VEIL: A Protected Services Framework for Confidential Virtual Machines

Adil Ahmad
Arizona State University

Botong Ou
Purdue University

Congyu Liu
Purdue University

Xiaokuan Zhang
George Mason University

Pedro Fonseca
Purdue University

Abstract

Confidential virtual machines (CVMs) enabled by AMD SEV provide a protected environment for sensitive computations on an untrusted cloud. Unfortunately, CVMs are typically deployed with huge and vulnerable operating system kernels, exposing the CVMs to attacks that exploit kernel vulnerabilities. VEIL is a versatile CVM framework that efficiently protects critical system services like shielding sensitive programs, which cannot be entrusted to the buggy kernel. VEIL leverages a new hardware primitive, virtual machine privilege levels (VMPL), to install a privileged security monitor inside the CVM. We overcome several challenges in designing VEIL, including (a) creating unlimited secure domains with a limited number of VMPLs, (b) establishing resource-efficient domain switches, and (c) maintaining commodity kernel backwards-compatibility with only minor changes. Our evaluation shows that VEIL incurs no discernible performance slowdown during normal CVM execution while incurring a modest overhead (2 – 64%) when running its protected services across real-world use cases.

CCS Concepts

• **Security and privacy → Trusted computing**.

Keywords

Confidential Virtual Machines, OS design, cloud security

ACM Reference Format:
Adil Ahmad, Botong Ou, Congyu Liu, Xiaokuan Zhang, and Pedro Fonseca. 2023. VEIL: A Protected Services Framework for Confidential Virtual Machines. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3623278.3624763

1 Introduction

AMD Secure Encrypted Virtualization (SEV) is a promising hardware mechanism to ensure the confidentiality and integrity of sensitive computations in cloud machines. SEV allows users to process their sensitive data in virtual machines (VMs) that are inaccessible to the cloud hypervisor and external devices, as well as encrypted in physical memory. The standard industry terminology for such VMs is *confidential virtual machines* (CVMs) [31, 59]. Given the popularity of VMs in the cloud, SEV has been rapidly adopted by major cloud providers like Microsoft Azure [93] and Google Cloud [58], which now offer CVM services to their users.

Unfortunately, a limitation of SEV is that an operating system kernel is part of its trusted computing base (TCB). In particular, although formally-verified kernels [71] or micro-kernels [16] can be used in a CVM, they typically require extensive system redesign, and formal verification is not an absolute guarantee of full system correctness [51]. Hence, users tend to employ commodity kernels made for convenience and compatibility, making the TCB particularly large. In fact, SEV officially only supports Linux [1], which has more than 31 million code lines and has hundreds of discovered vulnerabilities each year [4]. Given Linux's monolithic design, an attacker that leverages such vulnerabilities can steal sensitive user information or harm computational integrity in a CVM.

The lack of trust in large monolithic operating system kernels has driven a significant body of research to design *security monitors* that guarantee critical functionality despite kernel vulnerabilities. Unfortunately, existing monitors have undesirable trade-offs, especially for CVMs. Specifically, a common monitor design leverages a hardware-enforced privilege layer outside a virtual machine, e.g., the virtual machine monitor (VMM), to transparently monitor and control an untrusted operating system's behavior [30, 40, 66, 77, 94]. Unfortunately, external security monitors are at odds with the fundamental hardware-enforced CVM guarantee, which ensures integrity and confidentiality against outside software. While software enforcement techniques [42, 43, 45, 46] have been proposed for a security monitor, which could be leveraged within CVMs, these techniques still suffer from performance overhead and capabilities limitations that hinder deployment (§2).

This paper introduces VEIL, a CVM security monitor framework that efficiently protects critical system services—from preserving kernel code to enabling robust forensics—without trusting the kernel. VEIL leverages virtual machine privilege levels (VMPLs), a new hardware isolation mechanism available in all the latest AMD Milan server CPUs [27], to create a hardware-enforced privilege layer inside the CVM. VEIL only requires minor changes to commodity CVM kernels, none of which are related to core operating system functionality. Finally, the framework incurs a modest slowdown (up to 18%) when protected services are used while showing no discernible slowdown under normal execution.

VMPLs complement x86 rings to enforce additional memory isolation within a CVM. In particular, a VMPL (from 0 – 3) can be assigned to a virtual CPU (VCPU) during its initialization. VCPUs assigned higher privilege levels (e.g., VMPL-0) can define what

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 979-8-4007-0394-2/23/03…$15.00
https://doi.org/10.1145/3623278.3624763

memory region is read, written, or executed by VCPUs executing at lower privilege levels (e.g., VMPL-3). Once initialized, a VCPU cannot change its level, even if it can execute code in supervisor mode (CPL-0). This allows supervisor computations (e.g., the kernel) on such VCPUs to execute normally within their allowed regions while still enforcing memory isolation outside those regions.

Designing a security monitor framework using VMPLs requires addressing several challenges. First, VMPL provides a limited set of privilege levels, which are insufficient to support all needed services. Second, a security monitor and protected services must execute on all VCPUs (e.g., like the kernel), but VMPL only allows a VCPU instance to execute permanently at one privilege level. Third, in the presence of a security monitor, the operating system must execute at a lower privilege level where it is architecturally restricted by the hardware to leverage critical system features.

Veil overcomes the aforementioned challenges (§5). In particular, Veil carefully uses the limited VMPLs and combines them with traditional protection rings to enforce isolation between potentially-unlimited services. Instead of statically partitioning VCPUs between trusted and untrusted software, Veil replicates a single VCPU into copies that each executes a different software layer at a specific VMPL. Finally, Veil delegates all operating system functionality that is architecturally-restricted at lower privilege levels to its higher-privileged security monitor to maintain compatibility.

Veil can protect critical system services that require both confidentiality and integrity (§6). In particular, we show the flexibility of Veil by implementing three services, two of which require strong integrity guarantees to protect kernel code and system logs from corruption, while the third requires both confidentiality and integrity to shield sensitive user computations in protected *enclaves* [32, 92].

We built a Veil prototype (§7) to evaluate its practicality, security, and performance. Our prototype suggests that the CVM Linux kernel and host hypervisor can support Veil with minor (less than 1200 lines) code changes, making it easy to adopt. The framework and protected services required ~4100 code lines, small enough to be rigorously tested. We also analyzed and experimentally validated Veil's security (§8) to show that it can successfully defend itself against a broad class of attacks from the operating system. Finally, we evaluated the performance of Veil and its protected services using carefully-crafted custom benchmarks and real-world case-studies (§9). Our evaluation results show that Veil increases CVM boot time by less than 2 seconds, introduces a modest performance overhead between $2\% - 64\%$ to real-world programs that utilize a protected service, and has a negligible impact on system performance under normal CVM execution.

2 A Security Monitor for CVMs

Convenience and backwards-compatibility typically drive the use of commodity operating system kernels inside CVMs, resulting in a vulnerable software TCB [4]. One way to avoid this problem is to leverage a *security monitor*, a tiny software root-of-trust that enforces security invariants (e.g., sensitive data protection). This section describes current security monitor approaches and their trade-offs, which guide our CVM security monitor principles.

2.1 Current Approaches and Trade-Offs

Existing approaches implement security monitors by (a) leveraging a privileged hardware-enforced layer or (b) deprivileging the operating system using software techniques. We call these techniques *external hardware-based enforcement* and *internal software-based enforcement*, respectively, and discuss them below.

External hardware-based enforcement. Several systems [38, 40, 65, 66, 94, 112] leverage the introspection and control capabilities of VM monitors (VMM) to implement security monitors. For instance, the BlackBox system [65] leverages a tiny VMM and uses nested page tables (NPT) to restrict the operating system's access to a protected container's memory. Unfortunately, VMM-based security monitors are incompatible with CVMs, which prevent VM introspection because components outside CVM are not trusted.

In non-x86 systems, researchers have also leveraged software-controlled privileged layers (e.g., ARM TrustZone [30, 49], RISC-V machine mode [77]) that are both external to the VM and VMM for security monitors. Unfortunately, in practice, these layers are designed for machine management and are too privileged for cloud providers to allow access to cloud users. For instance, a cloud user that executes their security monitor in ARM TrustZone has access to all memory regions and could leak information from other users. While virtualization of the TrustZone layer is possible [67], it would still require trusting the cloud provider and their system admins.

Internal software-based enforcement. A security monitor can reside in the same hardware-enforced privilege layer as the operating system inside the CVM if the operating system is deprivileged using compilers [42, 43, 46] or source code instrumentation [45]. The remaining paragraphs in this section explain the trade-offs of these approaches in terms of performance and security.

Compiler approaches instrument the operating system's code to (a) implement bound checks on memory access operations to avoid corruption of trusted regions and (b) enforce control-flow integrity (CFI) to prevent unauthorized jumps to trusted regions. While these approaches can be ubiquitously applied to any system, they unfortunately incur non-negligible overheads even under normal system execution by requiring software checks on a significant number of memory accesses and branch instructions. For instance, the Virtual Ghost system increases system call latency for *all* computations by 3.9 times on average [42] which is undesirable.

In contrast to compiler approaches, the Nested Kernel [45] manually instruments the kernel's source code to force a security monitor call for sensitive operations (e.g., page table changes). The integrity of code instrumentation is ensured using binary code scanning and the x86 write-protection feature (CR0.WP [68]).

While the Nested Kernel is faster than compiler approaches, it is only designed to enable integrity by leveraging CR0.WP, and not confidentiality. Hence, the Nested Kernel (in its proposed form) cannot provide services like shielding sensitive programs from untrusted operating systems, since such services require confidentially keeping secret keys (e.g., for a secure communication channel between an application and a remote user). In fact, even services like system log tampering prevention that on the surface only seem to require integrity, indirectly require confidentiality for secure authentication [21] if the logs must be sent to remote parties through untrusted channels (e.g., the untrusted kernel's network

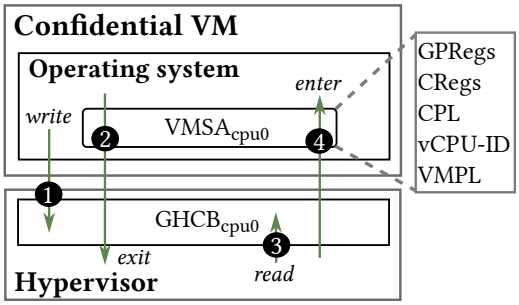

Fig. 1: Communication between a CVM and hypervisor on non-automatic exits (e.g., I/O-related).

stack [52]). Note that the Nested Kernel system can be updated to enable memory confidentiality (e.g., by mapping and unmapping regions from kernel page tables which are controlled by the Nested Kernel). However, this comes at an additional performance cost.

2.2 CVM Security Monitor Principles

Given existing security monitor approaches and their trade-offs, we decide on two principles for our framework: (a) *internal hardware-based enforcement* and (b) *versatile confidentiality and integrity protection.* In particular, hardware-based enforcement has the potential to obviate the high overhead of software mechanisms and avoid significant kernel changes for deprivileging. However, since CVMs only trust their protected context, the hardware-based enforcement must result in a security monitor internal to the VM. Additionally, a security monitor framework for CVMs should have both confidentiality and integrity guarantees, since it increases the versatility of protected services that the framework can support.

3 AMD SEV-SNP Background

This section describes how AMD's latest SEV product, SEV-SNP (Secure Nested Paging), protects CVMs and the workings of its new hardware feature, VMPL.

CVM protection at runtime. SEV-SNP prevents several direct and indirect attacks against CVMs from external software, including the hypervisor and BIOS, at runtime. This is achieved during address translation by checking the *reverse map* (RMP) table, which tracks the CVM's physical pages and their corresponding virtual addresses [23]. Direct attacks are prevented by ensuring CVM memory cannot be read to or written from outside. Indirect attacks are prevented by ensuring that a CVM's page tables (controlled by the hypervisor) remain consistent throughout the CVM's execution.

CVM protection at exits. A CPU executing a virtual machine, also called a *virtual CPU* (VCPU), must exit the virtual machine and invoke the hypervisor at hardware interrupts and hypercalls. For CVMs, SEV-SNP ensures that the VCPU state (e.g., general-purpose and control registers) is protected when an exit occurs. This state is saved in the *virtual machine save area* (VMSA), a per-VCPU memory region inside the CVM. When the VCPU resumes the CVM, its state is restored from the protected VMSA.

CVM-hypervisor communication. The hypervisor needs a portion of a VCPU's register state to service some hypercalls (e.g., IO-related). Hence, SEV-SNP allows the CVM to voluntarily provide this information to the hypervisor. This is achieved using a new instruction, `VMGEXIT`, and a shared memory region called the *guest-hypervisor communication block* (GHCB) [25]. Prior to the start of the communication, the CVM provides the GHCB's location to the hypervisor by writing this location to a model-specific register (MSR) that can be read by the hypervisor.

Fig. 1 illustrates the communication process. Before executing a hypercall, the CVM VCPU stores required information in its shared GHCB (1). Then, it executes a `VMGEXIT` to exit to the hypervisor (2). At this exit, the hardware stores the VCPU's state in its VMSA. The hypervisor reads the GHCB and provides the relevant hypercall service (3). Finally, the hypervisor executes `VMENTER` to resume the VCPU's context from its stored VMSA (4).

Importantly, this new communication mechanism is only needed for exits that require some state to be sent to the hypervisor (e.g., IO calls), which AMD calls non-automatic exits [23]. For other exits (e.g., timer interrupts) where no guest state is needed, called automatic exits, the VCPU directly exits (like a normal `VMEXIT`).

Virtual machine privilege levels (VMPL) This is a new privilege isolation mechanism available in SEV-SNP. It complements the existing *computer privilege levels* (CPL)—also called protection rings in x86—and allows the CVM to enforce what memory regions are accessible to *any* software running on a VCPU.

SEV-SNP provides four VMPLs, i.e., VMPL-0 to VMPL-3, where lower numbered levels are more privileged (like CPL). When a VCPU instance is created, its VMPL is assigned in its created VMSA and remains constant throughout the VCPU's lifetime. Note that apart from the boot VCPU instance, which is always created by the hypervisor at VMPL-0, all remaining VCPU instances (and their VMSAs) are created by the operating system in the CVM [25]. Hence, a CVM can freely assign any VMPL to its non-boot VCPUs.

Memory access control policies for VMPLs are hierarchical and expressive. For instance, privileged software on a VMPL-0 VCPU can specify access permissions for all VCPUs at lower levels, while software executing at VMPL-1 can only specify access permissions for VMPL-2 and VMPL-3. Additionally, an expressive set of permissions—*read, write, user-execution, and supervisor-execution*—can be assigned or restricted at each VMPL. Permissions are tracked in the RMP. A VMPL-0 privileged software (e.g., operating system) can modify permissions using a new instruction, `RMPADJUST`.

4 VEIL Overview

Built on our guiding principles (§2.2), VEIL is a general, trustworthy security monitor framework that *ensures the correct execution of critical system services in the presence of a buggy untrusted CVM operating system.* Inspired by the services enabled by prior work [42, 43, 45, 66], we show that VEIL is general enough to implement three major services in CVMs: (a) ensuring kernel code integrity, (b) protecting sensitive user computations in isolated execution contexts (commonly called *enclaves* [92]), and (c) preserving system logs for forensic analysis and attack reconstruction.

4.1 Threat Model and Assumptions

We trust that the AMD processor is correctly implemented. In particular, we trust it to correctly prevent direct access into the CVM from the outside world (e.g., other VMs), implement protection

features (e.g., VMPL), and perform all necessary operations for the remote CVM attestation protocol [47]. We follow the typical SEV threat model and assume that a hypervisor like InkTag [66] cannot be trusted since it is installed by untrusted cloud administrators.

Thanks to SEV remote attestation, a user can attest the load-time correctness of an installed operating system [47]. Hence, we assume that the attacker initially only controls all software and hardware external to the CVM (e.g., hypervisor, host machine BIOS). However, since the operating system contains exploitable vulnerabilities, we assume that the attacker will interact with the CVM (through network packets or hypervisor communication) and eventually compromise the CVM's operating system kernel. The attacker will use the compromised operating system and try to extract sensitive user information provided to the CVM or harm its integrity.

SEV does not guarantee availability and neither does our system. We also exclude data leaks through side channels [79, 89, 124, 127, 130], micro-architectural defects [34, 72, 87], and physical attacks (e.g., memory bus snooping [76], voltage scaling [33, 97]). Finally, we do not consider software bugs in the toolchain (e.g., CVM BIOS) provided by AMD to create CVMs [1].

4.2 Key Observation and Challenges

Our observation is that *virtual machine privilege levels (VMPL) (§3) can be employed to design a CVM security monitor framework based on our principles (§2.2)*. In particular, if VEIL executes its trusted software (e.g., a monitor) at a higher-privileged VMPL (e.g., VMPL-0) and the operating system at a lower-privileged VMPL (e.g., VMPL-3), it can leverage VMPL's protection to ensure correct execution of trusted software. This makes VEIL a hardware-enforced privilege layer inside the CVM with the ability to leverage efficient hardware checks. Also, VMPL protection can be enabled for both read and write accesses, ensuring both confidentiality and integrity.

Unfortunately, leveraging VMPL for our framework introduces several challenges as noted below:

C1: Insufficient implemented VMPLs. In theory, VEIL requires a separate VMPL to isolate each protected service or enclave, but the limited (4) VMPLs severely limit the number of implementable services or enclaves. A naive solution is to have a VMPL for all trusted components and one for all operating system components. However, this is insecure. For instance, the operating system might create a malicious enclave to run at the VMPL of trusted software.

C2: Resource-hungry VMPL assignment. A VCPU can switch between protection rings during execution (e.g., using `SYSENTER` to switch to the operating system's code at a system call), but its VMPL is statically assigned during creation (§3). Naively, all services and sensitive user computations must all have separate VCPUs, which is highly wasteful and severely limiting in terms of resources.

C3: Legacy kernel incompatibility. Since the kernel cannot execute at VMPL-0 anymore, it becomes architecturally-restricted for it to perform two essential functionalities: (a) boot additional VCPUs and (b) collaborate with the hypervisor for memory allocations [25]. Without proper care, this breaks CVM kernel compatibility.

5 VEIL Framework

VEIL has four components, namely the monitor (VEILMON), protected services, enclaves, and the untrusted software (collectively called the operating system) (Fig. 2). VEIL ensures that each component executes in a secure environment depending on their trustworthiness (§5.1). To avoid splitting VCPUs between different components, VEIL creates a replica of each VCPU for every component (§5.2). Finally, VEIL delegates all VMPL-0 functionality from the kernel to VEILMON to ensure legacy kernel compatibility (§5.3).

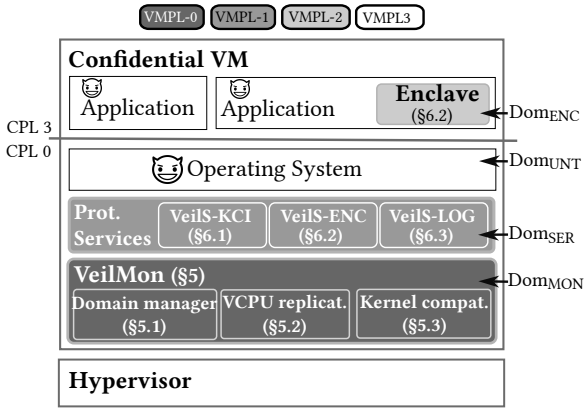

Fig. 2: An illustration of VEIL's system components with implemented multi-factor privilege domains.

5.1 Secure Dual-Factor Privilege Domains

VEIL implements four CVM privilege domains to securely execute its trusted software. We define a privilege domain as a new mode of execution within a VCPU formed by the combined privileges of traditional protection rings and VMPLs. The rest of this section explains how VEIL leverages domains.

***Dom$_{MON}$* (VMPL-0 + CPL-0).** This is the highest privileged domain and it is occupied by VEILMON. It allows VEILMON to execute any user or supervisor instruction and control VMPL memory access permissions for all domains. It is also the only domain that is afforded the architectural capabilities to create additional domains within the CVM (§5.2). The next paragraphs explain how VEILMON is securely loaded into memory and initialized in *Dom$_{MON}$*.

The memory contents (code and initial data) of VEILMON are measured during CVM launch and sent to the remote user for verification. In particular, these contents are compiled within the CVM boot image, a software component that initializes the CVM. During CVM launch, a SHA-256 hash of the boot disk image is generated and sent in a signed digest to a remote user for attestation [26]. In the attestation digest, the CPU also reports the VMPL of the software that requested the digest and additional data (e.g., information to establish a Diffie-Hellman shared key). Hence, the remote user can establish a secure communication channel with VEILMON by requesting an attestation digest from VMPL-0 software.

VEIL modifies the CVM boot process to ensure that VEILMON executes at *Dom$_{MON}$*. In particular, under native CVM execution, the hypervisor creates a single VCPU to set up initial boot and run the kernel at the highest CVM privilege (i.e., *Dom$_{MON}$*). VEIL replaces the kernel in this process with VEILMON. As needed, VEILMON creates new domains for protected services, the kernel, and enclaves.

***Dom_{SER}* (VMPL-1 + CPL-0).** Protected services execute within this domain. Compared to Dom_{MON}, this domain restricts access to VMPL-0 regions (where VEILMON resides) and the creation of additional domains. VEILMON achieves the former by executing `RMPADJUST` on all memory regions in this domain, while the latter is architecturally-restricted. None of the restricted functionality is required by the protected services, since they can rely on VEILMON. Hence, to better adhere to the *principle of least privilege*, we chose this domain for protected services. Finally, like VEILMON, protected services are also included in the CVM boot image.

***Dom_{ENC}* (VMPL-2 + CPL-3).** Enclaves use this domain, which is configured for mutual protection of both enclaves and the operating system. In particular, the protected service VEILS-ENC uses the domain's higher VMPL to prevent the operating system from accessing enclave memory. At the same time, VEILS-ENC ensures that enclave cannot execute supervisor (CPL-0) instructions or access unauthorized memory regions (Dom_{SER}, Dom_{MON}, and the operating system). If an enclave can execute supervisor code, it can remap the page table entries and access a different enclave's pages since all enclaves execute at Dom_{ENC} (with VMPL-2). Moreover, the operating system can protect itself from an unprivileged enclave using traditional address space isolation and retain control of core privileged functionality (e.g., memory allocations and management). We provide more details about enclaves in §6.2.

***Dom_{UNT}* (VMPL-3 + CPL-0/3).** Finally, the untrusted domain is used by the operating system and all its created processes. Executing at the least-privileged VMPL, the operating system is restricted from accessing memory regions of higher VMPL software. Specifically, VEILMON executes `RMPADJUST` to remove access to all sensitive memory and states from Dom_{UNT}. These permissions cannot be changed by the operating system: if it calls `RMPADJUST` for pages that are restricted in Dom_{UNT}, the CPU raises a nested page fault (#NPF) which leads to a system halt [25]. Additionally, at Dom_{UNT}, the kernel cannot execute a few architectural features (§5.3). However, they are only required during initialization and can be mediated by VEILMON; hence, the kernel's execution in Dom_{UNT} results in a typically negligible overhead (§9.1).

5.2 Replicated VCPUs for Domain Switch

Instead of resource-hungry static partitioning of VCPUs between domains (§4.2), VEIL creates replicas of every VCPU and assigns them to different domains for efficient utilization. Static partitioning wastes VCPU resources since a VCPU instance can only securely execute one domain due to permanent VMPL assignment during initialization (§3). For instance, if a VCPU initialized at Dom_{MON} directly transitions into a lower-privileged software (e.g., kernel), the software will gain all privileges of the security monitor. Replication ensures that the same VCPU can context switch to a different software by transitioning to a VCPU instance initialized at the software's domain. As we explain in the next paragraphs, this switch is completed using the hypervisor, and communication between domains is through shared memory.

Per-domain VCPU replication. VEILMON follows four steps to create a copy of a VCPU instance and assign it to a different domain. First, it allocates a new VMSA with the same VCPU-ID and target domain VMPL. Second, for Dom_{SER} and Dom_{ENC}, it initializes important architectural structures (e.g., stack, page tables, global and interrupt descriptor tables). This is not needed for the Dom_{UNT}, since the operating system kernel automatically initializes these structures. Third, VEILMON sets addresses for initialized structures (e.g., location of the page tables) and the correct software entry point (`rip`) in the VMSA. Finally, VEILMON executes a hypercall (`VMGEXIT`) to ask the hypervisor to execute the new domain.

Inter-domain communication blocks (IDCBs). These are shared memory regions used for bi-directional domain communication. For any two domains, IDCBs are allocated in the less privileged domain's memory to ensure all parties can access it. For instance, IDCBs between the operating system and VEILMON are allocated in a reserved part of the kernel's memory. Additionally, IDCBs are assigned at a per-VCPU granularity to avoid contention.

Hypervisor-relayed domain switch. Switching a domain requires exiting the VCPU context and re-entering using a different domain's VMSA; hence, it must be performed with the hypervisor's help. Fig. 3 shows inter-domain communication between VEILMON and the operating system. In particular, the operating system first transcribes its required service from VEILMON in the IDCB (❶). Then, the operating system writes a message to the hypervisor in GHCB (❷) asking for a domain switch to Dom_{MON}. It exits to the hypervisor using `VMGEXIT` (❸) and allows the hypervisor to process the message (❹). The hypervisor resumes the VCPU (with `VMENTER`) but it uses Dom_{MON}'s VMSA (❺). Hence, VEILMON executes (on the same VCPU), reads the message in IDCB, and processes the operating system's request (❻).

5.3 Privileged Functionality Delegation

Since the operating system kernel executes at Dom_{UNT}, it becomes architecturally-infeasible for it to perform two functionalities: (a) boot VCPUs during initial system boot or hotplugging scenarios and (b) accept pages from the hypervisor or change the current page state. Hence, VEIL delegates these functionalities to VEILMON, which checks for correctness.

VCPU boot delegation. VCPUs can be *hot-plugged* into a CVM at any time. Like domain creation, this process requires VMPL-0 software to create a new VMSA (using `RMPADJUST`) and start the VCPU's execution through a hypercall. We modify the kernel to handle initialization of the required VCPU state, but perform a domain switch to VEILMON for VMSA creation. VEILMON generates the VMSA and boots the VCPU at Dom_{UNT} (VMPL-3). For every new hotplugged VCPU, VEIL also creates replicas of the VCPU instance to execute trusted domains (e.g., Dom_{SER}) (§5.2).

Page state change delegation. A CVM can receive additional memory pages from the hypervisor and share some of its pages with the hypervisor (e.g., to use as a software bounce buffer for device I/O). However, before a page state occurs, the CVM must execute `PVALIDATE` on that page. We modify the kernel to redirect all `PVALIDATE` calls to VEILMON, which checks that these calls are not made for trusted memory regions, then executes them.

6 VEIL Protected Services

VEIL ensures the correct execution of system services in the presence of an untrusted CVM operating system. Any service can leverage such protection using VEIL. We implemented three services to showcase different applications of VEIL.

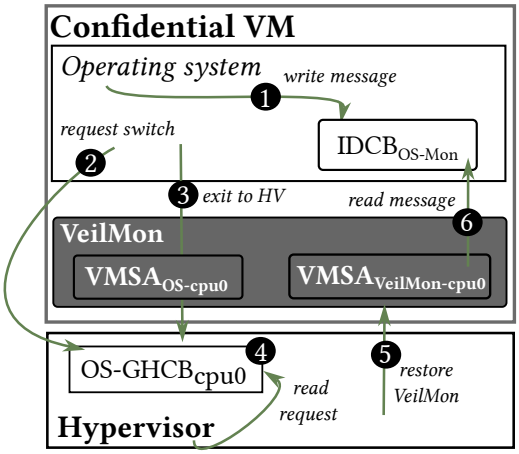

Fig. 3: An illustration of inter-domain communication between the operating system and VeilMon.

6.1 VeilS-Kci: Kernel Code Integrity

Kernel code injection allows attackers to arbitrarily modify the kernel. Hence, operating systems like Windows [94] and Samsung's Android OS [30] use an external privileged security monitor (e.g., at the VMM [111]) to enforce kernel code integrity—only approved kernel code executes in CPL-0. Fortunately, even though external security monitors are incompatible with CVMs (§2), Veil can enforce kernel code integrity using a protected service, VeilS-Kci.

Kernel memory W⊕X protection. VeilS-Kci ensures that the *write-or-execute* (W⊕X) semantic is upheld for kernel memory regions at Dom_{UNT} using VMPL. Hence, even if the attacker tricks the kernel into disabling its own prevention measures (e.g., by using a write gadget to unset the Not-eXecutable bit of a page table entry), they still cannot run malicious code in supervisor mode. VeilS-Kci achieves W⊕X by executing `RMPADJUST` and disabling (a) write permissions on all kernel code pages and (b) supervisor execution permissions from all kernel data pages.

Module signature verification and loading. Operating systems are designed to execute signed loadable kernel modules (e.g., additional device drivers) at runtime. VeilS-Kci securely supports this functionality in CVMs. A naive implementation would be for VeilS-Kci to only check signature integrity of a kernel module. This is insecure because it results in a classical *time-of-check-to-time-of-use* (TOCTOU) vulnerability. In particular, an attacker that has gained root privilege can modify module contents after the signature is verified. Hence, except for memory allocation which is left to the operating system, VeilS-Kci performs the remaining module initialization steps. This includes verifying the signature on a kernel module, loading the module in memory, relocating symbols using a protected symbol table, and write-protecting the prepared text region (by executing `RMPADJUST`).

6.2 VeilS-Enc: Shielded Program Execution

CVMs run sensitive computations containing user's personal information. We designed the VeilS-Enc service to provide additional protection to such sensitive computations from an untrusted CVM operating system. This approach creates a *nested trusted execution environment* for sensitive computations inside the CVM that is protected from the hypervisor and operating system.

VeilS-Enc shields sensitive computations through the in-process isolation model, which has been particularly successful in the cloud given the wide availability of Intel's Software Guard eXtensions (SGX) [92]. This model allows an application to create a protected context (or an *enclave*) inside its address space, which is inaccessible to all software outside the enclave. All sensitive code and data are stored in the enclave's memory. The enclave has well-defined protected entry points (e.g., starting functions), and it must exit to the untrusted application to execute code outside its context (e.g., on system calls and interrupts). VeilS-Enc's provided design and security abstractions are functionally-equivalent to SGX's (as we discuss in the remaining paragraphs of the section and §10).

Enclave initialization and measurement. The operating system lays out the initial memory regions of the enclave. Then, VeilS-Enc protects the enclave region from further direct modifications by the operating system and measures the region's initial state. This measurement is provided to the remote user for enclave attestation.

The program to be shielded inside an enclave is provided as a self-contained binary (e.g., with its own C library) with no outside calls. Using IOCTL to a kernel module (§7), the process asks the operating system to install the binary within an enclave. The operating system copies the binary into memory, relocates its symbols, and initializes other needed memory regions (e.g., stack). After installation, the operating system invokes VeilS-Enc to finalize the enclave.

VeilS-Enc ensures the operating system cannot access the enclave's memory or change its layout post-installation. In particular, VeilS-Enc asks VeilMon to create an enclave domain (Dom_{ENC} in §5.1) and revoke all permissions from enclave regions at Dom_{UNT} (using `RMPADJUST`). VeilS-Enc also clones the user process's page tables into its protected memory and performs several initialization scans (next paragraph). An enclave uses these protected page tables during execution, ensuring its initial layout is preserved.

VeilS-Enc ensures two invariants are satisfied while scanning page tables during initialization. First, there should be a *one-to-one* mapping between virtual and physical pages. This avoids malicious remapping from the operating system and simplifies enclave measurement for remote attestation. Second, each enclave's allocated set of physical pages should be disjoint. Since all enclaves execute at Dom_{ENC}, a common physical page will allow a malicious enclave to steal another enclave's contents. If either invariant is not satisfied, VeilS-Enc terminates the enclave initialization process.

After protecting enclave memory and layout, Veil creates a measurement of this region and reports it to a remote user. This measurement is a SHA-256 cryptographic hash like other enclave systems [77, 92] and it is derived from both page contents and metadata (e.g., permissions). The measurement is sent to the user through VeilMon's secure user communication channel which is established after the SEV remote attestation process (§5.2).

User-mapped GHCB for entry and exits. The untrusted application enters the enclave for secure computation. Later, the enclave exits to the untrusted world for the handling of system calls and interrupts. This section discusses the challenge in enabling enclave entry and exits, and our solution to address the problem.

During enclave entry and system call exits, an unprivileged (CPL-3) process must send a message to the hypervisor for a domain switch between Dom_{UNT} and Dom_{ENC} (§5.2). Recall that this message is sent through a guest-hypervisor communication block (GHCB) which requires a privileged write to a model-specific register (`wrmsr`) (§3). However, an unprivileged process is architecturally-restricted from executing this privileged write. VEIL solves this problem by instructing the operating system to (a) automatically set the GHCB MSR before scheduling an enclave-running process and (b) map a per-thread GHCB to the process' address space. Both the application and the enclave can write their messages to the mapped GHCB and execute a hypercall (`VMGEXIT`) for a domain switch. To prevent errant hypercalls, the hypervisor is instructed to only allow domain switches between Dom_{UNT} and Dom_{ENC} using this GHCB. If the operating system does not map the GHCB correctly, the CVM crashes on an attempted domain switch. Hence, the operating system cannot leverage control over the GHCB to harm the enclave's confidentiality.

Apart from system calls, enclaves also incur exits at interrupts. Enclaves rely on the untrusted world for I/O (through system calls), hence all interrupt exits faced by an enclave do not require any information from the enclave context and the CVM automatically exits to the hypervisor (§3). VEIL instructs the hypervisor to relay these interrupts to Dom_{UNT} for handling. If the hypervisor does not relay interrupts and forces interrupt handling at Dom_{ENC}, the CVM will halt with a nested page fault (#NPF), since the operating system interrupt handler is inaccessible at Dom_{ENC} (previous heading).

Secure collaborative memory management. The enclave process' memory region is collaboratively and securely managed by the OS and VEILS-ENC during execution. This process is analogous to how the SGX microcode and OS collaboratively manage SGX enclave regions [92]. The main difference is that enclave page tables are kept by VEILS-ENC (instead of the OS which maintains SGX enclave page tables), hence all updates to enclave page tables are made by VEILS-ENC. The rest of this section describes how secure collaboration works for demand paging and permission changes.

At runtime, if the operating system must free an enclave page, it will send a request to VEILS-ENC. At this request, the service creates an integrity hash of the enclave page with a freshness counter. Then, it encrypts the page's contents using a per-enclave secret key, removes the page mapping from the enclave page tables, and allows the OS to access the page and free it. Subsequently, when the enclave tries to access the page, it raises a page fault.

Page faults during enclave execution are trapped to the hypervisor, which is instructed to send them to the operating system (Dom_{UNT}). After retrieving the faulted page from disk, the operating system sends a request to VEILS-ENC to decrypt and remap the page into the enclave's page tables. At this point, VEILS-ENC (a) copies the page into protected memory, (b) decrypts the page, and (c) verifies that the OS retrieved the correct page using the stored fresh integrity hash. If the verification checks pass, VEILS-ENC adds the mapping to the enclave page tables. To ensure the correctness of remapping, the OS also tracks which physical page belongs to which enclave virtual address, like SGX.

The OS is only allowed to change permissions (e.g., at `mprotect`) of non-enclave regions, while enclave region permission changes are directly handled by VEILS-ENC. In the latter case, permission change requests are sent by the enclave to VEILS-ENC using the enclave's GHCB (previous section). Note that permission changes to non-enclave regions must also be synchronized with enclave page tables, since the enclave will use its own page tables to access these regions. In this case, the OS is instructed to call VEILS-ENC for synchronization of permission changes between both page tables.

System call redirection to untrusted application. System calls require userspace buffers (e.g., read a file into a user buffer) from a process's context, but the enclave memory is inaccessible to the operating system. Hence, the enclave must redirect system calls to the application (like OCALLs in SGX [92]). In particular, the enclave copies the required information for a system call (e.g., buffer regions) from the enclave memory to shared application memory. Then, the enclave exits and requests the application to execute the system call on its behalf. On return from system calls, the enclave must carefully sanitize results (e.g., check that returned pointers do not belong to trusted memory regions before referencing) to prevent IAGO attacks [37].

6.3 VEILS-LOG: System Audit Log Protection

The operating system collects detailed audit logs of security-critical machine events (e.g., kernel module installation) for forensic analysis. Unfortunately, a key limitation of commodity system auditing frameworks (e.g., Linux's Kaudit [119]) is that an attacker can trivially tamper with these logs after compromising the operating system [21, 103, 104]. VEILS-LOG enables the CVM to securely isolate logs from the operating system. A user can query VEILS-LOG through a secure channel (§5.1) to retrieve logs.

Reserved append-only log storage. VEILS-LOG reserves a large memory region for log storage (in Dom_{SER}) and provides APIs to the operating system that allow append-only access to the storage. Note that the size of the reserved region must be large enough that a user can retrieve logs before it overflows. Typically, machines produce about 1GB [63] of logs every day; hence, with a 1GB storage region, the user should retrieve logs everyday.

Execute-ahead log protection. Logs are protected before the system executes an event configured by the user to be critical (??). This ensures that logs are available if the attacker compromises the machine at said event. To achieve this, we insert a hook in the operating system's built-in auditing framework to send a log entry to VEILS-LOG using an inter-domain communication block and a domain switch (§5.2). VEILS-LOG appends the entry into the reserved log storage and performs a domain switch back to the operating system, which then executes the event. Note that the operating system is only trusted to relay correct logs until the point it is compromised. Logs until the kernel compromise are typically sufficient to analyze the attack origin and vector.

7 Implementation

This section describes the implementation steps we took. We will open-source our prototype to help foster development.

CVM Linux kernel support for VEIL. We modified the Linux kernel v5.16.0-rc4 provided in AMD's GitHub repository for SEV-SNP guests to support VEIL. None of our implemented changes are made to the core functionality of the kernel (e.g., memory and page table allocation). Instead, they either support kernel execution at

Dom_{UNT} (§5.3) or hook the kernel's execution to VEIL's protected services. For the latter, we modified Linux's kernel audit (kaudit) to call VEILS-LOG once a log entry is created (at the `audit_log_end` function), as well as the kernel module loading and unloading routines (`load_module` and `free_module`) to call VEILS-KCI. In total, we removed ~50 lines of code and added ~560 lines of code to the native kernel source code. We also wrote a kernel module to support enclave creation and protection. The module creates and initializes a protected region in a program's address space, allocates a GHCB for the program, and calls VEILS-ENC to finalize the enclave. The kernel module was written with ~700 lines of code.

Hypervisor support for VEIL. Our host ran Ubuntu 20.04.3 LTS with Linux v5.14.0-rc2 provided in AMD's GitHub repository [1] for SEV-SNP hosts. We made three changes to its KVM hypervisor. These changes (a) maintain VMSAs for newly-created domains (in struct `vcpu_svm`), (b) install hypercall handling routines for domain switching, and (c) switch Dom_{ENC} to Dom_{UNT} on automatic interrupt exits during enclave execution. The changes required ~10 lines of code deletion and ~400 additional lines of code.

Framework and protected services. VEIL's security monitor and protected services are written as a C library. VEILMON currently does not implement cryptographic functionality for communication and measurement. We expect this functionality to work like it does for other enclave systems [42, 77]. Moreover, at runtime, VEIL reuses some portion of the kernel's code. This was done only to ease the implementation burden (e.g., reuse module loading primitives). In the future, all such functionality can be independently implemented in VEIL's protected services. Finally, VEILMON's functionality is implemented with ~4100 lines of code in total. This is small enough to be ported to a safer language (e.g., Rust), provided formal guarantees (e.g., Komodo [49]), or robustly tested using existing system software testing approaches [55, 56, 88].

Enclave software development kit (SDK). We built this kit to facilitate the development of enclaves for VEILS-ENC. It contains a modified C library, based on musl-libc [98]. The library automatically (a) communicates with the VEIL kernel module to initialize and remove enclaves, (b) handles enclave entries and exits through well-defined APIs that invoke `VMCALL` to the hypervisor, and (c) handles system call redirection by copying system call-related memory regions (e.g., argument pointers) from enclave memory to untrusted memory (§6.2). The SDK also implements an internal heap allocator for enclaves using the dlmalloc [75] implementation. To implement this SDK, we added ~2200 lines of C code to musl-libc.

One of the challenges in implementing the SDK was automatically inferring grammar for enclave system call handling. We addressed this by implementing a system call sanitizer that leverages system call grammar rules from a famous and well-maintained OS fuzzer, Syzkaller [57]. In particular, our sanitizer uses the rules to create a C library that performs a *deep copy* of each system call argument and included memory pointers. While the specifications provided by Syzkaller proved to be generally robust, we found discrepancies in several system calls using our unit-tests. Hence, we manually refined our sanitizer to address them.

The sanitizer is guided by both a call and type specification. The call specification encodes the high-level information about arguments used in each system call. The type specification contains the signature of various types used in system call arguments (e.g., struct, pointer). It also contains high-level semantic information, such as the length constraint relationship between different arguments. For instance, in the `write` system call, the third argument specifies the length of the second argument, which is a buffer. Our system call sanitizer was written in ~1100 lines of Go code, and we wrote ~500 lines of C-based unit-tests to refine the sanitizer.

Since our SDK is in prototype stage, it has some limitations, none of which we believe significantly impact our performance results. In particular, the SDK and VEILS-ENC currently only support single-threaded enclaves and do not support secure collaborative page swapping (§6.2), instead all enclave pages are mapped during initialization. Supporting multiple enclave threads requires two changes. First, the OS kernel's scheduler must request the scheduling of the correct enclave thread from VEILMON. Second, VEILMON must create a VMSA for the enclave thread on each VCPU and synchronize them so that the thread can execute on any VCPU. Note that the OS changes are minor, while a significant portion of VEIL's code can be reused to implement VMSA creation and synchronization in the future. To avoid synchronization issues across VCPUs, we currently leverage `taskset` to pin the single enclave thread to one VCPU during its execution.

Our SDK prototype supports 96 system calls, but additional system calls can be ported using our sanitizer. Also, our SDK only enables basic protection against IAGO attacks [37] by ensuring all pointers returned by the operating system on system calls (e.g., at `mmap`) belong to memory regions outside the enclave. Complete protection against IAGO attacks is an active area of research [44] orthogonal to our key contribution.

Syscall coverage using Linux Test Project (LTP). We evaluated our SDK's system call handling by conducting tests using the LTP suite [5]. LTP's kernel tests contain testcases that (a) specifically evaluate system call robustness [8] and (b) general system functionality [6]. We evaluated our SDK on both.

On the system call robustness cases, our prototype successfully completed all tests for 85/96 supported system calls. We believe the reason for some system calls not passing all tests is that we did not implement support for all their semantic cases, opting instead to focus on the more common functions used by real-world applications. Prior study [123] shows that only a subset of the system call interface is required to run the majority of applications. This is why musl and popular library OSs also only support a subset of the POSIX semantics [73, 98, 122]. For the unsupported system calls, our SDK is designed to kill the enclave and exit on their execution. Hence, our SDK failed all tests for these system calls. In total, our SDK passed 276 out of 1393 system call test cases.

Our SDK also successfully executed 180 out of 639 system functionality tests. These evaluate different system aspects like cryptographic implementations and filesystems. A large chunk of the tests that passed were related to the supported filesystem calls. The remaining failed because they executed unsupported system calls (e.g., `ioctl` [7]) or bash scripts [9].

Although our SDK only passes a small portion of the LTP tests, it is still robust enough to run many important real-world programs (§9.2). Finally, a future Library OS integration can help address the shortcomings of our prototype (§10).

Table 1: Potential attacks against VEIL's framework and implemented defenses (§5.1–§5.3)

Attack	VEIL defence
At boot-time	
Load mal. code at Dom_{MON}/Dom_{SER}	Remote attestation
During domain enforcement	
Read/write at Dom_{MON}/Dom_{SER}	Restricted by VMPL
Adjust VMPL restrictions	`RMPADJUST` prohibited
Overwrite sensitive registers	Protected in Dom_{MON}
Overwrite page tables	Protected in Dom_{MON}
Create VCPU at Dom_{MON}/Dom_{SER}	Control creation
During inter-domain comm.	
Overwrite IDCB	Protected in Dom_{SER}
OS sends malicious request	OS request sanitized

8 Security Analysis and Validation

This section analyzes the security of VEIL by first discussing various attacks against the framework and implemented services. It concludes with the results of our experimental security validation.

8.1 Analyzing Framework Security

The VEIL framework is the root-of-trust for protected services. The attacker can try to (a) attack the framework during boot-time loading, (b) circumvent VEIL's domain enforcement at runtime, or (c) harm inter-domain communication.VEIL implements protections for attacks at each of these stages (Table 1).

Preventing boot-time attacks. At boot-time, the attacker can try to load a malicious boot disk into the CVM, instead of VEIL's boot disk. This would allow the attacker to execute malicious code at the privileged Dom_{MON} and Dom_{SER}. VEIL prevents this attack by leveraging SEV's remote attestation to measure and report initial disk contents to a remote user.

Preventing domain enforcement attacks. At runtime, an attacker can try to directly access a trusted domain's memory contents, overwrite their architectural state, and spawn new attacker-controlled VCPUs at privileged domains. VEIL prevents direct access into privileged domains by leveraging VMPL's restrictions. These restrictions cannot be removed by the attacker, since the attacker is unable to execute `RMPADJUST` on higher-privileged domain memory regions. Additionally, all sensitive domain state (e.g., registers, page tables) is protected in Dom_{MON}, which is inaccessible to the attacker. The attacker can try to spawn a new VCPU to access Dom_{MON}, but only VEILMON can create a new VCPU (by executing `RMPADJUST` for a new VMSA), and it only allows new VCPU instances to the operating system at the restricted Dom_{UNT} (§5.3).

Preventing inter-domain communication attacks. The attacker can try to overwrite the messages passed between different domains (e.g., to trick VEILMON into lifting VMPL permissions). Except for messages from the operating system, all IDCBs are stored in protected memory regions (Dom_{SER}) (§5.2). The messages received from the operating system are sanitized to ensure enforcement. Specifically, the OS passes pointers during its communication with VEILMON and protected services; hence, it could try to pass a pointer to protected regions and trick trusted software to overwrite these regions. To prevent this attack, before referencing an untrusted memory address pointer, VEILMON checks that it does not point to a protected region (e.g., VEILMON memory). VEILMON can perform this check since it keeps track of all protected memory regions at runtime. VEILMON also provides this information to protected services so that they may also perform the check.

8.2 Analyzing Protected Services Security

This section describes how each VEIL protected service enforces security invariants inside CVMs.

Enforcing kernel code integrity with VEILS-KCI. The attacker can try to inject malicious code into the kernel by overwriting existing text regions, creating new text regions, or loading malicious kernel modules. VEILS-KCI enforces *write⊕supervisor-execute* on all kernel memory using VMPL restrictions (§6.1). Hence, even if the attacker can disable the operating system's protections (e.g., SMEP, NX bits), they still cannot overwrite existing text regions or create new text regions. Moreover, these enforcements are never disabled at Dom_{UNT} and all kernel modules are loaded through VEILS-KCI, which checks their signature before installation.

Shielding program execution with VEILS-ENC. The attacker can try to compromise an enclave using the operating system and a different attacker-controlled enclave. In particular, the attacker can try to load a malicious binary into the enclave to steal provided user data, read or write enclave regions, and overwrite sensitive enclave states. VEILS-ENC prevents all these attacks (Table 2).

VEILS-ENC ensures the load-time correctness of the enclave by measuring the initial memory contents and layout. This trusted measurement is provided to the remote user for attestation through VEILMON's secure channel. Only after attestation passes, the remote user sends their sensitive information to the enclave.

At runtime, if the operating system tries to access enclave memory regions or the enclave's interrupted processor state (inside the VMSA), the CVM halts on a nested page fault (#NPF) since these regions are protected in Dom_{ENC} and Dom_{MON}, respectively. Additionally, the enclave's page tables are protected in Dom_{SER} (during initialization), hence the attacker cannot modify them either.

An attacker might try to load a malicious enclave at Dom_{ENC} to steal or modify other enclave contents. VEILS-ENC prevents attacks from malicious enclaves by ensuring that each enclave is initialized with a disjoint set of physical pages. Hence, even though a malicious enclave executes at Dom_{ENC}, it cannot read another enclave's pages. Moreover, the enclave is not allowed to execute supervisor code in Dom_{ENC}, therefore it cannot change defined mappings.

Finally, the attacker can also try to launch two attacks using the hypervisor and leak sensitive enclave information. First, the hypervisor can attempt to modify the enclave register state stored in the VMSA [96]. Second, the hypervisor can refuse to relay interrupts to the untrusted world during enclave execution, and force an execution of the operating system's code at Dom_{ENC}. The attacker is unsuccessful on both accounts. In particular, the enclave's VMSA is stored inside the CVM, hence it cannot be accessed. Additionally, Dom_{ENC} cannot access kernel code (since it is unmapped in the enclave's page tables) and neither can the enclave execute supervisor instructions (since it is restricted using VMPL). Hence, if the hypervisor does not relay interrupts to Dom_{UNT}, the CVM halts with a continuous set of #NPFs due to permission violation.

Table 2: Potential attacks against enclaves and implemented defenses (§6.2).

Attack	VeilS-Enc defence
From CVM OS	
Load incorrect binary	Enclave attestation
Read/write memory	Restrictions in Dom_{UNT}
Modify physical layout	PTs protected in Dom_{SER}
Violate saved state (e.g., `rip`)	VMSA protected in Dom_{MON}
Incorrect GHCB mapping	CVM crash on `VMGEXIT`
From hypervisor	
Violate saved state (e.g., `rip`)	VMSA protected in CVM
Refuse interrupt relay	CVM halts with #NPF
From malicious enclaves	
Access memory from Dom_{ENC}	Disjoint physical pages
Execute OS code in Dom_{ENC}	Disallowed in Dom_{ENC}

Protecting system audit logs with VeilS-Log. A compromised kernel can try to modify the stored log entries (produced by the kernel in an honest state) by directly overwriting the log buffer. The log buffer cannot be accessed in the operating system's Dom_{UNT}—it can only be accessed at Dom_{SER}. Only the remote user can ask for stored logs to be removed (after retrieval) through an authenticated and secure communication channel.

8.3 Validation

We designed and executed two attacks to validate the correctness of implemented protections. We found that Veil's protections hold against both attacks. The first attack tried to overwrite VeilMon page table entries, and harm the monitor's integrity. For this attack, we mapped the page tables to the operating system's address space. When we tried to modify the page tables from the operating system, the CVM halted with continuous nested page faults (#NPFs). This signals an expected VMPL violation. The second attack tried to overwrite a kernel module's text region after VeilS-Kci was activated. We set the *write* bit in the operating system's page tables to disable page table-based W⊕X protections. On overwrite attempt, the CVM halted with continuous #NPFs again.

9 Performance Evaluation

This section describes Veil's performance through several micro-benchmarks and case-studies. All experiments were executed on a server machine with an AMD EPYC 7313P 16 core CPU, 80 GB of DDR4-3200 memory, and a 500 GB SATA SSD storage drive. On this machine, we created an SEV-SNP virtual machine with 4 hardware-accelerated VCPUs, 2 GB of memory, and a 50 GB storage drive (using VIRTIO [13]), using AMD's GitHub repository [1] scripts.

9.1 Micro-Benchmarks and Analysis

Initialization time. During CVM boot, Veil must initialize and protect Dom_{MON} and Dom_{SER}. We measured the time taken (using `RDTSC`) to complete these steps during 10 CVM boot-ups. On average, Veil increased boot time of the CVM by ~2 seconds. Over 70% of this time is needed to protect domain state, which requires executing `RMPADJUST` on all physical pages. This results in a memory access to every page before adjusting permissions. Nevertheless, this is only a 13% increase over the native CVM boot time (which is already longer than regular VM boot times) and a one-time cost.

Domain switch cost. To measure the average cost of a hypervisor-relayed domain switch (§5.2), we performed 10,000 domain switches between the operating system and VeilMon and measured time using `RDTSC`. We found the average cost to be 7135 cycles for a domain switch. The major cost is SEV-SNP's register state save and restore that occurs on `VMGEXIT` and `VMENTER` (§3). State save and restore is known to be expensive for other trusted execution environments like SGX [101] too. Notably, a normal exit (using `VMCALL`) on a non-SEV-SNP VM takes ~1100 cycles on our machine. Nevertheless, the impact of this extra cost is limited if the CVM does not switch domains frequently (as described next).

Background system impact. Even by default, the kernel executes at Dom_{UNT} and relies on VeilMon for a few architectural functionality (§5.3). To measure the impact of this reliance, we executed SPEC CPU 2006 [117], a well-known collection of system benchmarks, memcached [11], and NGINX [100] inside a native CVM and a Veil CVM. The workloads and settings for memcached and NGINX are provided in Table 5. We noticed negligible difference (<2%) in performance for all three tests. This is because the overwhelming majority of system functionality required through VeilMon, namely booting VCPUs and validating CVM memory regions (§5.3), happens during initialization, not at runtime.

Runtime monitor cost analysis. Excluding the cost of implemented services (which we discuss in §9.2), the runtime cost of any security monitor implementation is the cumulative cost of a domain switch to the monitor (C_{ds}) multiplied by the number of times a domain switch occurs (N_{ds}).

Software security monitors like the Nested Kernel [45] have a very small C_{ds} since they neither require a ring-level switch nor a VM exit. However, the Nested Kernel has a large N_{ds} since it is frequently invoked (e.g., whenever the kernel needs to update its page tables or update control registers). This can result in non-negligible background overhead (e.g., a reported $15-20\%$ bandwidth reduction in some cases [45]). If the Nested Kernel is updated to support memory unmapping in page tables for read protection-based services (the importance of which we discussed in §2), its C_{ds} will also include an expensive TLB flush. Compiler-based monitors [42, 43] already support read protection but they require the expensive kernel CFI, which can reportedly incur more than 50% slowdown for webservers like NGINX [15].

Hypervisor-based monitors like BlackBox [65] do not require frequent context switches (small N_{ds}) since they rely on additional architectural features (e.g., EPT) for memory isolation instead of regular page tables. In CVM contexts, the difference in C_{ds} between VeilMon and a hypervisor-based solution is that the hypervisor C_{ds} is roughly half of the cost of a VeilMon C_{ds}, since it would not need to resume VeilMon's VCPU. However, this additional cost (incurred by Veil) is not significant, given that the alternative requires trusting cloud providers for hypervisor-based monitors.

In contrast to other monitors, while VeilMon's C_{ds} is higher, it still incurs negligible background impact at runtime (previous section) since it's N_{ds} is very low under normal execution. It also has several other advantages like a versatile read and write protection

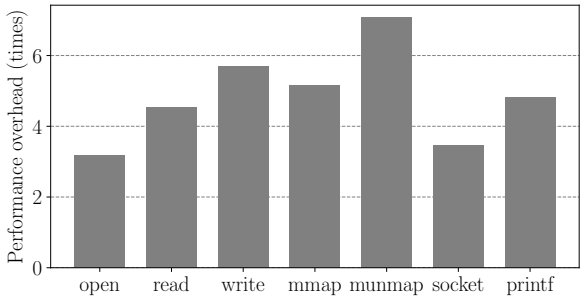

Fig. 4: Cost of redirecting popular system calls from a VEILS-ENC enclave to the outside world (CS2). The system call benchmark parameters are provided in Table 3.

Table 3: Settings for enclave system call benchmarks (Fig. 4).

Benchmark	Parameters
open	Open a text file with read and write permissions
read	Read 10 KB from a file to a memory-mapped region
write	Write 10 KB from a memory-mapped region to a file
mmap	Map a 10KB region using the NULL file descriptor
munmap	Unmap the 10KB region previously-mapped
socket	Open a socket using AF_INET and SOCKSTREAM
printf	Print a "Hello World!" message to the console

scope (§2) and does not require trusting cloud providers. Hence, we believe that VEILMON offers a good trade-off.

9.2 Case Studies on Protected Services

CS1: Secure module load/unload overhead. To measure the performance overhead of module installation when VEILS-KCI is activated, we loaded and unloaded a custom kernel module that prints out a statement to the kernel's debug message log. We chose this small module (of binary size 4728 bytes and final in-memory installed size of 24 kilobytes) since a large module load/unload will already take a long time and the additional VMPL protection update time will become amortized. We repeated the process 100 times and averaged results. We measured an average increase of 55k cycles at load and unload. It was similar for load and unload, since the additional steps (adjusting permissions using `RMPADJUST`) required are the same. This resulted in a 5.7% increase in load time and 4.2% increase in unload time, which is a small per-module cost to pay for kernel code integrity.

CS2: Enclave system call and runtime overhead. We measured the runtime overhead of enclaves using a system call benchmark and several real-world programs.

Fig. 4 shows enclave performance on common system calls related to file system, memory allocation, network, and console messages. On our machine, we ran these natively and inside an enclave for 10,000 iterations. Predictably, system calls in enclave contexts are between 3.3 – 7.1× slower since they require two costly domain switches (from Dom_{ENC} to Dom_{UNT} and back) and system call argument copies (§6.2). This is also true for other enclaves. For

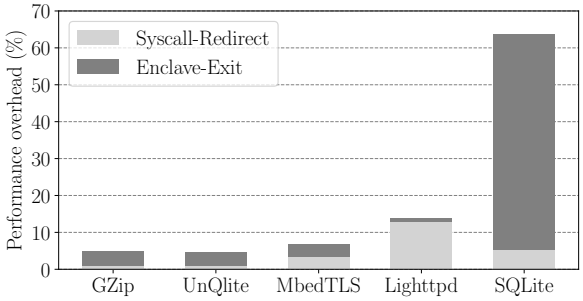

Fig. 5: Performance overhead incurred while shielding real-world programs using VEILS-ENC (CS2). The combined cost of the stacked bars is the complete overhead incurred by the application inside an enclave. From left to right, the enclave exit rate/second was 0.08k, 35.5k, 9.3k, 4.8k, and 22.4k.

Table 4: Settings for running enclave programs (Fig. 5).

Program	Parameters
GZip	Compressed a 10MB file generated using /dev/urandom
SQLite	Inserted 10k random entries into a test database
UnQlite	Ran provided *huge-db* test which inserts 1 million random entries into a test database
MbedTLS	Ran provided a self-test benchmark which executes 2.8k tests for AES, SHA, RSA, ChaCha etc.
Lighttpd	Ran locally with 1 worker thread and benchmarked using ApacheBench (ab) [121] for 10,000 (10KB) files

instance, Virtual Ghost enclaves incur 4.7× times performance overhead on file system benchmarks [42], while SGX with an optimized library operating system still incurs at least 4× slowdown for the read system call [36]. This cost becomes amortized during enclave execution, especially when system calls are infrequent.

We also used our SDK (§7) to port 5 real-world programs that can benefit from enclave protection. They include a webserver (lighttpd [86]), two databases (SQLite [118] and UnQLite [12]), a cryptographic program (MbedTLS [10]), and a compression engine (Gzip [3]). Each program required ~200 lines of code changes to enable enclave initialization, as well as configuration changes to build statically-linked binaries. Since many of these programs have tens of thousands of code lines, we believe this porting effort is minor. Note that the official SGX SDK [14] requires developers to manually specify pointers and lengths for system calls, a considerably more complex undertaking. Additionally, in the future, enclave initialization can be automated by changing the startup functions of musl-libc (e.g., `__libc_start_main`). Table 4 shows the settings and workloads for each program.

Fig. 5 shows the average performance slowdown of enclave protection for evaluated programs under 10 runs. We observed performance overheads from 4.9% to 63.9%. Since enclave slowdown is due to system call redirection and enclave exits, we divide the bar into overhead incurred by source. In general, we notice that enclave exit cost dominates, except when very large regions are copied at system calls (e.g., lighttpd must copy 10kB pages outside the enclave

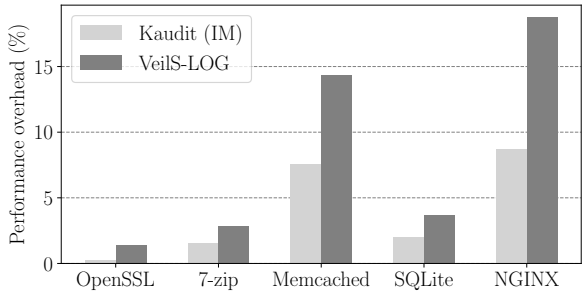

Fig. 6: Performance overhead while auditing different real-world programs using VeilS-Log (CS3). From left to right, the log rate/second was 1.5k, 1.8k, 61k, 2.3k, and 38k.

Table 5: Settings for auditing real-world programs (Fig. 6).

Program	Parameters
OpenSSL	Phoronix benchmark: pts/openssl [108]
7-Zip	Phoronix benchmark: pts/compress-7zip [107]
Memcached	Ran locally with 4 worker threads and benchmarked using memaslap [134] with 90:10 GET:SET split for 60s and a concurrency level of 16
SQLite	Phoronix benchmark: pts/sqlite-speedtest [109]
NGINX	Ran locally with 2 worker threads and benchmarked using ApacheBench (ab) [121] for 10,000 (10KB) files

on client requests). This is expected given the performance overhead we observed in the system call benchmarks (previous section). It is also expected that enclave overhead is dependent on enclave exit rate. In particular, SQLite which incurred ~36k exits/second incurred the highest overhead. The cost of enclave exits can be reduced by implementing *exitless* handling [29, 101, 116].

In general, VeilS-Enc's cost is modest and comparable to other enclave systems (e.g., SGX). Hence, we find it a promising solution to address the critical problem of running sensitive computations in CVMs with untrusted operating systems.

CS3: Secure system call auditing overhead. We compared the performance of VeilS-Log's protection with the native Linux system audit framework (Kaudit [119]) using real-world programs. We ran 5 programs—NGINX, Memcached, OpenSSL, 7-Zip, and SQLite.

The test parameters and benchmarks for each application are provided in Table 5. We configured the CVM (using the Linux auditctl [2] command) to log system calls[1] based on the ruleset used by prior work [21, 103, 104]. The ruleset includes important file creation, network access, and process execution calls.

We made one change to Kaudit to ensure fair comparison. In particular, natively Kaudit uses a user-space component called Auditd to write logs to disk. This component is known to be very inefficient [90], and is different from VeilS-Log which keeps logs in-memory. Hence, we modified Kaudit to keep logs in-memory too for both experiments.

[1]read, readv, write, writev, sendto, recvfrom, sendmsg, recvmsg, mmap, mprotect, link, symlink, clone, fork, vfork, execve, open, close, creat, openat, mknodat, mknod, dup, dup2, dup3, bind, accept, accept4, connect, rename, setuid, setreuid, setresuid, chmod, fchmod, pipe, pipe2, truncate, ftruncate, sendfile, unlink, unlinkat, socketpair, splice.

Fig. 6 shows the incurred overhead for Kaudit and VeilS-Kci over native execution. VeilS-Kci incurred a performance overhead of 1.4% to 18.7% while Kaudit incurred an overhead of 0.3% to 8.7%, compared to native execution. This performance gap is not very high, even under the very high log production rates of tested programs, and it shows that VeilS-Kci is suitable for system logging.

9.3 Key takeaways

Veil's protected services incur modest performance overhead, which is comparable to other widely-deployed systems (e.g., SGX). When a protected service is not used, Veil incurs no discernable performance overhead. Hence, we believe that Veil can be readily-adopted to secure today's CVMs.

10 Discussion and Future Work

VeilS-ENC and other enclave solutions. VeilS-Enc's abstractions are inspired by SGX. Like SGX, VeilS-Enc divides the process into untrusted and enclave regions, while ensuring enclave memory cannot be shared with any other software. Moreover, while SGX allows the operating system to maintain an enclave's page table unlike VeilS-Enc, the latter still allows the OS to securely make changes to the page tables (e.g., for collaborative demand paging) and manage enclave memory like SGX (§6.2).

Although its abstractions are functionally-equivalent to SGX and others [41], VeilS-Enc offers a more flexible tiered security approach than alternatives. In particular, SGX-like approaches only enable protection for computations inside enclaves; hence, enclaves must be used for all programs that require any degree of protection. With Veil, users can leverage native CVM protections (against untrusted hypervisors) for programs that are not highly sensitive, while only relying on VeilS-Enc for highly sensitive programs (e.g., servicing personally-identifiable information). This gives users more control over the security-performance trade-off.

Another advantage of VeilS-Enc is that it can be flexibly molded into non-SGX enclave models depending on user scenario. For instance, Chancel [18] leverages expensive compiler software fault isolation (SFI) to securely share a single SGX enclave's memory for multi-client applications. In contrast, VeilS-Enc can modify enclave page tables to securely and efficiently share memory regions between two mutually-trusting enclave processes. Additionally, since VeilS-Enc executes at a privileged mode (unlike SGX enclaves), it can leverage CPU features like MPK for fine-grained intra-enclave component isolation [20, 106]. Finally, like eOPF [22], VeilS-Enc can leverage privileged instructions (e.g., `WBINVD`) to isolate and invalidate CPU structures and defeat enclave side-channels.

System call batching. A significant cost for enclave solutions (including VeilS-Enc) are synchronous system call exits [29, 101] since the enclave must incur a high exit cost and busy-wait while a system call is handled (§9.2). One way to minimize synchronous exits is by batching system calls and leveraging free background threads to process the batched calls [116]. This optimization can be incorporated for VeilS-Enc alongside multi-threaded enclave support (§7) to improve performance. We leave this to future work.

Library OS (LibOS) integration. LibOSs offer robust system call support and other advantages like fully-containerized filesystems to enclaves [32]. Given the functional equivalence of VeilS-Enc

and SGX, the best integration choice for the former is an SGX LibOS (e.g., Graphene [36]). The main porting effort in this case would be writing a custom platform abstraction layer that would transform SGX commands and instructions into VEIL-specific requests. For instance, the SGX entry instruction (`EENTER`) would become a hypervisor-relayed domain switch request (§5.2). Employing VEIL with LibOSs can benefit from fast process-level startup techniques, such as on-demand fork [131], and fast in-process sandbox [106] to overcome VM and kernel overheads.

11 Related work

Concurrent VMPL research. Concurrent with this work, two other recent systems–SVSM and Hecate–leverageVMPL.

AMD released a secure VM service module (SVSM) [24] written in Rust to provide migration and a virtual TPM to CVMs. SVSM considers a different threat model than Veil's, where the CVM OS is trusted but offloads some functionality to a higher privileged software for simplicity. This module also uses VMPL protections, but it does not support VEIL's services or domain isolation. In the future, we plan to integrate VEIL and SVSM, to bring the benefits of Rust to VEIL and extend SVSM with VEIL's services.

Hecate [53] leverages VMPL to securely *lift-and-shift* legacy VMs to SEV-SNP. In particular, Hecate implements a monitor at VMPL-0 that intercepts the legacy VM's system interactions and transparently translates them into SEV-SNP compatible operations. In addition, the monitor optionally protects the kernel from malicious network traffic and implements kernel code integrity. However, unlike Veil, Hecate does not leverage its design to enable isolated security services or enclave abstractions to protect trusted applications if the guest OS becomes compromised.

Kernel and hypervisor security monitors. SILVER [128] and UCON [129] provide VM monitor-enforced access control policies on sensitive kernel structures. Nooks [120] and LVDs [99] isolate device drivers from the core kernel using MMU protections and lightweight virtualization (`VMFUNC`), respectively. IskiOS [60] leverages Intel MPK to create isolated shadow stacks. Many of the isolation targets of these systems can be used by future VEIL services to enable additional kernel security.

In addition to monitors for operating systems, researchers have proposed security monitors for cloud hypervisors. HypSec [85] protects VMs from large buggy hypervisors by introducing a minimal core hypervisor. Nexen [114] leverages the Nested Kernel principles to create a protected Xen hypervisor that prevents a wide-range of known hypervisor vulnerabilities [50]. Like hypervisor-based monitors (§2), these solutions are also incompatible with CVMs since they do not trust any software outside the CVM.

Shielded program execution. Many systems protect computations from an untrusted operating system [18, 28, 40–42, 46, 48, 62, 65, 66, 77, 92]. VEILS-ENC leverages techniques used in these systems, while maintaining compatibility with CVMs. vSGX [132] is the only other SEV system that shields programs from the operating system. vSGX allows a single computation to run inside an enclave CVM, while redirecting system calls to an untrusted CVM. Hence, each computation needs its own CVM, which is wasteful not just in terms of memory, but also because a platform can only run a limited amount of CVMs [26]. In contrast, VEILS-ENC can enable potentially unlimited enclaves inside a single CVM.

SEV attacks and defenses. SEV was found vulnerable to various attacks, and is patched against many of these attacks in SEV-SNP [23]. In particular, the earliest version of SEV kept CVM register state unprotected in hypervisor memory, allowing an attacker to compromise CVM integrity [64] or fingerprint programs running inside CVMs [125]. Attacks utilizing the ciphertext side channels [80, 83, 126] and insecure I/O implementations [82] of SEV were also found. SEV-SNP mitigates these problems by saving register state in protected CVM memory and disallowing hypervisor access to encrypted memory regions. However, SEV is still vulnerable to memory side channels [79, 81, 82, 84, 89, 95] and controlled channels [124, 130]. Many software mitigations have been proposed for these attacks, such as trying to detect attacks [39, 61, 115], isolating shared resources (e.g., cache) [22, 54, 70, 105, 113, 133], adding noise to timer readings to make it imprecise [91, 102], and applying cryptographic memory randomization [17, 19, 74, 110].

User-level isolation in privileged modes. Lord of the x86 Rings [78] is a portable approach for user-space privilege isolation by leveraging intermediate x86 rings (ring 1 and 2). vTZ [67] leverages ARM TrustZone to create a co-running secure VM for each guest, where trusted guest programs execute. The former solution requires trusting the operating system, while the latter requires a hardware layer that is not supported in the CVM threat model. TrustZone-based systems also suffer from other problems including controlled channel attacks as outlined by prior work [35].

Secure system auditing. Existing research prevents the operating system from tampering with system logs using tamper-evident hashes [69, 103, 104], an external hardware device [21], or trusted virtualization extensions [52]. Tamper-evident hashes only guarantee log integrity verification, and some implementations [69, 103] require a protected execution environment (e.g., SGX) to securely keep these hashes. External hardware devices are incompatible with CVMs since devices cannot be securely queried and virtualization layers are occupied by the untrusted cloud hypervisor.

12 Conclusion

VEIL is a CVM security monitor framework that efficiently enables a wide-range of protected services—from kernel code integrity to shielded program execution—in the presence of an untrusted operating system. Our implementation shows that CVMs can support VEIL with minor changes, while incurring modest performance overheads for using protected services.

Acknowledgment

We thank the anonymous reviewers and our shepherd for their helpful feedback. This work was partly supported by the National Science Foundation (NSF) under the grant CNS-2145888.

References

[1] AMDESE/AMDSEV: AMD Secure Encrypted Virtualization. https://github.com/AMDESE/AMDSEV.
[2] auditctl(8) – Linux Manpage. https://linux.die.net/man/8/auditctl/.
[3] Gzip – GNU Project Free Software Foundation. https://www.gnu.org/software/gzip/.
[4] Linux Kernel CVEs | All CVEs. https://www.linuxkernelcves.com/cves.
[5] Linux Test Project. https://github.com/linux-test-project/ltp.

[6] Linux Test Project: ltp/testcases/kernel. https://github.com/linux-test-project/ltp/tree/master/testcases/kernel.

[7] Linux Test Project: ltp/testcases/kernel/device-drivers. https://github.com/linux-test-project/ltp/tree/master/testcases/kernel/device-drivers.

[8] Linux Test Project: ltp/testcases/kernel/syscalls. https://github.com/linux-test-project/ltp/tree/master/testcases/kernel/syscalls.

[9] Linux Test Project: ltp/testcases/kernel/tracing. https://github.com/linux-test-project/ltp/tree/master/testcases/kernel/tracing.

[10] mbedtls. https://tls.mbed.org.

[11] Memcached - A Distributed Memory Object Caching System. https://memcached.org/.

[12] UnQLite – An Embedded NoSQL Database Engine. https://unqlite.org/.

[13] Virtio - KVM. https://www.linux-kvm.org/page/Virtio.

[14] 01org. Intel(R) Software Guard Extensions for Linux* OS (source code). https://github.com/01org/linux-sgx.

[15] Abubakar, M., Ahmad, A., Fonseca, P., and Xu, D. SHARD: Fine-Grained Kernel Specialization with Context-Aware Hardening. In *Proceedings of the 30th USENIX Security Symposium (Security)* (Virtual Event, Aug. 2021).

[16] Accetta, M. J., Baron, R. V., Bolosky, W. J., Golub, D. B., Rashid, R. F., Tevanian, A., and Young, M. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC)* (Boston, MA, June 2010).

[17] Ahmad, A., Joe, B., Xiao, Y., Zhang, Y., Shin, I., and Lee, B. Obfuscuro: A Commodity Obfuscation Engine for Intel SGX. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2019).

[18] Ahmad, A., Kim, J., Seo, J., Shin, I., Fonseca, P., and Lee, B. Chancel: Efficient Multi-client Isolation Under Adversarial Programs. In *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS)* (2021).

[19] Ahmad, A., Kim, K., Sarfaraz, M. I., and Lee, B. OBLIVIATE: A Data Oblivious File System for Intel SGX. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)* (February 2018).

[20] Ahmad, A., Lee, S., Fonseca, P., and Lee, B. Kard: Lightweight Data Race Detection with Per-thread Memory Protection. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Virtual Event, Apr. 2021).

[21] Ahmad, A., Lee, S., and Peinado, M. Hardlog: Practical Tamper-Proof System Auditing Using a Novel Audit Device. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (Oakland)* (May 2022).

[22] Ahmad, A., Schultz, A., Lee, B., and Fonseca, P. An Extensible Orchestration and Protection Framework for Confidential Cloud Computing. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Jul 2023).

[23] AMD. AMD SEV-SNP: Strengthening SEV with Integrity Protections and More. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

[24] AMD. AMDESE/linux-svsm. https://github.com/AMDESE/linux-svsm.

[25] AMD. SEV-ES Guest-Hypervisor Communication Block Standardization. https://developer.amd.com/wp-content/resources/56421.pdf.

[26] AMD. SEV Secure Nested Paging Firmware ABI Specification. https://www.amd.com/system/files/TechDocs/56860.pdf.

[27] AnandTech. AMD to Launch 3rd Generation EPYC on March 15: Milan with Zen 3. https://www.anandtech.com/show/16537/amd-to-launch-3rd-generation-epyc-on-march-15th-milan-with-zen-3.

[28] ARM. Arm confidential compute architecture. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture, 2022.

[29] Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M., et al. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Savannah, GA, November 2016).

[30] Azab, A. M., Ning, P., Shah, J., Chen, Q., Bhutkar, R., Ganesh, G., Ma, J., and Shen, W. Hypervision Across Worlds: Real-Time Kernel Protection from the ARM TrustZone Secure World. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)* (Scottsdale, Arizona, Nov. 2014).

[31] Azure, M. DCasv5 and ECasv5 Series Confidential VMs. https://learn.microsoft.com/en-us/azure/confidential-computing/confidential-vm-overview.

[32] Baumann, A., Peinado, M., and Hunt, G. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, CO, Oct. 2014).

[33] Buhren, R., Jacob, H.-N., Krachenfels, T., and Seifert, J.-P. One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization. In *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS)* (Virtual Event, Nov. 2021).

[34] Bulck, J. V., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proceedings of the 27th USENIX Security Symposium (Security)* (August 2018).

[35] Cerdeira, D., Santos, N., Fonseca, P., and Pinto, S. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)* (San Francisco, CA, May 2020).

[36] che Tsai, C., Porter, D. E., and Vij, M. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)* (Santa Clara, CA, July 2017).

[37] Checkoway, S., and Shacham, H. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (March 2013).

[38] Chen, H., Zhang, F., Chen, C., Yang, Z., Chen, R., Zang, B., and Mao, W. Tamper-Resistant Execution in an Untrusted Operating System Using A Virtual Machine Monitor, 2007.

[39] Chen, S., Zhang, X., Reiter, M. K., and Zhang, Y. Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjá Vu. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)* (Dallas, TX, Oct.–Nov. 2017).

[40] Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dwoskin, J., and Ports, D. R. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Seattle, WA, Mar. 2008).

[41] Costan, V., Lebedev, I., and Devadas, S. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proceedings of the 25th USENIX Security Symposium (Security)* (Austin, TX, August 2016).

[42] Criswell, J., Dautenhahn, N., and Adve, V. Virtual Ghost: Protecting Applications from Hostile Operating Systems. In *Proceedings of the 19th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Salt Lake City, UT, Mar. 2014).

[43] Criswell, J., Lenharth, A., Dhurjati, D., and Adve, V. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)* (Stevenson, WA, Oct. 2007).

[44] Cui, R., Zhao, L., and Lie, D. Emilia: Catching Iago in Legacy Code. In *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS)* (Feb. 2021).

[45] Dautenhahn, N., Kasampalis, T., Dietz, W., Criswell, J., and Adve, V. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Istanbul, Turkey, Mar. 2015).

[46] Dong, X., Shen, Z., Criswell, J., Cox, A. L., and Dwarkadas, S. Shielding Software from Privileged Side-Channel Attacks. In *Proceedings of the 27th USENIX Security Symposium (Security)* (Baltimore, MD, Aug 2018).

[47] Enarx. AMD SEV Remote Attestation Protocol. https://enarx.dev/docs/technical/amd-sev-attestation.

[48] Feng, E., Lu, X., Du, D., Yang, B., Jiang, X., Xia, Y., Zang, B., and Chen, H. Scalable Memory Protection in the PENGLAI Enclave. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Jul 2021).

[49] Ferraiuolo, A., Baumann, A., Hawblitzel, C., and Parno, B. Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)* (Shanghai, China, Oct. 2017).

[50] Fonseca, P., Wang, X., and Krishnamurthy, A. MultiNyx: A Multi-Level Abstraction Framework for Systematic Analysis of Hypervisors. In *Proceedings of the 13th European Conference on Computer Systems (EuroSys)* (Porto, Portugal, Apr. 2018).

[51] Fonseca, P., Zhang, K., Wang, X., and Krishnamurthy, A. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)* (Belgrade, Serbia, Apr. 2017).

[52] Gandhi, V., Banerjee, S., Agrawal, A., Ahmad, A., Lee, S., and Peinado, M. Rethinking System Audit Architectures for High Event Coverage and Synchronous Log Availability. In *Proceedings of the 32nd USENIX Security Symposium (Security)* (Anaheim, CA, Aug 2023).

[53] Ge, X., Kuo, H.-C., and Cui, W. Hecate: Lifting and Shifting On-Premises Workloads to an Untrusted Cloud. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA, 2022).

[54] Godfrey, M., and Zulkernine, M. Preventing Cache-Based Side-Channel Attacks in a Cloud Environment. *IEEE Transactions on Cloud Computing* (2014).

[55] Gong, S., Altinbüken, D., Fonseca, P., and Maniatis, P. Snowboard: Finding Kernel Concurrency Bugs through Systematic Inter-Thread Communication

Analysis. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)* (Virtual Event, Oct. 2021).
[56] Gong, S., Peng, D., Altinbüken, D., Fonseca, P., and Maniatis, P. Snowcat: Efficient Kernel Concurrency Testing using a Learned Coverage Predictor. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)* (Koblenz, Germany, Oct. 2023).
[57] Google. google/syzkaller: syzkaller is an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller.
[58] Google. Introducing Google cloud confidential computing with confidential VMs. https://cloud.google.com/blog/products/identity-security/introducing-google-cloud-confidential-computing-with-confidential-vms.
[59] Google Cloud. Confidential computing concepts | Google Cloud. https://cloud.google.com/confidential-computing/confidential-vm/docs/about-cvm.
[60] Gravani, S., Hedayati, M., Criswell, J., and Scott, M. L. Fast Intra-Kernel Isolation and Security with IskiOS. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)* (2021).
[61] Gruss, D., Lettner, J., Schuster, F., Ohrimenko, O., Haller, I., and Costa, M. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *Proceedings of the 27th USENIX Security Symposium (Security)* (Vancouver, BC, 2017).
[62] Guan, L., Liu, P., Xing, X., Ge, X., Zhang, S., Yu, M., and Jaeger, T. TrustShadow: Secure Execution of Unmodified Applications with Arm TrustZone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)* (Niagara Falls, NY, 2017).
[63] Hassan, W. U., Bates, A., and Marino, D. Tactical Provenance Analysis for Endpoint Detection and Response Systems. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)* (San Francisco, CA, May 2020).
[64] Hetzelt, F., and Buhren, R. Security Analysis of Encrypted Virtual Machines. *ACM SIGPLAN Notices* (2017).
[65] Hof, A. V., and Nieh, J. BlackBox: A Container Security Monitor for Protecting Containers on Untrusted Operating Systems. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Carlsbad, CA, July 2022).
[66] Hofmann, O. S., Kim, S., Dunn, A. M., Lee, M. Z., and Witchel, E. InkTag: Secure Applications on an Untrusted Operating System. In *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Houston, TX, Mar. 2013).
[67] Hua, Z., Gu, J., Xia, Y., Chen, H., Zang, B., and Guan, H. vTZ: Virtualizing ARM TrustZone. In *USENIX security symposium* (2017).
[68] Intel. Intel 64 and ia-32 architectures software developer's manual. *Volume 3A: System Programming Guide* (2016).
[69] Karande, V., Bauman, E., Lin, Z., and Khan, L. SGX-Log: Securing System Logs with SGX. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS)* (2017).
[70] Kim, T., Peinado, M., and Mainar-Ruiz, G. STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud. In *Proceedings of the 21st USENIX Security Symposium (Security)* (Bellevue, WA, Aug. 2012).
[71] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)* (Big Sky, MT, Oct. 2009).
[72] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)* (May 2019).
[73] Kuenzer, S., Badoiu, V.-A., Lefeuvre, H., Santhanam, S., Jung, A., Gain, G., Soldani, C., Lupu, C., Teodorescu, S., Raducanu, C., Banu, C., Mathy, L., Deaconescu, R., Raiciu, C., and Huici, F. Unikraft: Fast, Specialized Unikernels the Easy Way. *Proceedings of the Sixteenth European Conference on Computer Systems* (2021).
[74] Le, D. V., Hurtado, L. T., Ahmad, A., Minaei, M., Lee, B., and Kate, A. A Tale of Two Trees: One Writes, and Other Reads. Optimized Oblivious Accesses to Large-Scale Blockchains. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)* (2020).
[75] Lea, D. Dlmalloc, 2010.
[76] Lee, D., Jung, D., Fang, I. T., Tsai, C.-C., and Popa, R. A. An Off-Chip Attack on Hardware Enclaves via the Memory Bus. In *Proceedings of the 29th USENIX Security Symposium (Security)* (Boston, MA, Aug 2020).
[77] Lee, D., Kohlbrenner, D., Shinde, S., Asanović, K., and Song, D. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proceedings of the 15th ACM European Conference on Computer Systems (EuroSys)* (2020).
[78] Lee, H., Song, C., and Kang, B. B. Lord of the x86 Rings: A Portable User Mode Privilege Separation Architecture on x86. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018).
[79] Lee, S., Shih, M., Gera, P., Kim, T., Kim, H., and Peinado, M. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *Proceedings of the 26th USENIX Security Symposium (Security)* (Vancouver, BC, Aug 2017).
[80] Li, M., Wilke, L., Wichelmann, J., Eisenbarth, T., Teodorescu, R., and Zhang, Y. A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP. In *2022 IEEE Symposium on Security and Privacy (SP)* (2022).
[81] Li, M., Zhang, Y., and Lin, Z. Crossline: Breaking "Security-by-crash" based Memory Isolation in AMD SEV. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (2021).
[82] Li, M., Zhang, Y., Lin, Z., and Solihin, Y. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *28th USENIX Security Symposium (USENIX Security)* (2019).
[83] Li, M., Zhang, Y., Wang, H., Li, K., and Cheng, Y. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *30th USENIX Security Symposium (USENIX Security 21)* (2021).
[84] Li, M., Zhang, Y., Wang, H., Li, K., and Cheng, Y. TLB Poisoning Attacks on AMD Secure Encrypted Virtualization. In *Annual Computer Security Applications Conference* (2021).
[85] Li, S.-W., Koh, J. S., and Nieh, J. Protecting Cloud Virtual Machines from Hypervisor and Host Operating System Exploits. In *Proceedings of the 28th USENIX Security Symposium* (2019).
[86] Lighttpd. Lighttpd - fly light. https://www.lighttpd.net/.
[87] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the 27th USENIX Security Symposium (Security)* (July 2018).
[88] Liu, C., Gong, S., and Fonseca, P. KIT: Testing OS-Level Virtualization for Functional Interference Bugs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Vancouver, BC, Apr. 2023).
[89] Liu, F., Yarom, Y., Ge, Q., Heiser, G., and Lee, R. B. Last-Level Cache Side-Channel Attacks Are Practical. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)* (May 2015).
[90] Ma, S., Zhai, J., Kwon, Y., Lee, K. H., Zhang, X., Ciocarlie, G., Gehani, A., Yegneswaran, V., Xu, D., and Jha, S. Kernel-Supported Cost-Effective Audit Logging for Causality Tracking. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)* (Boston, MA, July 2018).
[91] Martin, R., Demme, J., and Sethumadhavan, S. Timewarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-Channel Attacks. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)* (2012).
[92] McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C. V., Shafi, H., Shanbhogue, V., and Savagaonkar, U. R. Innovative Instructions and Software Model For Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)* (June 2013).
[93] Microsoft. Azure Confidential VMs Using SEV-SNP (DCasv5/ECasv5) are Now Generally Available. https://techcommunity.microsoft.com/t5/azure-confidential-computing/azure-confidential-vms-using-sev-snp-dcasv5-ecasv5-are-now/ba-p/3573747.
[94] Microsoft Docs. Virtualization-Based Security (VBS). https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs.
[95] Morbitzer, M., Huber, M., and Horsch, J. Extracting Secrets from Encrypted Virtual Machines. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy* (2019).
[96] Morbitzer, M., Huber, M., Horsch, J., and Wessel, S. Severed: Subverting AMD's Virtual Machine Encryption. In *Proceedings of the 11th European Workshop on Systems Security* (2018).
[97] Murdock, K., Oswald, D., Garcia, F. D., Van Bulck, J., Gruss, D., and Piessens, F. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)* (May 2020).
[98] Musl-Libc. musl-libc, 2017. https://www.musl-libc.org.
[99] Narayanan, V., Huang, Y., Tan, G., Jaeger, T., and Burtsev, A. Lightweight Kernel Isolation with Virtualization and VM Functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)* (2020).
[100] NGINX Inc. NGINX High Performance Load Balancer, Web Server, & Reverse Proxy. https://www.nginx.com.
[101] Orenbach, M., Lifshits, P., Minkin, M., and Silberstein, M. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)* (Belgrade, Serbia, Apr. 2017).
[102] Osvik, D. A., Shamir, A., and Tromer, E. Cache Attacks and Countermeasures: the Case of AES. In *Cryptographers' Track at the RSA Conference* (2006).
[103] Paccagnella, R., Datta, P., Hassan, W. U., Bates, A., Fletcher, C., Miller, A., and Tian, D. CUSTOS: Practical Tamper-Evident Auditing of Operating Systems Using Trusted Execution. In *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2020).

[104] Paccagnella, R., Liao, K., Tian, D., and Bates, A. Logging to the Danger Zone: Race Condition Attacks and Defenses on System Audit Frameworks. In *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)* (Nov. 2020).

[105] Page, D. Partitioned Cache Architecture as a Side-Channel Defence Mechanism. *Cryptology ePrint Archive* (2005).

[106] Peng, D., Liu, C., Palit, T., Fonseca, P., Vahldiek-Oberwagner, A., and Vij, M. uSWITCH: Fast Kernel Context Isolation with Implicit Context Switches. In *2023 IEEE Symposium on Security and Privacy (Oakland)* (San Francisco, CA, 2023).

[107] Phoronix. 7-Zip Compression. https://openbenchmarking.org/test/pts/compress-7zip-1.9.0.

[108] Phoronix. OpenSSL Benchmark. https://openbenchmarking.org/test/pts/openssl.

[109] Phoronix. SQLite SpeedTest Benchmark. https://openbenchmarking.org/test/pts/sqlite-speedtest.

[110] Rane, A., Lin, C., and Tiwari, M. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *Proceedings of the 24th USENIX Security Symposium (Security)* (Washington, DC, Aug. 2015).

[111] Riley, R., Jiang, X., and Xu, D. Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. In *Recent Advances in Intrusion Detection: 11th International Symposium (RAID)* (2008).

[112] Seshadri, A., Luk, M., Qu, N., and Perrig, A. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)* (Stevenson, WA, Oct. 2007).

[113] Shi, J., Song, X., Chen, H., and Zang, B. Limiting Cache-Based Side-Channel in Multi-Tenant Cloud using Dynamic Page Coloring. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)* (2011).

[114] Shi, L., Wu, Y., Xia, Y., Dautenhahn, N., Chen, H., Zang, B., and Li, J. Deconstructing Xen. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2017).

[115] Shih, M.-W., Lee, S., Kim, T., and Peinado, M. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2017).

[116] Soares, L., and Stumm, M. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Vancouver, Canada, Oct. 2010).

[117] SPEC. SPEC CPU 2006. https://www.spec.org/cpu2006/.

[118] SQLite Consortium. SQLite home page.

[119] SUSE. Understanding Linux Audit. https://documentation.suse.com/sles/12-SP4/html/SLES-all/cha-audit-comp.html.

[120] Swift, M. M., Bershad, B. N., and Levy, H. M. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)* (Bolton Landing, NY, Oct. 2003).

[121] The Apache Software Foundation. ab - Apache HTTP Server Benchmark Tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[122] Tsai, C.-C., Arora, K. S., Bandi, N., Jain, B., Jannen, W., John, J., Kalodner, H. A., Kulkarni, V., Oliveira, D., and Porter, D. E. Cooperation and Security Isolation of Library OSes for Multi-Process Applications. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)* (Amsterdam, The Netherlands, Apr. 2014).

[123] Tsai, C.-C., Jain, B., Abdul, N. A., and Porter, D. E. A Study of Modern Linux API Usage and Compatibility: What to Support When You're Supporting. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)* (London, UK, Apr. 2016).

[124] Van Bulck, J., Weichbrodt, N., Kapitza, R., Piessens, F., and Strackx, R. Telling your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *Proceedings of the 26th USENIX Security Symposium (Security)* (August 2017).

[125] Werner, J., Mason, J., Antonakakis, M., Polychronakis, M., and Monrose, F. The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (AsiaCCS)* (2019).

[126] Wilke, L., Wichelmann, J., Morbitzer, M., and Eisenbarth, T. Sevurity: No Security without Integrity: Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *IEEE Symposium on Security and Privacy (Oakland)* (2020).

[127] Xiao, Y., Zhang, X., Zhang, Y., and Teodorescu, R. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *Proceedings of the 25th USENIX Security Symposium (Security)* (August 2016).

[128] Xiong, X., and Liu, P. SILVER: Fine-Grained and Transparent Protection Domain Primitives in Commodity OS Kernel. In *Proceedings of the 16th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)* (2013).

[129] Xu, M., Jiang, X., Sandhu, R., and Zhang, X. Towards a VMM-Based Usage Control Framework for OS Kernel Integrity Protection. In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies (SACMAT)* (2007).

[130] Xu, Y., Cui, W., and Peinado, M. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)* (San Jose, CA, May 2015).

[131] Zhao, K., Gong, S., and Fonseca, P. On-demand-fork: A microsecond fork for memory-intensive and latency-sensitive applications. In *Proceedings of the USENIX European Conference on Computer Systems (EuroSys)* (2021).

[132] Zhao, S., Li, M., Zhang, Y., and Lin, Z. vSGX: Virtualizing SGX Enclaves on AMD SEV. In *2022 IEEE Symposium on Security and Privacy (SP)* (2022), IEEE.

[133] Zhou, Z., Reiter, M. K., and Zhang, Y. A Software Approach to Defeating Side Channels in Last-Level Caches. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)* (Vienna, Austria, Oct. 2016).

[134] Zhuang, M., and Aker, B. memaslap - Load Testing and Benchmarking a Server. http://docs.libmemcached.org/bin/memaslap.html.