Address Integrity

To defend against advanced control-flow hijacking attacks as described in our threat model (Section 3), Ombro must protect the integrity of return and return-from-interrupt addresses in Xen. We address this by using the vISA primitives provided by SVA to implement a *split stack* in Xen, where return addresses are stored on one stack (called the *control stack*) while local variables are stored on a separate stack (called the *data stack*). The control stack is protected against tampering using SVA's kernel-mode memory protection mechanism (Section 2.2), while memory writes utilizing dynamic pointers or offsets that could be controlled by an attacker are only permitted to access the data stack in ordinary (unprotected) Xen memory.

7.1 Security Guarantees

Ombro ensures return address integrity (all functions return control flow to their dynamic callers) by enforcing the following invariants on the hypervisor at runtime:

Invariant 1. *Function calls always save the return address on the control stack, or do not save any return address (e.g. tail calls).*

Invariant 2. *Returns will always retrieve the return address from the correct location on the control stack, i.e. into which the return address was saved by the matching dynamic caller.*

Invariant 3. *Control stacks cannot be corrupted by any code outside of trusted SVA-OS intrinsics, even when memory safety errors are exploited.*

Invariant 4. *System software cannot use an SVA-OS intrinsic to tamper with a control stack's contents or a control stack pointer on its behalf.*

7.2 Enforcement Design

In a split stack design, the control and data stack are tracked by separate stack pointers that can be incremented and decremented independently [10, 59]. In Ombro, we use the native x86 stack pointer register, `RSP`, for the control stack, so that call and return instructions naturally use the control stack for return addresses. This maintains Invariant 1 and contributes to Invariant 2. The data stack is tracked by a free general-purpose register reserved from the callee-saved set (`R15` in our prototype). The compiler is modified accordingly to facilitate this; function prologues and epilogues create and destroy stack frames using the data stack pointer, leaving the control stack pointer to be adjusted only by the return-address pushes and pops performed by call and return instructions.

Call and return instructions are the only ones permitted to modify the control stack pointer `RSP` outside of SVA-OS intrinsics. They always respectively decrement or increment `RSP` by exactly 8 bytes (the Ombro compiler will not emit returns that pop additional values off the stack), ensuring that only the relevant return address is affected. No other data is stored on the control stack besides the single return address pushed by each call and popped by the corresponding return; the data stack is used for local variables, argument passing, and callee-saved registers. As forward-edge CFI prevents functions being entered except through a call (non-tail-call jumps can only target another location within the current function), calls and returns are guaranteed to occur in correctly nested (matching) order. Hence Invariant 2 is ensured.

Because calls and returns occur in nested order, underflow of the control stack pointer cannot occur. Overflow is addressed by placing a guard page at the end of each control stack; guard pages are marked as invalid in the page tables, ensuring that any attempt to read or write beyond the space allocated for a control stack will be intercepted and prevented by an SVA fault handler.

Ombro instruments all host-Ring-0 code outside of SVA-OS intrinsic implementations with SFI checks on memory stores (Section 2.2), ensuring that any attempts to write to a protected virtual address region will be caught and prevented. SVA's enforcement of code segment integrity (Section 2.1) in conjunction with forward-edge CFI [13, 22] ensures that these SFI checks cannot be bypassed even in the presence of memory safety errors. The control stack is allocated within the SFI-protected virtual address region by SVA on Xen's behalf. Only call and return instructions are exempted from SFI checks (so that they can access the control stack as intended); these are generated by the compiler such that they always access the stack using a predictable static offset from `RSP`, so they cannot be used to access any other location in service of an exploit. Hence Invariant 3 is ensured.

The SVA vISA provides no means for system software to set or adjust the host-Ring-0 control stack pointer `RSP`, or to write to any location on a control stack, except pushing/popping from it via calls and returns. SVA allocates a control stack for each CPU (Xen allocates hypervisor stacks on a per-physical-CPU basis) and points `RSP` to it as a side effect of SVA's boot-time per-CPU initialization; thereafter, SVA maintains the integrity of `RSP` across all intrinsic calls, context switches, and VM entry/exit. Per our threat model (Section 3), the hypervisor is considered benign prior to exploitation by a memory safety error; SVA initialization occurs during early boot before significant attack surfaces become available (there is no network yet, nor have any guest VMs been started, including the *dom0* control domain). The SVA-OS intrinsic implementations themselves are part of the trusted computing base and thus assumed to correctly implement the vISA, ensuring that Invariant 4 is upheld after initialization.

8 Implementation

The prototype we built to evaluate Ombro is based on source code for the SVA-OS runtime support library (see Section 2.1, Figure 1, and Section 4) inherited from the Shade [34] project and other previous SVA work [14–16, 22]. We significantly modernized and refactored the codebase to address limitations of prior work, which included adding multi-processor support and overhauling the page reference tracking system to be more flexible (Section 6). Overall, we improved the code to be substantially less fragile and more maintainable, and generalized aspects of the code that were specific to using SVA with OS kernels (and FreeBSD in particular), such that it could support a bare-metal hypervisor like Xen while retaining support for OS kernels (including those with integrated hypervisors). We upgraded SVA to be based on the LLVM 10.0.0 compiler [2] instead of LLVM 3.1 as in Shade [34] and prior work. In parallel, we ported the x86-64 implementation of Xen 4.12 [3] to target Ombro's version of the SVA vISA, linking it at compile time with the SVA-OS runtime support library (as in prior SVA work).

Our software trusted computing base (TCB) relative to our threat model (Section 3) consists of the SVA runtime library (11,453 lines of code (LOC)), our CFI and SFI passes added to the LLVM compiler (792 LOC), and non-pass modifications to LLVM implementing our split stack transformation (497 LOC added/changed), totaling 12,742 LOC.[5]

Porting Xen 4.12 to the SVA vISA and supporting Ombro's split stack transformation entailed adding/changing $\leq$3,389 LOC[6] (out of 313,377 total in Xen—about 1%), mostly in low-level code dealing with page tables, VM entry/exit, VMCSes, and system boot. By comparison, prior work's port of Linux 2.4.22 to SVA [16] modified 5,066 LOC out of 632,469 total (0.8%). This shows that the difficulty of an SVA port scales roughly but not exactly with the scope of the system.

As Section 4 describes, Ombro's security design assumes that all domains are used only in VMX-accelerated modes (HVM and PVH), but our prototype retains partial support for classic PV mode. Although Xen 4.12 supports a PVH *dom0*, we found it useful to use a PV *dom0* in order to have a working, debuggable environment while porting VMX-related components to SVA, as the *dom0* is responsible for controlling Xen and providing hardware drivers. We did not attempt to fully port Xen's PV code to SVA, leaving some native assembly and unsafe (to our threat model, not Xen's) workarounds in place, though this could be completed with a little extra work and minor enhancements to the SVA vISA. Since we benchmarked (Section 10) within a PVH *domU*, we do not expect significant performance differences between a PV and PVH *dom0*, especially since we used the same configuration for both Ombro and baseline Xen.

Further Implementation Experience Discussion. Some readers may be interested in further discussion of our experience porting Xen to the SVA vISA and how that experience, as well as our incremental performance evaluations throughout, fed back into our design process and motivated specific design changes. For reasons of space, that discussion is deferred to Appendix B.

9 Security Analysis

Prior work [7, 8, 18, 25, 27] has shown that defenses relying on CFI [4] to mitigate memory safety attacks must *prevent* the corruption of return addresses in order to repel advanced control-flow hijacking attacks, as static determination of the call sites to which control flow may return leaves sufficient loopholes for an attacker to perform arbitrary computation [8]. To this end, Ombro implements a split control/data stack in Xen (Section 7) and uses SVA's SFI-based kernel-mode memory protection facility (Section 2.2) to protect the control stack (and hence return addresses) from tampering.

Zhou et al. [58] proposed a framework for evaluating the attack surface exposed by a security policy providing return-address protection, drawing from Göktaş et al.'s [27] taxonomy classifying the different types of control-flow "gadgets" that can be used by an attacker to assemble a code-reuse attack in the presence of CFI. *Call-site* (CS) gadgets begin at a return point following a call instruction, and *Entry-point* (EP) gadgets begin at the entry point to a function. Orthogonally, they classify the methods by which gadgets can be linked together to form a "chain" useful for computation: by corrupting return addresses on the stack (*return-oriented programming*), indirect jump targets (*jump-oriented programming*), or function pointers (*call-oriented programming*). Depending on the CFI policy in effect and the availability of gadgets in the target

[5] We counted non-comment/whitespace lines using `sloccount` [52] for the SVA library and passes and manually from `git diff`s for the rest.

[6] Most changes are gated behind `#ifdef`s, leaving the original code in place; we took the difference between vanilla Xen and the Ombro port using `sloccount` and added twice the number of removed lines from `git diff` as an upper bound on the undercount.

program, it may be possible and/or necessary to mix different gadget types and chaining methods to achieve a practical chain. Pure call-oriented programming, in which only function pointers are manipulated to link gadgets, turns out to be difficult or impractical to achieve in many cases, particularly in scenarios where it is not possible for the attacker to repeatedly exploit memory safety errors; return- or jump-oriented gadget linking must typically be used to perform initial setup before the attacker can pivot to a call-oriented chain [7, 27].

Because Ombro's split stack design prevents any corruption of Xen's control stack (which includes return as well as return-from-interrupt addresses), return-oriented gadget linking is categorically precluded. Additionally, the SVA vISA does not include LLVM's indirect jump (`indirectbr`) instruction and does not need it to support Xen, the Linux kernel [16], or the FreeBSD kernel [13, 14]. Thus, jump-oriented gadget linking is precluded. Cumulatively, this severely limits the ability of an attacker to assemble a useful gadget chain, particularly in single-exploit scenarios, as only call-oriented chaining is possible. That, in turn, is further limited by Ombro's forward-edge CFI protection.

Ombro's label-based forward-edge CFI scheme is based on that of KCoFI [13]; it is relatively coarse-grained, allowing indirect calls to target any valid function entry point (but nowhere else). Since the protected control stack already prevents return-oriented gadget chaining, and the prohibition of `indirectbr` prevents jump-oriented chaining, the net effect of adding this forward-edge CFI enforcement is to limit an attacker to a restricted form of pure call-oriented programming: only entry-point (EP) gadgets can be used and they can only be chained together via corrupted function pointers.

Ombro may be susceptible to the pure-call-oriented "Control Jujutsu" attack described by Evans et al. [25], who showed that, in popular programs, common function pointer coding patterns make it possible to assemble purely call-oriented chains of EP gadgets even when return addresses are fully protected and strong static analysis is used to limit forward-edge control flow transfers to those intended by normal program operation. Specifically, they noticed that this was possible in Apache and Nginx due to extensive use of function pointers to provide C++-style runtime polymorphism in C, which exacerbates the lack of context-sensitivity in static (label-based) CFI. We observe that Xen frequently utilizes similar code patterns, using function pointers to delegate at runtime to method implementations specific to particular virtualization modes, hardware capabilities, etc. Hence, similar attacks might be possible against Xen protected by Ombro. It is unclear, however, whether such attacks can work in practice against a bare-metal hypervisor like Xen. Evans et al. relied on the proliferation of arbitrary-code-execution system calls such as `system()` and `execve()` to invoke a shell rather than attempting to achieve arbitrary computation through code reuse alone; these elements do not exist in Xen, and similarly desirable functionality from an attacker's perspective (e.g., giving the attacker's domain *dom0* privileges or mapping a victim domain's memory into the attacker's) is not necessarily invoked from as many places in the codebase.

Although our prototype implementation only restricts indirect calls to calling any valid function, it would be straightforward to extend our label-based CFI approach to utilize a more precise control flow graph based on more advanced static analysis. Because Xen is a monolithic executable and does not support run-time module loading, whole-program analysis could be used to identify functions that are never address-taken (i.e. never indirectly called) and refrain from emitting CFI labels for them, eliminating them as viable targets for gadget chaining. While such functions might be reachable "downstream" of other functions that *are* called indirectly, this could make return-to-library [49] and similar short-chain code-reuse attacks more difficult, furthering the goal of making it difficult to reach functionality of ultimate interest to an attacker. (Most indirectly-called functions in Xen are focused on low-level platform-specific and scheduling-related functionality, and do not interact directly with code manipulating sensitive high-level fields such as access controls.)

Since it is built using SVA, Ombro prevents many attacks (such as code injection) even if an attacker successfully diverts control flow. Because SVA prevents tampering with host-kernel-mode code pages (i.e. Xen or SVA code) to ensure that all privileged code is compiled with the trusted vISA translator (Section 2.1), it is impossible for an attacker to pivot from code reuse to a more flexible code-injection attack, which is typically the goal of code-reuse payloads in practice [27, 35]; attackers must perform *all* malicious computation via code reuse. While ROP "compilers" do exist (e.g. [48]), they are incapable of compiling complex high-level payloads, particularly for the restrictive code-reuse modalities necessary to defeat Ombro's CFI. This effectively limits attackers to non-control data and data-oriented programming (DOP) attacks, which are outside our threat model's scope (Section 3). DOP could also face practical headwinds similar to Control Jujutsu.

10 Performance Evaluation

To evaluate Ombro's performance, we selected a portfolio of real-world application macrobenchmarks (Section 10.1) that we believe are reflective of typical cloud workloads. We used the Phoronix Test Suite's pts/kernel suite [40, 44] as a starting point, with the following adjustments:

- We excluded benchmarks that failed to compile or run on our test system running unmodified Xen (see system details below). We used a more recent version (2.4.48) of the Apache benchmark rather than exclude it due to its real-world importance and use by related work [41].

- Because the full pts/kernel suite takes days to run and includes numerous configurations of the same applications with minor differences, we ran a single configuration of

each application, selecting the longest-running one that completed in less than five minutes. Our full suite runs in approximately three hours. Phoronix runs each benchmark at least three times and until the standard deviation of all runs is less than 3.5% or a benchmark-specific cutoff threshold is met; the result reported for each benchmark is the (arithmetic) mean of these runs [45].

- We chose to add a Memcached benchmark (v1.6.9, also from Phoronix) because it is used by related work [41] and represents a stress test for Ombro's overheads. We used the "Get" configuration with four connections.

We also developed and ran microbenchmarks (Section 10.2) for hypercall latency (VM entry/exit), EPT fault handling, and inter-processor interrupts (IPIs); our selection of microbenchmarks parallels related work [41].

For all of our experiments, we used a Dell Precision 7820 workstation with an Intel Xeon Silver 4208 (Cascade Lake) CPU (8 cores/16 threads at 2.10 GHz with 11 MB L3 cache) [11], 32 GB 2666 MHz DDR4 memory, a 256 GB M.2 NVMe SSD, and a 2 TB SATA 7200 RPM hard drive. All disk I/O for the experiments used the SSD (the hard drive was unused); the *dom0* (control VM) had direct access to the SSD and the *domU* (guest VM)'s virtual disk was directly backed by a physical partition (not a file-based disk image).

For our baseline, we ran unmodified ("vanilla") Xen 4.12 with an Arch Linux (kernel 5.15.12-arch1-1, packages updated 2022-01-04[7]) *dom0* running in PV (traditional paravirtualization) mode. The benchmarks ran within a *domU* running the same Linux distribution in PVH (VMX-accelerated with paravirtual optimizations) mode. The *domU* was allocated 16 vCPUs and 24 GB of RAM. We used GCC 11.1.0 to compile the benchmark applications. We disabled Spectre [37] mitigations such as IBRS [30] for all benchmark runs to provide a fair comparison, since our prototype did not implement them (for reasons of time). We likewise enabled eager FPU saving in vanilla Xen as Ombro saves FPU state on every vCPU context switch (Section 5.2).

We evaluated Ombro using the implementation described in Section 8, i.e., Xen 4.12 ported to the SVA vISA and compiled with CFI, SFI, and split-stack transformations. All other configurations were the same as for the baseline.

10.1 Macrobenchmark Results

Table 2 summarizes our macrobenchmark results comparing baseline unmodified Xen (labeled "vanilla") with Ombro. As Table 2 shows, Ombro incurs no detectable overheads (differences are within or close to noise margins) on most benchmarks, even though we selected a predominantly I/O-intensive (i.e. challenging to virtualize) benchmark set. Several exceptions, particularly Memcached, RocksDB, and LevelDB, are discussed further in Section 10.3 and Appendix A.1.

[7] Arch Linux is a "rolling" distribution without versioned releases, so the package date stands in lieu of a version number.

Table 2: Macrobenchmark Results

Benchmark	**Units**	**Vanilla**		**Ombro**	
	(↑↓ is better)	**Result**	**Std. Dev.**	**Ovhd.**	**Std. Dev.**
PostgreSQL	TPS ↑	4266.496	26.7%	-17.84%	18.5%
PostgreSQL	ms (avg. lat.) ↓	61.644	21.0%	-16.72%	18.9%
MBW	MiB/s ↑	6694.581	0.3%	-2.88%	0.1%
BenchmarkMutex	ns ↓	39.967	0.9%	-0.67%	0.0%
PostMark	TPS ↑	4335.000	1.0%	-0.58%	1.0%
pmbench	μs ↓	0.113	4.5%	-0.55%	4.2%
OSBench (Create Files)	μs/event ↓	24.270	0.3%	-0.07%	0.8%
ctx_clock	clocks ↓	240.000	0.0%	0.00%	0.0%
OpenSSL (RSA 4096)	signs/s ↑	1516.867	0.3%	0.05%	0.2%
StressNG (RdRand)	bogo ops/s ↑	250298.447	0.0%	0.06%	0.0%
Apache	req/s	212698.913	0.4%	0.34%	0.1%
t-test1	s ↓	23.587	0.9%	0.75%	0.6%
iPerf (TCP)	Mbits/s ↑	28029.667	2.3%	0.75%	1.1%
SQLite	s ↓	83.164	0.8%	1.06%	0.7%
Hackbench	s ↓	140.056	0.8%	4.53%	1.8%
Schbench	μs ↓	18762.667	0.7%	4.59%	8.0%
IPC (TCP Socket)	messages/s ↑	473697.400	13.3%	4.63%	12.1%
LevelDB	μs/op ↓	428.367	0.1%	12.62%	0.3%
RocksDB	ops/s ↑	34557.667	0.1%	12.82%	0.3%
Memcached	ops/s ↑	46960.733	0.2%	21.81%	2.1%
Geometric mean				**0.87%**	

Table 3: Microbenchmark Results (TSC cycles)

Benchmark	**Vanilla**		**Ombro**		
	Cycles	**Std. Dev.**	**Cycles**	**Std. Dev.**	**Ovhd.**
No-op hypercall	750	18.5%	1465	14.2%	95%
EPT fault	4180	10.0%	5693	8.88%	36%
vCPU self-IPI	1731	12.9%	2353	14.6%	36%
IPI (vCPU→vCPU)	2966	14.6%	3531	14.4%	19%

We also ran our benchmarks with Ombro's split stack, CFI, and SFI transforms disabled to determine if Ombro's security instrumentation contributes significant performance overhead. Our results show no discernible difference, i.e., nearly all of Ombro's overhead comes from the vISA itself, not the instrumentation. Appendix A.2 contains detailed results.

10.2 Microbenchmarks

Table 3 summarizes our microbenchmark results for key hypervisor operations. We ran 2^{28} timed iterations of each benchmark (reporting the arithmetic mean) after 2^{12} untimed warmup iterations. Results are in hardware timestamp counter (TSC) cycles measured using the `rdtsc` instruction (with Xen's `rdtsc` interception disabled) before and after the operation (or sending/receiving on the respective CPUs for IPIs; our processor synchronizes the TSC across cores [30]).

Each microbenchmark includes a VM entry/exit cycle, which is minimally exercised by the no-op hypercall benchmark. Ombro's base hypercall overhead is substantially greater than the overhead of more complex operations, which are themselves greater than our macrobenchmark overheads in Section 10.1. This indicates that VM entry/exit is the primary source of Ombro's overhead. Since hardware acceleration is designed to make VM exits relatively rare, Ombro's overhead only substantially impacts difficult-to-virtualize workloads that incur frequent VM exits.

10.3 Overhead Sources and Their Remedies

Prior work [41] illustrated that Memcached's performance is highly sensitive to changes in VM entry/exit latency be-

cause it is frequently bottlenecked by the need to send interprocessor interrupts (IPIs) between guest vCPUs, e.g. by Linux's implementation of the `futex()` system call. IPIs are efficient on a "bare-metal" (unvirtualized) system but expensive to virtualize because hypervisors cannot safely expose the interrupt controller (local APIC) to guests, requiring a VM exit to virtualize the APIC in software whenever an IPI is to be sent. Although VMX currently provides exit-free hardware-virtualized *delivery* of virtual interrupts (Section 5.1), the *sending* of them is not yet virtualized, making this a major pain point for virtualizing popular applications like Memcached. Besides Memcached, our RocksDB and LevelDB benchmarks are of a similar nature (multi-threaded in-memory databases) and exhibit the same sensitivity.

To confirm our hypothesis that Ombro's overheads come almost exclusively from VM entry/exit overheads (Section 10.2) and that this is responsible for our three macrobenchmark outliers, we conducted an informal experiment where we modified vanilla Xen to add artificial (busy-wait) overhead after each VM exit. With this modification, vanilla Xen's macrobenchmark results closely matched those of Ombro (the same benchmarks showed similar slowdowns).

We conclude, therefore, that Ombro's overheads on these outliers are not of great concern, as the IPI virtualization problem is shared with vanilla Xen (and x86 virtualization in general). We explore this further in Appendix A.1, where we compare vanilla Xen with a bare-metal (unvirtualized) system; we observe that baseline Xen's overheads on Memcached, RocksDB, and LevelDB (and others) far outweigh the additional impact of Ombro on Xen. We note also that Intel has announced plans to introduce hardware-accelerated *IPI virtualization* in future processors [12] to address this issue; we expect this will eliminate Ombro's overheads on these workloads as it will eliminate the underlying VM exits.

As Appendix A.2 details, none of Ombro's effective overhead is attributable to its compile-time security instrumentation (split stack, CFI, and SFI). This bodes well for the prospects of using the SVA approach and infrastructure to implement stronger security hardening, such as full memory safety [20, 43, 57], on hypervisors, as hypervisor execution evidently does not represent a sufficient fraction of system runtime to make instrumentation on it costly in absolute terms.

11 Related Work

Compiler-based virtual machines decouple the instruction set used to express computation from the instruction set implemented by the hardware. Ombro builds directly upon prior work with the Low Level Virtual Architecture [5, 17] and Secure Virtual Architecture [16], described in Section 2.

Previous work has enforced CFI and/or return address integrity on systems code, but none have enforced return address integrity for hypervisors. KCoFI [13] and KCFI [26] enforce control flow integrity on OS kernel code but do not provide return address integrity. Silhouette [58] provides CFI and a protected shadow stack for application code running in privileged mode on embedded systems and inspired our attack surface analysis (Section 9). Kage [24] uses compiler-based techniques similar to Ombro's to provide CFI and a protected shadow stack for an embedded real-time OS while splitting the kernel into trusted and untrusted layers. IskiOS [28] provides a protected shadow stack for the Linux kernel by utilizing Intel's Memory Protection Keys feature [30] but does not enforce it on hypervisor code. Ombro could incorporate IskiOS's technique in lieu of SVA's MPX-based SFI enforcement option if Intel deprecates MPX as planned [31].

HyperSafe [51] enforces control flow integrity on hypervisor code and controls how the hypervisor configures the MMU to prevent attackers from corrupting page tables, hypervisor code pages, or CFI labels. Unlike Ombro, it does not provide return address integrity and is therefore susceptible to advanced ROP attacks [7, 8, 18, 27]. The HyperSafe authors designed and evaluated a shadow stack variant in conjunction with their CFI scheme but found it to have high overhead (over 300%) due to its reliance on the x86 platform's `WP` bit as an isolation mechanism [51]. In contrast, Ombro's use of SFI for kernel-mode isolation provides efficient control stack protection. HyperSafe also does not appear to constrain VMX features (e.g., by protecting the VMCS) whereas Ombro does.

Whole-VM trusted execution environments (TEEs) that protect guest VMs from compromised hypervisors, such as CloudVisor [41, 56], H-SVM [32, 33], and HyperCoffer [55], are an orthogonal approach to hardening the hypervisor itself against attack. Ombro's SVA-based VISC approach could readily lend itself to implementation of a whole-VM TEE using SFI and vISA restrictions in lieu of the hardware-based isolation mechanisms used by prior work, similar to Virtual Ghost's [14] and Apparition's [22] approach for OS kernels. In such a system, a whole-VM TEE could be *combined* with Ombro-style hypervisor hardening for additional defense-in-depth with (we believe) minimal cumulative overhead.

12 Future Work and Conclusions

Several interesting directions for future work exist. We can explore additional security policies that SVA-based systems could enforce on hypervisor code, such as code pointer integrity [38] and memory safety [20, 43, 57]. We can also explore replacing hardware-enforced security enforcement and isolation with compiler instrumentation techniques. For example, we could explore whether compiler-based enforcement could allow us to create hypervisors with isolated components as previous work [53] does using hardware isolation features.

In conclusion, we have expanded the SVA vISA and ported the Xen hypervisor to it, have used the vISA to implement the first protected shadow (split) stack for a hypervisor, and have demonstrated its efficiency on real-world benchmarks.

We thank the anonymous reviewers and shepherd for their helpful feedback. This work was supported by NSF grant CNS-1629770 and ONR award N00014-17-1-2996.

References

[1] The Rust Programmng Language. https://www.rust-lang.org [Online; accessed 2022-06-08].

[2] The LLVM Compiler Infrastructure Project. [Online; accessed 2022-01-12].

[3] Xen Project. https://xenproject.org [Online; accessed 2021-08-07].

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

[5] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke. LLVA: A Low-Level Virtual Instruction Set Architecture. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-36, pages 205–216, San Diego, CA, 2003. IEEE Computer Society.

[6] Daniel P. Bovet and Marco Cesati. *Understanding the LINUX Kernel*. O'Reilly, Sebastopol, CA, 2^{nd} edition, 2002.

[7] Nicholas Carlini and David Wagner. ROP Is Still Dangerous: Breaking Modern Defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, San Diego, CA, August 2014. USENIX Association.

[8] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-flow Bending: On the Effectiveness of Control-flow Integrity. In *Proceedings of the 24th USENIX Security Symposium (SEC)*, pages 161–176, Washington, D.C., 2015.

[9] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the 14th USENIX Security Symposium (SEC)*, pages 12–12, Baltimore, MD, 2005.

[10] Clang Documentation. SafeStack. https://clang.llvm.org/docs/SafeStack.html [Online; accessed 18-June-2021].

[11] Intel Corporation. Intel Xeon Silver 4208 Processor. https://ark.intel.com/content/www/us/en/ark/products/193390/intel-xeon-silver-4208-processor-11m-cache-2-10-ghz.html. [Online; accessed 2021-08-11].

[12] Intel Corporation. Intel Architecture Instruction Set Features and Future Extensions Programming Reference. https://software.intel.com/content/www/us/en/develop/download/intel-architecture-instruction-set-extensions-programming-reference.html, May 2021. [Downloaded 2022-01-12].

[13] John Criswell, Nathan Dautenhahn, and Vikram Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, pages 292–307, San Jose, CA, May 2014.

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

[15] John Criswell, Nicolas Geoffray, and Vikram Adve. Memory Safety for Low-Level Software/Hardware Interactions. In *Proceedings of the 18th USENIX Security Symposium*, Security'09, pages 83–100, 2009.

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

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

[18] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, San Diego, CA, August 2014. USENIX Association.

[19] Hoss Firooznia (Universitato de Roĉestro Esperantistoj). About Esperanto. https://esperanto.lodestone.org/esperanto/en [Online; accessed 2022-06-08].

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

[21] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan Cox, and Sandhya Dwarkadas. Spectres, Virtual Ghosts, and

Hardware Support. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP'18, pages 5:1–5:9, Los Angeles, CA, 2018. ACM.

[22] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L. Cox, and Sandhya Dwarkadas. Shielding Software from Privileged Side-Channel Attacks. In *Proceedings of the 27th USENIX Security Symposium*, Security'18, pages 1441–1458, 2018.

[23] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. pages 164–177, Bolton Landing, NY, USA, October 2003.

[24] Yufei Du, Zhuojia Shen, Komail Dharsee, Jie Zhou, Robert J Walls, and John Criswell. Holistic Control-Flow Protection on Real-Time Embedded Systems with Kage. In *Proceedings of the 31st USENIX Security Symposium*, Security '22. USENIX Association, 2022.

[25] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 901–913, Denver, CO, 2015. ACM.

[26] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-Grained Control-Flow Integrity for Kernel Software. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 179–194, Saarbrücken, Germany, March 2016.

[27] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of Control: Overcoming Control-Flow Integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, pages 575–589, San Jose, CA, May 2014.

[28] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L. Scott. Fast Intra-Kernel Isolation and Security with IskiOS. In *Proceedings of the Twenty Fourth International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '21, 2021.

[29] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 969–986. IEEE, 2016.

[30] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*. May 2018. 325462-067US.

[31] Intel Corp. Introduction to Intel® Memory Protection Extensions, July 2013. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-memory-protection-extensions.html [Online; accessed 2020-11-10].

[32] S. Jin, J. Ahn, J. Seol, S. Cha, J. Huh, and S. Maeng. H-SVM: Hardware-Assisted Secure Virtual Machines under a Vulnerable Hypervisor. *IEEE Transactions on Computers*, 64(10):2833–2846, Oct 2015.

[33] Seongwook Jin, Jeongseob Ahn, Sanghoon Cha, and Jaehyuk Huh. Architectural Support for Secure Virtualization under a Vulnerable Hypervisor. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 272–283. IEEE, 2011.

[34] Ethan Johnson, Komail Dharsee, and John Criswell. Secure Guest Virtual Machine Support in Apparition. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2019, pages 17–30, New York, NY, USA, 2019. ACM.

[35] Mateusz Jurczyk and Sergei Glazunov. Google Project Zero: In-the-Wild Series: Windows Exploits. https://googleprojectzero.blogspot.com/2021/01/in-wild-series-windows-exploits.html, 2021. [Online; accessed 2021-11-02].

[36] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: The Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, Ottawa, Ontario, Canada, Jun 2007.

[37] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, SP'19, San Francisco, CA, 2019. IEEE.

[38] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 147–163, Berkeley, CA, USA, 2014. USENIX Association.

[39] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO'04, pages 75–86, Palo Alto, CA, 2004. IEEE Computer Society.

[40] Phoronix Media. Phoronix Test Suite. https://www.phoronix-test-suite.com. [Online; accessed 2019-03-11].

[41] Zeyu Mi, Dingji Li, Haibo Chen, Binyu Zang, and Haibing Guan. (Mostly) Exitless VM Protection from Untrusted Hypervisor through Disaggregated Nested Virtualization. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1695–1712. USENIX Association, August 2020.

[42] Microsoft. Introduction to Hyper-V on Windows 10, 2018. https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/ [Online; accessed 2021-08-07].

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

[44] OpenBenchmarking. Common Kernel Benchmarks. https://openbenchmarking.org/suite/pts/kernel. [Online; accessed 2021-08-11].

[45] OpenBenchmarking. Phoronix Test Suite Documentation. https://github.com/phoronix-test-suite/phoronix-test-suite/blob/master/documentation/phoronix-test-suite.md. [Online; accessed 2022-01-12].

[46] Oracle Corporation. VirtualBox. https://www.virtualbox.org [Online; accessed 2022-06-08].

[47] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information Systems Security (TISSEC)*, 15(1):2:1–2:34, March 2012.

[48] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit Hardening Made Easy. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 25–25, Berkeley, CA, USA, 2011. USENIX Association.

[49] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the Expressiveness of Return-into-libc Attacks. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID)*, pages 121–141, Menlo Park, CA, 2011.

[50] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, SOSP'93, pages 203–216, Asheville, NC, 1993. ACM.

[51] Z. Wang and X. Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P)*, pages 380–395, May 2010.

[52] David A. Wheeler. SLOCCount. http://www.dwheeler.com/sloccount/ [Online; accessed 2022-06-08].

[53] Dan Williams, Yaohui Hu, Umesh Deshpande, Piush K. Sinha, Nilton Bila, Kartik Gopalan, and Hani Jamjoom. Enabling Efficient Hypervisor-as-a-Service Clouds with Ephemeral Virtualization. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2016, New York, NY, USA, 2016. ACM.

[54] Xen Project. Understanding the Virtualization Spectrum, 2014. https://wiki.xenproject.org/wiki/Understanding_the_Virtualization_Spectrum [Online; accessed 2021-08-05].

[55] Yubin Xia, Yutao Liu, and Haibo Chen. Architecture Support for Guest-Transparent VM Protection from Untrusted Hypervisor and Physical Attacks. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 246–257. IEEE, 2013.

[56] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-Tenant Cloud with Nested Virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 203–216, New York, NY, USA, 2011. ACM.

[57] Tong Zhang, Dongyoon Lee, and Changhee Jung. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 631–644, 2019.

[58] Jie Zhou, Yufei Du, Zhuojia Shen, Lele Ma, John Criswell, and Robert J. Walls. Silhouette: Efficient Protected Shadow Stacks for Embedded Systems. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1219–1236. USENIX Association, August 2020.

[59] Philipp Zieris and Julian Horsch. A Leak-Resilient Dual Stack Scheme for Backward-Edge Control-Flow Integrity. In *13th ACM Asia Conf. on Computer & Communications Security (ASIACCS)*, Incheon, Republic of Korea, June 2018.

A Additional Benchmarks

In this appendix, we provide results and discussion of additional benchmarks that we conducted in order to shed light on the main results presented in Section 10. We did not include these results in the main body of the paper because they either served to demonstrate existing issues known from prior work that we do not claim as novel results, or presented no statistically significant contrast and thus could be summarized adequately in the main text without detailed results.

A.1 Unmodified Xen vs. Unvirtualized

In Section 10.1, we observed that a few benchmarks from our subset of the Phoronix suite, particularly Memcached, RocksDB, and LevelDB, exhibited non-negligible overheads (unlike most of our macrobenchmarks) under Ombro as compared to baseline ("vanilla") Xen. Through further experimentation (Section 10.2) and reference to existing literature in the field [41], we were able to conclude (Section 10.3) that the primary cause of these particular benchmarks' poor behavior was likely due to the fact that they frequently send inter-processor interrupts (IPIs) to communicate between threads running on different virtual CPUs (vCPUs), which on current Intel processors requires taking a VM exit [12, 30].

As VM exits are expensive (slow) operations for hypervisors to handle due to the substantial amount of processor state that must be saved and loaded during a world switch, hypervisor and hardware design generally tries to make them infrequent as a share of total execution time. Workloads that do not make that possible (such as these problematic benchmarks) can therefore expect to incur substantial overheads simply from being virtualized in the first place.

To illustrate this issue, we re-ran our macrobenchmark suite (Section 10.1) on our test machine in a "bare-metal" (i.e. unvirtualized) configuration and compared the results to that of vanilla Xen, as shown in Table 4. For consistency with our main results, vanilla Xen is retained as the baseline, with the bare-metal results compared to it, yielding negative "overheads", i.e. speedups, for bare metal. As can be seen, the benchmarks that showed non-negligible overheads under Ombro in Section 10.1 all run substantially faster on bare metal than under vanilla Xen, the difference between the two being far greater in magnitude than the difference between Ombro and vanilla Xen. In fact, most of the benchmarks, including the ones to which Ombro added no or negligible overhead over vanilla Xen, show significant gaps between unvirtualized and virtualized execution.

Our methodology for the bare-metal benchmark runs was to boot a copy of the same Linux installation used for the *domU* (unprivileged guest domain) in Xen-based benchmark runs directly from the system bootloader instead of through Xen. As it is the same system except for the addition of a few driver packages needed to support running on physical hardware, this minimizes differences between the configurations, but the comparison is nonetheless imperfect: the bare-metal runs had access to the system's full 32 GB of RAM, whereas the Xen-based runs were limited to 24 GB in the *domU*, as we needed to leave some of the system's memory for Xen and the *dom0*. This, as well as effects such as the difference between virtualized and unvirtualized disk I/O, could potentially amplify the difference measured between vanilla Xen and bare metal. We therefore do not attempt to draw strong quantitative conclusions from the detailed results of this comparison and suffice to note qualitatively that the outliers in our Ombro benchmarks are clearly especially difficult cases for vanilla Xen as well. This is consistent with the conclusions of prior work [41] and the fact that Intel is planning to introduce *IPI virtualization* to future processors so that workloads such as these no longer need to incur frequent VM exits [12].

A.2 Ombro without Instrumentation

As part of our performance evaluation (Section 10), we wished to measure whether Ombro's control flow integrity (CFI), software fault isolation (SFI), and split stack transformations had measurable impacts on performance. This would allow us to separate overheads due to instrumentation from those incurred simply by porting Xen to the SVA virtual instruction set (vISA).

To this end, we set a flag in the compiler that instructed it to not add CFI and SFI checks to the generated code or to perform the split stack transformation when building Xen in the Ombro configuration. This results in a build of Ombro which does not have functional hardening protections beyond vanilla Xen but still uses SVA intrinsics rather than native assembly to perform low-level operations. Thus, it measures any overheads from extra data copying or code indirection entailed by routing through the virtual instructions as well as the overheads of any runtime security checks peformed by the intrinsics themselves.

We ran our macrobenchmark suite on this "no-instrumentation" build of Ombro/Xen and compared it with vanilla Xen, as summarized in Table 5. As in Appendix A.1, we use vanilla Xen as our baseline for consistency with our main results in Section 10.1. The overheads listed for "Ombro without instrumentation" can therefore be compared head-to-head with the "Ombro" numbers in Section 10.1.

As can be seen, the results for Ombro without instrumentation do not exhibit a clear contrast from the Ombro results in Section 10.1 that rises above the noise floor. (In fact, the geometric mean for Ombro without instrumentation shows *higher* overhead than ordinary Ombro, which can be clearly attributed to experimental noise given the high standard deviations on the benchmarks that turned out the most favorably for ordinary Ombro, particularly the PostgreSQL benchmarks.) We therefore conclude that the CFI and SFI instrumentation and the split stack transformation add no measurable overhead to the "core" vISA port. This makes sense in light of our conclusion from Sections 10.2 and 10.3 that Ombro's overheads

are driven primarily by an increase in VM entry/exit latency, not by overheads on Xen's own execution (e.g. scheduling and VM-exit handling such as hardware emulation).

The observation that CFI, SFI, and split-stack enforcement on Xen do not measurably impact overall performance indicates that Ombro's increased VM entry/exit latency is coming from the extra data copying and operations performed by SVA's implementation of VM entry/exit, rather than from Xen itself being slowed down by the compile-time security transformations. As our Ombro benchmarks were conducted with SVA's standard bitmasking SFI implementation selected (Section 2.2) instead of the optional MPX-accelerated SFI implementation from Apparition [22] (which is in principle faster), this leads to a secondary conclusion that optimizing SVA's CFI and SFI instrumentation is neither necessary nor worthwhile for Ombro, even though it has been for past SVA-based systems.

B Implementation and Porting Experience

Section 8 describes our prototype implementation of Ombro (i.e. the SVA compiler and runtime library plus our port of the Xen hypervisor to the SVA vISA) as used in our performance evaluation (Section 10). However, as the construction of this prototype represents a great deal of the work involved in this project, this appendix discusses that experience further for the benefit of interested readers. We discuss practical observations from the experience of building the prototype (B.1) as well as how our observations of the prototype's performance fed back into the design process (B.2).

B.1 Engineering Observations

The process of porting Xen to SVA took two programmers (a PhD student and a research programmer) roughly two years to complete. This includes the substantial infrastructural improvements to SVA described in Section 8; qualitatively, we improved SVA from a minimally functional research prototype (previous SVA systems could not even run large-scale applications like the Apache web server without crashing) to one that, while perhaps not "production-grade", can confidently run a complete Xen system with multiple guests hosting a full spectrum of complex end-user applications. These include a fully working MATE desktop GUI under the Ubuntu-based Linux Mint and a Wayland-based window manager (Sway) under Arch Linux (we used Arch for our benchmarks but exercised both distributions heavily during development); web browsers like Firefox and Chrome; development tools like Clang and GCC; and a full Phoronix suite of kernel-intensive real-world macrobenchmarks (Section 10). Notably, we did not need to exclude *any* benchmarks for lack of compatibility with Ombro, although a few failed to compile on the unmodified baseline system, likely due to the system compiler being too new.

We expect that, particularly with these SVA infrastructure improvements in place, an experienced hypervisor developer could repeat our port of Xen (or port another hypervisor) to SVA/Ombro in substantially less time than we took, as we spent a lot of those two years learning about hypervisor and VMX inner workings and debugging opaque low-level issues.

Once we completed the port of Xen to SVA and had a fully working system, it was relatively straightforward to implement return address protection via a compile-time split stack transformation (Section 7). The split-stack-related changes to the compiler, SVA runtime library, and Xen were relatively small (see LOC numbers in Section 8) and took only about a month to complete, during which we also began the performance evaluation and writing of the paper.

We believe this success highlights the power and flexibility of the SVA approach. SVA provides a well-defined interface for low-level software/hardware interactions along with a robust toolkit of primitives useful for enforcing confidentiality and integrity policies, such as kernel-mode memory protection and mediation of hardware data structures (e.g. page tables and VMCSes). This foundation gives security researchers a proven framework within which they can experiment with novel security policies and enforcement mechanisms, allowing them to focus on the security and performance tradeoffs of their contributions rather than reinventing and reimplementing solutions to the myriad known pitfalls of securing kernel-mode code.

B.2 Performance-Driven Design Changes

As our performance analyses (Section 10 and Appendix A) show, the entirety of Ombro's statistically significant overhead (only seen on certain IPI-heavy benchmarks) comes from the basic port of Xen to the SVA vISA, not from its compile-time security instrumentation on Xen (CFI, SFI, and split stack). In earlier versions of our vISA design, its overheads were substantially larger, and appeared on more benchmarks, than in the final version presented in this paper. These overheads yielded insights that drove several design changes to the vISA which dramatically improved performance to bring Ombro more in line with a non-SVA baseline.

Active vs. Loaded VMCSes in the vISA. The first such change, detailed in Section 5.3, corrected a limitation of the Shade [34] version of the vISA that was not previously evident because Shade's hypervisor support was not sufficiently sophisticated to support a real-world hypervisor or evaluate performance at any level higher than microbenchmarks that exercised the functionality of individual VMX instructions. Shade interpreted Intel's concept of there only being one "active VMCS" at a time on a processor [30] as meaning that a previous VMCS had to be explicitly unloaded (using the `VMCLEAR` instruction) before a different one could be loaded (via `VMPTRLD`). While porting Xen, however, it quickly became clear that Xen expects to be able to maintain multiple *loaded* VMCSes even as it switches between different *active* ones in vCPU context switches—i.e. to perform subsequent

VMPTRLDs on multiple VMCSes within a working set without VMCLEARing any of them until the respective vCPU is ready to be torn down or migrated to a different physical CPU.

To see how much impact Shade's more restrictive interface would have on real-world performance, we modified Xen to explicitly flush the outgoing vCPU's VMCS on every context switch and informally measured the impact on a *domU* guest running a context-switching stress test program we had created for debugging. If the (single-threaded) guest was the only significant load on the machine, no slowdown was evident compared to vanilla Xen—unsurprising, since few context switches were occurring. However, when we forced context switches by running a CPU-intensive application on all cores in the *dom0*, we found that the *domU*'s performance dropped precipitously, by over 4x.

Clearly, the hardware is able to take significant advantage of Xen's default behavior by retaining multiple VMCSes in on-chip cache across vCPU context switches. Ombro therefore (Section 5.3) improves the vISA to support calling the `loadvm` intrinsic on a new VMCS without having to first call `unloadvm` on a then-current one. While this (slightly) complicates the vISA's conceptual model of VMCS behavior compared to Shade, the performance improvements are well worth it, demonstrating that this facet of the native ISA is indeed essential to preserve at the vISA level.

Tying Guest State to SVA Thread Switches vs. VM Entry/Exit. The sole remaining source of vISA overhead that appeared in our benchmarks (Section 10 and Appendix A) is attributable to overhead on VM entry and exit. While entry/exit overheads should ideally not be performance-critical due to hardware-accelerated virtualization that makes VM exits rare, real systems fall short of this ideal. As our performance analysis in Section 10.3 explains, this is particularly true for IPI-heavy workloads that incur frequent VM exits. It is therefore desirable to minimize the SVA vISA's impact on entry/exit latency as much as possible.

As Section 5.2 discusses, Shade's version of the `runvm` intrinsic [34] was designed to present a conceptually clean abstraction wherein no guest state is ever active on the processor when running in host mode (outside of the `runvm` intrinsic itself) or vice versa. This necessitated that `runvm`'s implementation context-switch *all* system state elements, on every entry and exit, that could be modified by a guest and which could affect host software's view of system state.

While porting Xen, we realized that this design decision was overly prescriptive on hypervisor and host-OS design and negatively impacted performance, since it forced heavyweight state components such as the FPU and MSRs to be switched on every VM entry/exit. In practice, the hypervisor/kernel is not expecting SVA to completely hide the guest's existence from it; rather, it will save and restore its own state on higher-level context switches (between guest vCPUs or host userspace threads) to accommodate the guest's occupation of the processor. This allows the hypervisor/kernel to minimize unnecessary copying by refraining from disturbing major state components like the FPU except when it decides to schedule a different vCPU or thread to run; it also allows it to implement lazy FPU saving [6] if desired.

Since its original versions [16, 17], SVA has provided vISA primitives to assist OS kernels in safely transitioning between their own execution state and that of userspace threads as they handle interrupts, traps, and system calls and make context switches. This, we realized, is exactly the model used by real-world hypervisors for making context switches between vCPUs. It was therefore natural to eliminate Shade's excessive orthogonality between userspace-thread and guest-vCPU context switches in favor of unifying them under SVA's existing *thread* abstraction [13, 14]. As described in Section 5.2, guest vCPU state is now stored in the same fields within SVA's thread context structures as used for userspace threads, the only difference being that the hypervisor/OS kernel chooses to enter that context via a call to the `runvm` intrinsic (VM entry to VMX non-root mode) rather than via SVA's `iret` function (return to host Ring 3 from interrupt/trap/syscall handler).

Besides streamlining the SVA vISA, this conceptual change improved Ombro's VM entry/exit overhead from over 200% to just 95% (Section 10.2) and macrobenchmark overhead on the worst IPI-heavy outlier from 60% to 21.81% (Section 10.1). Based on further informal experimentation, we believe reducing the overhead further may be possible, but the current implementation has reached a point of diminishing returns, making it more profitable to focus on eliminating the root cause of excessive VM exits in the outlier benchmarks, e.g. through Intel's upcoming hardware IPI virtualization support [12] as discussed in Section 10.3 and Appendix A.1.

C Background on the Name *Ombro*

For readers who are curious what the name *Ombro* signifies, it is a word meaning "shade" or "shadow" in the constructed international auxiliary language Esperanto [19]—the only artificial language that has successfully become a "living" human language. This follows a loose tradition within our research group of naming systems after a general theme of "ghosts" or "shadows", which originated with systems building on the Virtual Ghost [14] project. These initially included Apparition [22] and Shade [34] (direct descendants of Virtual Ghost) and expanded to include some non-SVA-based shadow stack projects such as Silhouette [58], IskiOS [28], and Kage [24] (the latter two translating "shadow" in Greek and Japanese respectively). *Ombro* was doubly appropriate for the work presented in this paper as it can be interpreted either as a translation of "Shade" (the project we directly extend) or as referring to the "shadow stacks" we provide for Xen to protect return addresses.

The Esperanto origin of *Ombro* is also apropos to the nature of a virtual instruction set computing (VISC) system like SVA, as Esperanto is an academically-constructed yet practically-purposed human language designed to streamline

second language learning by minimizing irregularities and exceptions—much as the SVA virtual instruction set architecture (vISA) does in relation to native hardware ISAs to make analysis and protection of low-level system software easier. Like SVA's role in mediating the interface between hardware ISAs and low-level software, Esperanto is not meant to *replace* native languages but to supplement them in situations where they struggle to fulfill their communication goals.

The *Ombro* name was selected by the lead author (Ethan Johnson) who studied and learned Esperanto as a hobby during the development of this work. He has attained intermediate proficiency in the language and welcomes correspondence either in English *aŭ en Esperanto*.

Table 4: Unmodified Xen 4.12.0 vs. No Hypervisor (arrows indicate whether higher or lower is better)

Benchmark	**Units**	**Vanilla Xen**	**Std. Dev.**	**Bare Metal**	**Std. Dev.**	**% Ovhd.**
RocksDB	ops/s ↑	34557.667	0.1%	852098.333	1.1%	-2365.73%
Memcached	ops/s ↑	46960.733	0.2%	84270.000	0.6%	-79.45%
PostgreSQL	TPS ↑	4266.496	26.7%	7625.012	7.8%	-78.72%
Hackbench	s ↓	140.056	0.8%	37.645	4.2%	-73.12%
LevelDB	μs/op ↓	428.367	0.1%	168.999	0.2%	-60.55%
IPC (TCP Socket)	messages/s ↑	473697.400	13.3%	739706.667	2.2%	-56.16%
PostgreSQL	ms (avg. lat.) ↓	61.644	21.0%	32.989	8.5%	-46.48%
Apache	req/s	212698.913	0.4%	257396.147	0.3%	-21.01%
OSBench (Create Files)	μs/event ↓	24.270	0.3%	19.570	0.1%	-19.37%
PostMark	TPS ↑	4335.000	1.0%	5102.000	0.0%	-17.69%
pmbench	μs ↓	0.113	4.5%	0.094	1.0%	-16.54%
MBW	MiB/s ↑	6694.581	0.3%	7793.890	0.8%	-16.42%
ctx_clock	clocks ↓	6694.581	0.3%	7793.890	0.8%	-16.42%
t-test1	s ↓	23.587	0.9%	21.874	0.5%	-7.26%
iPerf (TCP)	Mbits/s ↑	28029.667	2.3%	29551.667	0.3%	-5.43%
SQLite	s ↓	83.164	0.8%	81.030	0.6%	-2.57%
BenchmarkMutex	ns ↓	39.967	0.9%	39.200	0.0%	-1.92%
StressNG (RdRand)	bogo ops/s ↑	250298.447	0.0%	251830.763	0.0%	-0.61%
OpenSSL (RSA 4096)	signs/s ↑	1516.867	0.3%	1499.200	0.6%	1.16%
Schbench	μs ↓	18762.667	0.7%	20819.200	3.1%	10.96%

Table 5: Unmodified Xen 4.12.0 vs. Ombro without CFI, SFI, or Split Stack Transformations (arrows indicate whether higher or lower is better)

Benchmark	**Units**	**Vanilla Xen**	**Std. Dev.**	**Ombro without instrumentation**	**Std. Dev.**	**% Ovhd.**
PostgreSQL	ms (avg. lat.) ↓	61.644	21.0%	59.805	19.9%	-2.98%
MBW	MiB/s ↑	6694.581	0.3%	6884.724	0.2%	-2.84%
pmbench	μs ↓	0.113	4.5%	0.110	3.8%	-2.60%
PostgreSQL	TPS ↑	4266.496	26.7%	4364.444	24.1%	-2.30%
OSBench (Create Files)	μs/event ↓	24.270	0.3%	24.207	0.1%	-0.26%
BenchmarkMutex	ns ↓	39.967	0.9%	39.933	1.0%	-0.08%
OpenSSL (RSA 4096)	signs/s ↑	1516.867	0.3%	1517.400	0.1%	-0.04%
SQLite	s ↓	83.164	0.8%	83.152	0.8%	-0.01%
ctx_clock	clocks ↓	240.000	0.0%	240.000	0.0%	0.00%
PostMark	TPS ↑	4335.000	1.0%	4335.000	1.0%	0.00%
StressNG (RdRand)	bogo ops/s ↑	250298.447	0.0%	250127.343	0.0%	0.07%
t-test1	s ↓	23.587	0.9%	23.642	1.1%	0.23%
Apache	req/s	212698.913	0.4%	208431.403	0.1%	2.01%
iPerf (TCP)	Mbits/s ↑	28029.667	2.3%	27447.667	2.4%	2.08%
Schbench	μs ↓	18762.667	0.7%	19934.857	9.8%	6.25%
Hackbench	s ↓	140.056	0.8%	149.015	0.5%	6.40%
IPC (TCP Socket)	messages/s ↑	473697.400	13.3%	430804.400	8.6%	9.05%
LevelDB	μs/op ↓	428.367	0.1%	494.449	0.1%	15.43%
RocksDB	ops/s ↑	34557.667	0.1%	28721.333	0.4%	16.89%
Memcached	ops/s ↑	46960.733	0.2%	35734.733	1.5%	23.91%
Geometric Mean						**3.32%**