An Extensible Orchestration and Protection Framework for Confidential Cloud Computing

Adil Ahmad and Alex Schultz, *Arizona State University;*
Byoungyoung Lee, *Seoul National University;* Pedro Fonseca, *Purdue University*

https://www.usenix.org/conference/osdi23/presentation/ahmad

This paper is included in the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-34-2

Open access to the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

An Extensible Orchestration and Protection Framework for Confidential Cloud Computing

Adil Ahmad[†], *Alex Shultz*[†], *Byoungyoung Lee*[*], *Pedro Fonseca*[§]
[†]Arizona State University [*]Seoul National University [§]*Purdue University*

Abstract

Confidential computing solutions are crucial to address the cloud privacy concerns. Although SGX has witnessed significant adoption in the cloud, the reliance on hardware implementation is restrictive for cloud providers in terms of orchestrating deployments and providing stronger security to their clients' enclaves. eOPF addresses this limitation by providing a comprehensive, secure hypervisor-level instrumentation framework with the ability to monitor all enclave-OS interactions and implement protected services. eOPF overcomes several challenges including bridging the semantic gap between the hypervisor and SGX and attesting the co-location of the framework with enclaves. Using eOPF, we implement two protected services that provide platform resource orchestration and complementary enclave side-channel defense. Our evaluation shows that eOPF incurs very low performance overhead (<2%) in its default state and only modest overhead (geometric mean of 17% on SPEC) when strong, complementary side-channel defenses are enabled, making eOPF an efficient and practical solution for the cloud.

1 Introduction

The previous two decades have shown a substantial growth in internet services enabled by the cloud. Unfortunately, using cloud services requires users to outsource sensitive code, data, or both to cloud infrastructures shared by untrusted individuals. Moreover, the rise in cyber-attacks and corresponding increasing governmental regulations on sensitive information management (e.g., CCPA, GDPR) have made cloud privacy a first-order concern for many cloud providers and users. Thus, the cloud model success increasingly depends on providing strong privacy guarantees.

Cloud providers have been trying to accommodate the user demand for privacy using confidential computing solutions. Such solutions allow secure computation on cloud machines without trusting the machine's huge and vulnerable software codebase like the operating system (OS). Among several approaches, the hardware-protected Intel Software Guard eXtensions (SGX) *enclaves* have turned out to be the most popular key building block. In particular, SGX is already deployed by major cloud providers (e.g., Microsoft Azure [24], IBM Cloud [55]), thanks in no small part due to the extensive software ecosystem (e.g., development kits and library OSs) that aids the development of new SGX programs and porting existing codebases [17, 25, 76, 83].

Despite the strong security properties of SGX, its inflexible hardware implementation poses pragmatic challenges for cloud providers and users. For instance, modern cloud services aim to be elastic, which often comes with a pay-as-you-go model that requires detailed fine-grained resource usage accounting. Unfortunately, SGX only provides detailed enclave-usage data to the OS, which is untrusted even when cloud providers run containerized instances since the OS' large codebase is susceptible to attacks from untrusted users on the machine. Moreover, since SGX's inception, many attacks have been discovered against enclaves, which are currently difficult for cloud providers to mitigate. In particular, hardware updates for several attacks (e.g., digital side-channels) were never implemented by Intel, eroding user trust in the security capabilities of SGX and exposing users to attacks.

This paper proposes eOPF, a framework designed to provide a privileged trusted software environment for cloud providers to deploy secure services on enclave-running platforms. eOPF leverages virtualization extensions to enable trustworthy and complete interposition between enclaves and the OS. By virtue of such interposition, eOPF allows cloud providers to build protected services that enhance enclaves. In particular, this paper shows how eOPF can be used to (a) securely orchestrate enclaves (e.g., control and monitor enclave resource usage) and (b) add complementary enclave side-channel defenses.

Leveraging a framework like eOPF to enable services for enclaves poses several technical challenges. First, to enable protection and resource monitoring, the framework should interpose between the OS and enclave and mediate all OS-enclave interactions. Unfortunately, this capability is not na-

tively available to the virtualization layer. Second, for remote users to trust that their enclaves are protected, they should determine that their enclaves are co-located with the framework. Unfortunately, there is currently no mechanism to guarantee such co-location. Third, even if previous challenges are solved, it is necessary to show how to securely implement orchestration and protection services using the framework.

eOPF achieves complete interposition of all enclave-OS interactions through a combination of hardware-enabled interception features and several indirect mechanisms (§4.1). In particular, Intel CPUs allow a virtualization-based framework to intercept all SGX supervisor instructions, which are used to manage enclave creation and destruction. To reliably trap on all events during enclave execution (e.g., enclave start and stop events), eOPF carefully leverages a combination of memory protection (namely extended page tables), the x86 single-step mode, and interrupt-interception mechanisms.

We address the co-location challenge by designing, to our knowledge, the first platform-enclave co-attestation protocol allowing enclave users to trust that their enclaves are protected (§4.2). Instead of naively leveraging the virtualization framework for enclave installation, which does not prove co-location to a remote user, eOPF leverages a combination of the cloud provider's initial provisioning, intercepted enclave installation, and SGX remote attestation to achieve co-location guarantees for cloud users.

In its current form, eOPF includes a library of functions to allow cloud providers and users to enable several orchestration and protection services (§5). For instance, eOPF implements a library of side-channel defenses that users can select during runtime. The defense capabilities are implemented at a resource-level (e.g., page tables, caches) in a principled manner to isolate resources responsible for side-channel and ensure full protection. Additional services can be flexibly implemented through further software libraries.

We implemented a proof-of-concept eOPF framework with services on the Bareflank extensible framework [3]. In addition, we analyze the end-to-end security of the system and show that eOPF is effective at preventing diverse attacks against its interposition, co-attestation, and implemented services. Furthermore, we demonstrate eOPF's performance (§8) using benchmarks and real-world programs—the SPEC CPU 2006 integer suite [13], Redis [12], and Lighttpd [9]. Our results indicate that the base framework (without side-channel defenses) incurs less than 2% performance impact to enclaves, and when all side-channel defenses are enabled, it incurs a geometric mean performance overhead of 17%, hence suitable for diverse use-cases in today's clouds.

2 Confidential Cloud Computing

This section describes the confidential cloud computing system model, threat model, and research goal of eOPF. Fig. 1 provides an overview of the system model.

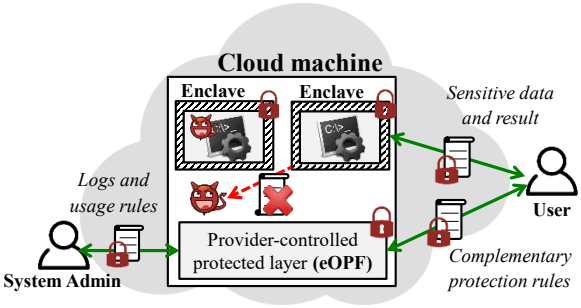


Figure 1: Our confidential computing model.

2.1 System Model

We assume that users want to run sensitive computations on the cloud (e.g., healthcare analytics on genetic information of several individuals [16]). They trust the cloud provider (like other confidential computing approaches [2, 47, 48]) but do not trust other users on the machine. The cloud provider does not trust users and aims to protect users from each other, since some may be malicious.

The cloud provider leverages SGX to enable users to securely run computations in enclaves, without trusting the bulk of the software stack or other users. SGX has several advantages over other approaches. First, SGX provides strong confidentiality and integrity guarantees against a wide-range of attacks [33]. Second, SGX is now widely-available in Intel server machines [4], a sizeable portion of all servers in the market today. Third, there are mature software development kits (SDKs) allowing users to port their programs to SGX enclaves [17] and library operating systems [76, 83] that allow users to easily run legacy programs inside enclaves.

Our model also assumes that the cloud provider runs a type-1 hypervisor on the machine (e.g., AWS Nitro [2], protected KVM [54]) and provisions containerized instances for users. Type-1 hypervisors provide better security guarantees due to a thin software codebase running at the virtualization layer (i.e., Intel VMX [52]). Containerized instances increase resource efficiency and simplify resource provisioning; hence, containerized instances underlay increasingly popular cloud models, such as microservices and serverless computing [1, 11, 72]. Moreover, since users run their sensitive computations inside SGX enclaves, the traditional isolation limitations of container instances do not apply. Nevertheless, our model also directly applies to scenarios where the cloud provider provisions virtual machines (VMs) (§9).

Since the cloud machine runs enclaves of different users, the cloud provider needs to deploy a *flexible, protected* layer to easily manage enclave instances, including managing resource oversubscription (e.g., AWS burstable instances [6]) to maximize resource efficiency. Furthermore, the cloud provider wants to use this layer to offer *enhanced, complementary protection* for enclaves against attacks that SGX does not protect, potentially by charging a higher cost.

2.2 Threat Model and Assumptions

The cloud provider and honest users assume that a dishonest user (or other third-parties) may compromise the machine's operating system, by leveraging a kernel vulnerability or misconfiguration. After OS compromise, they assume that an attacker will launch attacks to (a) steal sensitive information from enclaves using digital side-channels [40, 43, 60, 84, 88] or (b) launch attacks against the platform or other users using enclaves (e.g., to prevent malware introspection [74]).

Assumptions. This heading describes our assumptions about the security of the SGX processor and hypervisor, as well as the availability of a trusted key management service.

SGX processor. We trust that the processor is correctly implemented. In particular, it correctly prevents direct access of enclaves from external software and implements all cryptographic and remote attestation primitives.

Hypervisor. We trust the hypervisor is correct and securely initialized on the cloud machine by the trusted cloud provider. A cloud provider can securely initialize a hypervisor by leveraging UEFI secure boot [87] or verified late launch (e.g., Intel TXT [52, 63]). Leveraging a trusted platform module (TPM) [23], the provider can also attest the correct initialization remotely. Note that although we trust the hypervisor, its compromise cannot harm existing SGX guarantees since enclaves are protected from hypervisors. Please refer to §7.3 for a hypervisor TCB discussion.

Key management service. We assume the availability of a trusted local or remote key management service (KMS). A trusted local key management service can be designed using a TPM. In either scenario, we assume that our system has secure access to the KMS (e.g., using an isolated channel to a local device [89] or authenticated encrypted channel).

Out-of-scope. We do not consider attacks through micro-architectural defects, software vulnerabilities inside enclave programs, system calls, and physical attacks. We also exclude attacks through micro-architectural defects (e.g., speculative execution attacks [29, 56, 85]). Defenses enabled by our system (§5.2) for side-channels also prevent the exploitation of micro-architectural defects [27, 56] in SGX enclaves through these channels. However, the root cause of micro-architectural defects are hardware bugs, and as such they are already routinely addressed by Intel through microcode or hardware updates [7, 51]. Existing schemes [57, 75, 76] can prevent vulnerability exploitation in buggy enclave programs and protect enclaves from malicious system call results [25, 49]. Finally, physical attacks that infer DRAM access patterns and electromagnetic analysis are very expensive [58].

2.3 Research Goal

Given the mistrust of the OS, this paper's research goal is to *design a hypervisor-level instrumentation framework that allows cloud providers to enable protected services on enclave platforms.* The framework is designed to be flexible and support two use-case classes: (a) secure enclave orchestration (e.g., preventing dishonest users from running enclaves, monitoring enclave resource usage) and (b) complementary side-channel defense for enclaves (e.g., by isolating resources).

Combining a hypervisor-level framework with SGX is favorable for cloud providers and users. From a cloud provider's perspective, hypervisor-only approaches [47, 48] offer more control but they require significant investment to design in-house full enclave abstractions and implement the corresponding SDKs. From a user's perspective, hypervisor-only approaches offer flexible functionality (e.g., resource isolation) but they have a single point-of-failure (i.e., the cloud hypervisor) in terms of data protection. Our approach solves both problems by leveraging SGX with its robust software ecosystem [17, 76, 83] and complementary data protection guarantees in the event of a cloud hypervisor compromise. Hence, co-leveraging SGX and a hypervisor is a *best-of-both-worlds* scenario for cloud providers and users.

3 Background on Intel SGX

Intel SGX [64] allows a process to create protected execution contexts called *enclaves*. This section describes memory protection, lifecycle, and remote attestation aspects of SGX since they are relevant to eOPF.

Enclave page cache (EPC). This is a reserved physical memory region where enclaves reside. SGX relies on the operating system to over-subscribe the EPC using *demand paging* (i.e., securely retrieving pages from an encrypted backing store using page faults and updating page tables).

Enclave lifecycle. An enclave is created by the OS using SGX supervisor leaf instructions (`ENCLS`). During enclave execution, the untrusted and enclave parts of the process execute SGX user leaf instructions (`ENCLU`) for a world switch.

Enclave Creation. The OS executes `ECREATE` to create an enclave context. After context creation, the OS invokes `EADD` to copy initial code and data, provided by the user, from non-enclave to enclave pages. Then, the OS executes `EEXTEND` to measure the copied page (explained in the next section). Finally, the OS executes `EINIT` to finalize the enclave.

Enclave Entry/Exit/Resumption. The untrusted part of the process can transition to the enclave mode using `EENTER`. Afterward, the enclave executes `EEXIT` to transition back to the untrusted mode, for two reasons: (a) synchronous exits (i.e., to perform a system-call or shutdown the enclave) and (b) asynchronous exits (i.e., to handle page faults, interrupts, and exceptions). After handling the reason for an exit, the process executes `ERESUME` to resume the enclave.

Remote attestation. SGX enables remote users to assert that their code and initial data is correctly loaded into an enclave

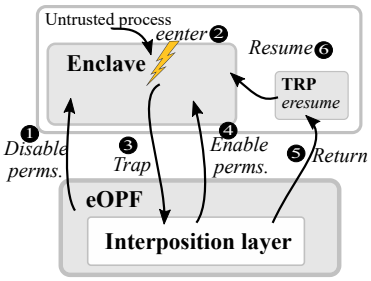
by sending them the enclave measurement (MRENCLAVE or M_e) signed using the SGX CPU's attestation key.

The enclave measurement process is entirely deterministic [30]. The measurement algorithm has an initialization, update, and finalization stage. During initialization (ECREATE), the CPU creates an initial SHA-256 hash using the OS-provided SGX Enclave Control Structure (SECS), which contains the enclave's metadata (e.g., base address and size). In the update stages, the CPU updates the hash using each page added into the enclave (at EADD) alongside an OS-provided security information (SECINFO) block. The SECINFO block contains information about the page's metadata (e.g., offset and permissions). In the same stage, the CPU measures each added page in 512-bit blocks (at EEXTEND). Lastly, in the finalization stage, the enclave's measurement is hashed one last time with the total count of bits that are updated in the MRENCLAVE (at EINIT).

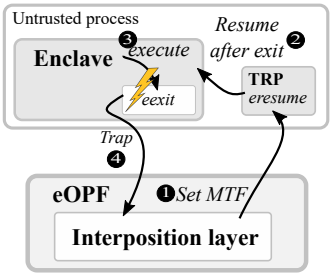
In formal terms, assuming an enclave of N pages (P^1 to P^N) with Z total bits, the entire enclave measurement is:

$$M_e = H_{fin}(H_{upd}(...H_{upd}(H_{upd}(IV, SECS), P^1)..., P^N), Z)$$

In this equation, IV are the initialization vectors. Additionally, for simplicity, we assume that $H_{upd}(state, P)$ also includes a hash of the SECINFO of page P.



(a) EPT-based enclave entry and resume monitor



(b) Single-step-based synchronous enclave exit monitor

Figure 2: eOPF's interposition on enclave entries and synchronous enclave exits.

4 eOPF Design

eOPF provides a privileged trusted software environment for cloud providers to build protected services on their SGX-compatible confidential computing platforms. eOPF leverages hypervisor-level instrumentation to enable trustworthy and complete interposition between enclaves and the OS. This interposition allows users to run protected services that augment enclave security and improve resource management (e.g., measure enclave execution time).

Hypervisor-level or virtual machine extensions (VMX) [52] allow eOPF to monitor and control the execution of the OS, e.g., observe and manipulate page tables. In particular, by leveraging VMX, eOPF can intercept supervisor instructions executed by the OS and exceptions raised by the machine. Moreover, eOPF can also leverage VMX features to protect its TCB from the OS and external devices.

Designing a secure hypervisor-level instrumentation framework for enclaves poses several challenges that we address:

C1: Semantic VMX-SGX gap. Complete and reliable interposition of enclave interactions is needed to build protected services. While VMX framework can natively trap on SGX supervisor instructions for enclave management, it cannot natively trap SGX user instructions that determine when an enclave starts or stops. These latter events are typically junctions of information exchange between enclaves and the OS; hence, interposition is critical to augment enclave security.

C2: Co-location attestation hurdle. SGX's remote attestation allows a remote user to know that their programs are running inside an enclave, but provides no guarantees that this enclave is running on the cloud provider's machine. Without such guarantees, users cannot tell that their enclaves are protected by eOPF.

C3: Practical service libraries. It is necessary to show how to leverage the enclave instrumentation framework to build protected services. To help users easily build such services, it is necessary to design and implement easy-to-use libraries with core functions (e.g., transparently augment enclave protections against classes of attacks).

4.1 Enclave Life-Cycle Interposition

eOPF achieves complete interposition over all interactions between an enclave and the OS using native x86 features and new indirect interposition mechanisms.

Enclave management monitor. eOPF leverages the native capabilities of x86 virtualization to trap all SGX supervisor instructions (ENCLS), which are used for enclave creation, deletion, and other management tasks. In particular, eOPF sets the ENCLS-interception bit and its corresponding instruction bitmap in the x86 Virtual Machine Control Structure (VMCS) [52] to trap ENCLS instructions. On a trap, eOPF undertakes three sequential steps. First, eOPF implements service-specific operations needed for the instruction (refer to §4.2 and §5.1). Second, eOPF executes the trapped instruction using its trusted code and parameters provided by the

OS. Third, eOPF resumes the OS' execution from after the instruction by updating the processor's program counter.

EPT-enforced enclave entry and resume monitor. eOPF tracks enclave entry and resume events using the extended page tables (EPT). EPT allows a virtualization framework to protect regions of the physical memory from unauthorized read, write, and execute operations. By removing execute permissions from the enclave page cache (EPC) region, eOPF can ensure that every time enclave code is executed it raises a trap. The challenge, however, is that the trap is raised as an enclave exit, and there is no guarantee that the OS will resume the enclave after eOPF resolves the trap.

eOPF addresses the challenge by creating a trusted resume pointer (TRP), a reserved location within the process' address space that is guaranteed to execute `ERESUME`. eOPF inserts the TRP at a location where it does not significantly impact the OS' process memory management (i.e., only shares top-level page table with the remaining addresses). The OS is notified through a shared memory channel and kernel module (§6) to reserve the TRP region. eOPF write-protects the TRP and page tables that address this location using EPT, ensuring the TRP cannot be modified by the OS.

Fig. 2-(a) illustrates the EPT-based enclave entry and resume scheme employed by eOPF. On every processor core, eOPF leverages the EPT to disable execute permissions for all enclave page cache (EPC) regions (❶). Hence, when a process transitions into the enclave region (i.e., using `EENTER` or `ERESUME`) (❷), the CPU traps the operation with an EPT violation (❸). eOPF resolves the violation by enabling execution permissions (❹) and redirecting the program counter (`rip`) to the TRP (❺). Finally, the enclave resumes (❻).

Dual enclave exit monitors. eOPF uses the x86 single step mode and interrupt interception features to track synchronous and asynchronous enclave exits, respectively.

Single-step-based synchronous exit monitor. eOPF leverages the x86 single step mode to trap synchronous exits (e.g., for an exit-based system call). In particular, since system software is not allowed to intercept enclave execution apart from debug mode, the execution (from `EENTER` to `EEXIT`) within an enclave is considered a single step [32].

Fig. 2-(b) illustrates the synchronous exit monitor process during an exit-based enclave system call. eOPF enables an the single-step mode by setting the MTF in the current processor's VMCS (❶) before entering the enclave (❷). Hence, the processor's execution traps to eOPF's monitor when the enclave executes `EEXIT` (❸∼❹). eOPF disables this trap allowing the exit to be processed by the system. This process is repeated at the next enclave entry.

Interrupt-based asynchronous exit monitor. Apart from synchronous exits, the enclave performs asynchronous exits in order to service interrupts (e.g., raised by the timer hardware). eOPF ensures that all interrupts are trapped by setting the interrupt-interception bit inside the VMCS.

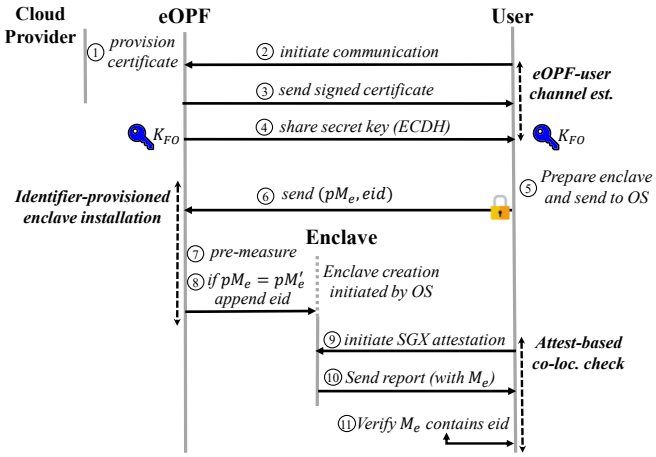


Figure 3: The platform-enclave co-attestation protocol.

4.2 Platform-Enclave Co-Attestation

SGX's attestation does not tell a user that their enclaves are running on their cloud provider's machine. In particular, the attestation report only contains information about the platform's security version (microcode) [53]. This is a significant challenge that motivates eOPF's co-attestation protocol.

Without a guarantee of co-location between enclaves and eOPF, an attacker (e.g., a malicious user) who has compromised a cloud machine's operating system could trick users into sending data to enclaves unprotected by eOPF. In particular, the attacker could exfiltrate a user's code from a cloud machine, send it to their own SGX-capable machine, and install it inside an unprotected enclave. Afterwards, the attacker could route all network traffic from the cloud machine to their own machine, and trick the user into sending their confidential data to the unprotected enclave. This potential attack would not require collusion with the VMM, since VMMs must allow network traffic to a cloud user's machine.

A naive approach to prevent this attack would be to leverage eOPF's interposition (last section) and design an entirely new in-house SGX attestation approach. Unfortunately, that is a significant undertaking that would require complex attestation functionality to be redundantly re-implemented and make enclave security completely reliant on eOPF, instead of complementary to SGX protections.

eOPF implements a more secure, novel co-attestation protocol that leverages both eOPF's interposition and SGX remote attestation. The key insight of our approach is that eOPF's interposition allows it to bind an *infeasible-to-guess* secret (as a watermark) to a user's enclave created on its machine, which will be transmitted and validated through SGX remote attestation to the remote user.

The eOPF co-attestation protocol has three stages. First, with the help of the trusted cloud provider, a remote user establishes a secure communication channel with an eOPF instance. Through this channel, the user sends a secret to this eOPF instance. Second, during enclave creation, the eOPF

instance securely and transparently inserts that secret into initial enclave memory. Third, the remote user leverages SGX attestation to verify the initial enclave contents and validate that co-attestation secret is valid, thereby confirming that the enclave is co-located with a cloud provider eOPF instance. In all these steps, eOPF uses a side-channel resistant cryptographic library (e.g., EverCrypt [69]) to protect secret keys. Fig. 3 illustrates our co-attestation protocol.

eOPF-user channel establishment. With the provider's help, an eOPF instance on a cloud machine and a user of the machine establish a secure communication channel. In particular, during initial platform provisioning, the cloud provider installs the eOPF framework on the machine with a signed digital attestation certificate (①). This certificate and the corresponding private key is securely stored by eOPF using a trusted key management service (e.g., protected storage device or a trusted platform module) (§2.2).

When a remote user wants to run enclaves on the machine, the user will first establish a secure communication channel with the eOPF framework (②–④). In particular, the user asks the framework to authenticate itself (②) and the framework responds with its signed certificate (③). If the certificate signature is valid—based on the cloud provider's off-the-band provided public key ($PubK_C$)—the remote user and the eOPF instance establish a shared secret key (K_{FO}) (④). Note that eOPF does not require direct access to the network. In particular, all eOPF-user communication can be routed through the operating system. This approach is safe since all communication after shared channel establishment is end-to-end encrypted (using K_{FO}) and is similar to how enclaves use the operating system as an untrusted network transport.

Identifier-provisioned enclave installation. Once a secure channel between the user and an eOPF instance is established, the eOPF instance installs a secret identifier into a user-specified enclave during enclave creation. We explain this process in the next paragraphs.

The user compiles a special enclave binary with one empty reserved memory page (4KB) at the end using a custom linker script and sends it to the OS. The reserved memory page will be used to hold a random 4KB secret (called *eid*). The user also creates a *premeasurement* (pM_e) of this enclave binary. The pM_e is a hash of all enclave binary pages using SGX's enclave measurement algorithm (described in §3) except the last reserved page. In formal terms, assuming the enclave has N pages (P^1 to P^N), the pM_e is calculated as follows:

$$pM_e = H_{upd}(...H_{upd}(H_{upd}(IV, SECS), P^1)..., P^{N-1})$$

The user sends the enclave binary to the OS (⑤). Simultaneously, the user sends the pM_e and *eid* to eOPF using their secure communication channel (⑥).

During enclave creation, eOPF recreates pM_e to attest that the correct user enclave is being initialized on the machine (⑦). In particular, on enclave creation (§4.1), eOPF recreates the hash using an internal SHA-256 library configured with the OS-provided parameters to `ECREATE` and `EADD` instructions (i.e., SECS, SECINFO, and page contents). If the premeasurement matches pM_e, eOPF transparently modifies the last enclave page to include the *eid* (⑧). This requires trapping `EADD` and replacing the contents inside the physical page being added to the enclave.

There are three requirements for the above operations to securely happen. First, the OS should not modify an enclave page while it is being measured by eOPF. Second, the OS should not read the *eid* while it is being copied into the enclave. Third, after copying *eid*, there should be no additional pages added to the enclave. eOPF fulfills the first two requirements using EPT. In particular, eOPF write-protects the SECINFO and page contents of the enclave page before the premeasurement process. Similarly, while adding *eid* to the reserved page, eOPF removes all permissions from the page before executing `EADD`. These protections are only disabled after `EADD` executes (§4.1). Finally, eOPF does not allow any `EADD` operation on the enclave after adding *eid*, ensuring that it really is the user's enclave, and not a malicious enclave designed by the OS to steal *eid*.

Please refer to §9 for a discussion on how this co-attestation step can be potentially achieved without premeasurement.

Attestation-based co-location check. Once the enclave is securely provisioned with a secret identifier (*eid*) that is only known to the eOPF instance, a remote user can leverage SGX remote attestation (§3) to check whether their enclave contains that identifier or not (⑨∼⑪). In formal terms, assuming a correct enclave page with *eid* is P^{eid} and Z total bits, the correct enclave measurement should be as follows:

$$M_e = H_{fin}(H_{upd}(pM_e, P^{eid}), Z)$$

Since *eid* is a 4KB identifier, there is an infinitesimally small chance for an attacker to randomly guess (2^{-32768}); hence, this measurement can only hold if the enclave memory contains *eid* provisioned by eOPF, proving co-location.

5 eOPF Protected Services

eOPF allows cloud providers to implement protected services in an extensible manner. This section demonstrates eOPF's value by presenting the design of two services that help cloud providers manage resources and augment enclave security.

5.1 Secure Enclave Orchestration

The secure enclave orchestration service gives cloud providers the ability to control what enclaves run on their platform, detect when enclaves are used for malicious purposes, and obtain detailed enclave-related resource usage for accountability and billing. This section describes how eOPF allows cloud providers to achieve such orchestration.

Protected launch control. eOPF ensures that only users approved by the cloud provider are allowed to run enclaves on

cloud machines. In modern SGX machines, cloud providers leverage flexible launch control (FLC) [53] to provision a platform and provide launch tokens to their customers without relying on Intel's provisioning service. Unfortunately, flexible launch is controlled by the untrusted OS using MSRs, `IA32_SGXLEPUBKEYHASH{0-3}` and the attacker can exploit this feature to launch arbitrary enclaves. eOPF leverages virtualization features to trap all writes to MSRs (i.e., `WRMSR`) and disallows modifications to launch control MSRs. Hence, all valid changes to the FLC feature must come from the cloud provider directly to eOPF.

Malware scanning. An attacker may try to hide malware on the cloud machine using shielded environments like enclaves [39,74]. For instance, research shows that attackers can use TPMs to hide attack targets from forensic analysts [39]. A typical approach to detect malware on a machine is by scanning binaries and signature matching against a database of known malware. Although a simple approach, this is effective in practice (e.g., one study shows 59% of known malware can be detected by signature matching tools [81]).

eOPF enables secure scanning of enclave contents within its framework during enclave creation. In particular, during enclave creation, as each page is being added to the enclave (§4.1), eOPF compares the hash of contents against known malware hashes. eOPF also provides the ability to prevent the attacker from installing a barebones enclave and leveraging it to insert malware (e.g., by enabling execute permissions on data pages). This is achieved by intercepting and rejecting `EMODPE`, an `ENCLS` leaf instruction leveraged for changing existing enclave page permission changes and adding additional pages.

One concern with enclave content scanning is user privacy especially in scenarios where enclave code is an intellectual property (e.g., services like 23andMe [16] with proprietary healthcare analysis algorithms). Such concerns can be mitigated if the cloud provider runs their scanning tool inside an enclave and makes the source code of the scanner publicly-available for enclave attestation by remote users. If a proprietary scanner is used, the provider can employ SGX sandbox enforcement mechanisms [19,49]. With these, users can trust that the proprietary scanning tool will be unable to leak sensitive information from the scanning enclave.

Resource usage statistics. Once an allowed enclave is running on the machine, eOPF collects detailed statistics about the enclave's machine resource usage and periodically sends it to a system administrator.

By default, eOPF collects information about two resources: CPU time and memory. In particular, eOPF collects how much time (in cycles using `RDTSCP`) is dedicated to the user's enclaves by implementing timers at enclave entries and exits. To prevent the OS from modifying CPU timer information, eOPF disallows all changes to timer-related MSRs [52]. eOPF also collects how much memory is allocated to the enclave. This is achieved by monitoring enclave page addition (`EADD`) and enclave page removal (`EWB`) instructions.

If a user enables complementary enclave side-channel defense, enclaves use additional resources (§5.2). eOPF also collects statistics of such usage for reporting purposes. In particular, eOPF tracks whether hyperthreading is disabled on a CPU core to defeat per-core side-channels. If the user selects static memory allocation for paging side-channel defense, this information is also collected. Finally, eOPF reports whether the enclave is using an isolated last-level cache or not, and how many partitions within the LLC are reserved for the user.

5.2 Complementary Side-Channel Defense

This service allows users to enable complementary principled defenses against digital side-channels. Digital side-channel attacks allow untrusted software on a machine (e.g., the OS) to observe the interactions of trusted software and the hardware platform [70]. Observation allow attackers to infer memory access patterns of an enclave program, which has been shown to leak sensitive enclave data (e.g., cryptographic keys) because many programs have data-dependent pathways [26].

To reason about defeating side-channels, we divide hardware resources based on how they can be observed (hence, exploited) into cross-core and per-core resources. For instance, last-level cache is shared by all processor cores, hence it can be observed by attacker on any core, while the L1/L2 caches are private to each processor core and can only be observed if the attacker runs code within the same core.

Using our classification and by integrating techniques from literature [60,61,66,67], this service offers principled side-channel defense that can be flexibly enabled by users with minimal effort. In particular, the service isolates cross-core resources, ensuring an attacker cannot simultaneously observe enclave access onto the resource from any other core. Moreover, the service invalidates or deactivates per-core resources to ensure an attacker is unable to observe enclave access semantic when they run sequentially on a processor core after an enclave, or parrallely on an enclave-running core.

5.2.1 Cross-Core Resource Isolation

Page tables. The page table is created and maintained by the untrusted OS. The OS can infer page-granular (4KB for SGX enclaves) access patterns of an enclave through an enclave's page tables. In particular, the OS can modify the enclave's page tables to induce page faults [88] or stealthily observe the access bits of the enclave's page table entries [84]. To avoid these attacks, eOPF allows the OS to create and delete the page tables, at enclave creation and deletion, respectively. However, eOPF prevents modifications to the page tables during enclave execution. Therefore, eOPF employs *temporal isolation* to protect the page tables.

After enclave creation, eOPF write-protects an enclave's page tables using EPT. During each enclave entry, eOPF also checks the `CR3` value to ensure that the OS did not try to create duplicated enclave page tables. Hence, eOPF ensures that the attacker cannot induce enclave page faults during execution. Furthermore, eOPF scans the enclave's page tables and sets the access bit of each entry, ensuring that the attacker cannot leak information through access bits. At enclave shutdown, eOPF disables write-protection to let the OS handle page table deallocation. Please refer to §9 as to how this defense can be extended to support oblivious page swapping.

Last-level cache (LLC). The LLC contains cache lines from all programs executing on all processor cores. Hence, the LLC is vulnerable to cache attacks [26,74]. To defeat these attacks, eOPF partitions the LLC such that an enclave's cache lines are *spatially isolated* from untrusted programs.

Cache Allocation Technology (CAT) allows isolating cache lines of different CPU processors across different partitions in the LLC. Leveraging CAT, eOPF divides the LLC into enclave and non-enclave partitions. At enclave entries and resumes, eOPF switches the processor to the enclave partition, while the untrusted software (on other processors) use the non-enclave partition. On enclave exits, eOPF reverts the processor back to the non-enclave partition. While each partition can support unlimited enclaves, CAT can only support 15 distrusting partitions concurrently at this time. In particular, the latest CAT implementation has 16 domains [52] and 1 partition must remain reserved for untrusted software. In the future, if additional domains are implemented, eOPF can support additional distrusting partitions.

eOPF creates new LLC partitions using CAT-related MSRs, `IA32_L3_MASK_N`. A processor follows the partition specified in its register `IA32_PQR_ASSOC`. Whenever a partition is changed, all cache-lines must be invalidated to enforce the change. eOPF achieves this using `WBINVD`. Furthermore, eOPF prevents modifications to CAT MSRs during enclave execution to ensure full control over CAT.

5.2.2 Per-Core Resource Invalidation and Deactivation

Intra-core computational units (ICUs). Such units include Arithmetic Logic Units (ALUs) and Translation-Lookaside Buffer (TLBs). An attacker can abuse hyper-threading, a hardware feature that allows concurrent execution of two threads on the same processor core, to infer an enclave's access semantics onto ICUs [21, 44]. To defeat these attacks, eOPF ensures that hyper-threading is *deactivated* on the processor core that is running an enclave.

eOPF notifies the OS using its shared memory channel (§6) that a certain enclave should execute without hyper-threading. The OS can disable hyper-threading in software (e.g., OpenBSD does this by default [10]) by programming the x86 Local APIC [52]. In particular, once hyper-threading is disabled on a processor, it does not raise hardware interrupts. Hence, eOPF monitors each core for hardware interrupts and if it observes hardware interrupts on enclave-running (hyper-threaded) processor cores, it terminates the enclave. On each enclave exit, ICUs are automatically flushed by the SGX processor, leaving no observable intermediatte effect.

L1 and L2 cache. Enclave and untrusted programs that run sequentially or in parallel (using hyper-threading) on a processor core share cache lines across the L1 and L2 caches. An attacker can exploit this sharing to leak enclave contents through cache attacks [26,43]. eOPF *deactivates* hyper-threading (previous section) to prevent parallel attacks. To protect against sequential attacks, eOPF *invalidates* the L1/L2 cache (using `WBINVD`) at enclave exits. Hence, all enclave contents are flushed back to memory and the attacker observes an empty cache state on each attack.

Branch predictor units (BPUs). Branch predictor units like the branch target buffer (BTB) and pattern history table (PHT) predict the control-flow of a computation in an out-of-order CPU. Attackers can use them to infer an enclave's control-flow by observing whether a particular branch was taken or not [40,60]. Unfortunately, the SGX CPU does not provide native mechanisms to invalidate these units. Nevertheless, eOPF deactivates the components critical for their side-channel exploits and designs software invalidation.

Prior work has shown that reliable attacks on the BTB require specialized units such as the Last Branch Record (LBR) or Intel Processor Trace (PT) [60], particularly because of the BTB's small size in comparison to other predictors. The LBR and PT are performance tools that cannot be used by enclaves. Hence, eOPF deactivates the LBR and PT by setting MSRs, `IA32_DEBUGCTLA` and `IA32_RTIT_CTL`, respectively, and denying all modifications to these MSRs.

eOPF uses knowledge of the PHT's structure to implement a software invalidation technique, ensuring the attacker is unable to observe intermediate enclave artifacts on the PHT. In particular, the PHT contains 16,384 entries and is indexed by the lowest $log_2 N$ bits of a conditional branch instruction's address [40]. Each PHT entry is a 2-bit Finite State Machine with 4 states: (a) Strongly Not-Taken, (b) Weakly Not-Taken, (c) Weakly-Taken, and (d) Strongly-Taken. An entry is updated each time the processor takes (or does not take) a branch. Using this knowledge, eOPF generates conditional branches aligned to each PHT entry. For each branch, the code performs an always-true arithmetic comparision and takes the branch. Therefore, each PHT entry moves towards the Strongly-Taken state. eOPF runs the code thrice to ensure that the final state of each PHT entry is Strongly-Taken. Hence, the attacker always observes a uniform state of the PHT.

6 Implementation

We built a prototype of eOPF using the Bareflank extensible framework [3]. However, in practice, eOPF can be built using

any VMX framework or type-1 hypervisor (§2.1). By default, eOPF enables EPT protections, traps all enclave events and critical processor-related events (e.g., WRMSR), and enables the enclave orchestration service (refer to §4.1 and §5.1). Apart from bootstrapping and VMX-specific code, the base framework includes a SHA-256 hash generator [65] for co-attestation and kernel module for eOPF-OS communication. Furthermore, we implemented three eOPF modules: a *paging module* (PM) for page table protections, a *caching module* (CM) for L1/L2 and last-level cache protections, and a *branching module* (BM) for BPU protections.

Our prototype does not currently implement communication with the user. Such communication is a one-time cost at enclave creation; hence, it does not impact runtime results. Also, Bareflank does not currently support IOMMU to protect the framework against device-based attacks. Nevertheless, IOMMU should be enabled by default in cloud machines and it is not an additional slowdown factor incurred by eOPF.

7 Security Analysis

This section provides a security analysis of eOPF and protected services by discussing several attacks and implemented defenses (Fig. 4). It concludes with a brief TCB discussion.

7.1 Analyzing Framework Security

Preventing attacks against interposition. eOPF requires secure interposition of enclave and important system events. To prevent this interposition, the attacker can try to overwrite the VMCS, a data structure that contains the ENCLS-interception bitmap, the monitor trap flag (MTF), and is responsible for enabling other important interception functionality (e.g., WRMSR traps). Moreover, the attacker can try to modify the TRP and trick eOPF into thinking the enclave resumed. eOPF prevents all aforementioned attacks (§4.1).

The VMCS and its extended instruction bitmaps are located in protected eOPF memory and the attacker is unable to modify these structures to disable ENCLS or other system functionality interposition. The protected memory is created from virtualization extensions. In particular, eOPF uses EPT protections to prevent software access and IOMMU protections to prevent device access [89]. The critical data structures (or tables) of EPT and IOMMU are also stored within the protected memory; hence, the OS cannot access them. Finally, the TRP is located in a reserved region of the enclave process' address space, it cannot be overwritten due to memory protections, and its page tables are also write-protected.

Preventing attacks against co-attestation. The attacker can attempt to circumvent co-attestation by trying to leak secret eOPF keys provisioned in the machine or the enclave unique ID (*eid*), trying to guess *eid*, and replaying communication. eOPF prevents all such attacks (§4.2).

Potential attacks	**eOPF defense**
Against interposition (§4.1)	
Modify VMCS	Prevent software and device access using EPT and IOMMU, resp.
Disable mem. protections	Access-protect EPT/IOMMU structures
Modify TRP	Write-protect TRP and its PTs
Against co-attestation (§4.2)	
Leak eOPF secrets	Store in protected memory/storage and use SC-resistant crypto libraries
Steal *eid* using enclave	Only install to pre-measured enclave
Guess *eid*	Use very large number
Replay communication	Use Random nonces
Against enclave orchestration (§5.1)	
Modify LC MSRs	Trap writes to MSRs
Hide resource usage	Trap EADD; prevent TSC MSR changes
Hide malware in enclaves	Scan initial content and disable changes
Against side-channel defense (§5.2)	
Write to page tables	Write-protect using EPT
Disable/modify CAT	Trap writes to MSRs
Enable hyper-threading	Monitor interrupts on signalled cores
Enable LBR/PT	Trap corresponding MSRs

Figure 4: Table illustrates how eOPF defends against several attacks directed at its framework and protected services.

The platform is securely provisioned by the trusted cloud provider and all secret keys established during provisioning are securely maintained within the system's protected key management system (e.g., a persistent dedicated storage). Every subsequent cryptographic operation is also securely performed in a side-channel resistant manner ensuring the attacker cannot leak keys while they are being used. Moreover, during enclave installation, the *eid* is directly received by eOPF through a secure communication channel with a remote user and securely kept in protected memory. The possibility of the attacker being able to guess the correct *eid* is infinitesimally small (2^{-32768}). This is harder than guessing an RSA cryptographic key. Finally, even though the OS can record and replay network packets, all network communication is secured using random nonces to prevent replay attacks.

7.2 Analyzing Services Security

Preventing attacks against enclave orchestration. After compromising the OS, the attacker can attempt to run arbitrary enclaves by modifying launch control MSRs or hide their enclave resource usage from the cloud provider. Additionally, an attacker might try to hide malware inside enclaves. eOPF prevents all aforementioned attacks.

eOPF ensures that SGX flexible launch control features can only be configured by the cloud provider by intercepting all writes to the launch control MSRs. Hence, even a user that has compromised the OS cannot run enclaves on a machine without obtaining a launch token from the cloud provider. Since eOPF interposes all enclave supervisor and user interactions, it can trivially measure how the enclave is using resources. In particular, it uses RDTSCP to measure CPU

time on enclave entries and traps all instructions that insert or remove enclave pages. To prevent the attacker from keeping malware inside enclaves, eOPF scans enclave contents at load time and prevents subsequent code changes.

Preventing attacks against side-channel defense. The attacker can attempt to disable side-channel defenses by modifying entries inside enclave page tables (e.g., reset access bit), modify CAT configuration to disable last-level cache isolation, resume hyper-threading to leverage per-core side-channels, and re-enable LBR/PT to exploit BTB-related side-channels. eOPF protects against all aforementioned attacks (§5.2).

Access and dirty bits are set in enclave page table entries, and the tables are write-protected (using EPT) to prevent additional modifications. To modify CAT partitions, the OS would need to write to CAT-related MSRs (using `WRMSR`) and such a write is trapped by eOPF. If the attacker re-enables hyper-threading, they will be caught since the processor core will raise an interrupt that will be intercepted by eOPF. Finally, LBR and PT configurations can only be changed by writing to MSRs, and any such attempt is caught by eOPF.

7.3 Analyzing eOPF's TCB

This section analyzes eOPF's TCB followed by a brief discussion on the impact of attacks on eOPF to a user.

eOPF allows the OS to handle most functionality and only interposes on sensitive interactions (e.g., MSR writes). Hence, from a cloud machine's perspective, eOPF only marginally increases the TCB, which can be rigorously tested.

On a virtualized cloud machine, eOPF's complete TCB includes the hypervisor. We find this acceptable for several reasons. In particular, even though hypervisors can be large, the attacker-exploitable interface is typically significantly narrower than monolithic OSs, resulting in fewer discovered vulnerabilities in hypervisor codebases [28, 77]. The exploit of these vulnerabilities can be made significantly more challenging by using memory lockdown and compiler instrumentation to ensure hypervisor code integrity and control-flow integrity, respectively, with a small performance impact [86]. Moreover, eOPF's TCB can be reduced using hypervisor compartmentalization [77]. In such scenarios, eOPF can execute alongside a tiny security monitor and enforce security invariants while remaining isolated from the large cloud hypervisor.

Finally, since eOPF is external to the enclave and the microcode, it cannot access enclave contents and, during its operation, it is not exposed to enclave secrets. Hence, attacks against eOPF cannot harm the existing SGX guarantees.

8 Performance Evaluation

This section describes eOPF's performance through custom benchmarks and diverse real-world programs.

Setup. We evaluated eOPF using SGX desktop and server machines (Fig. 5). Although SGX is deprecated on desktops, we used the desktop because we observed a large number of enclave exits on it. In particular, the desktop has a smaller EPC which leads to frequent page faults (which cause exits) when running large enclaves [68]. Many of eOPF's side-channel defenses incur extra costs at exits; hence, the desktop machine allows us to better observe worst-case overheads.

	Desktop	Server
Hardware		
CPU model	i7-8700	Xeon Gold 6348
CPU sockets	1	2
Cores × threads	6 × 2	28 × 2
Clock speed	3.20GHz	2.60GHz
Cache (L1/L2/LLC)	64KB/256KB/12MB	64KB/1.2MB/42MB
LLC ways	16	12
RAM size	16GB	512GB
EPC size	128MB	128GB
Software		
Linux kernel	5.4	5.11
SGX SDK	2.3	2.15
SGX driver	Legacy 2.6	DCAP 1.41

Figure 5: Machine platforms used for evaluation.

We leveraged two software optimizations to reduce enclave exits, both of which are well-supported by modern systems. First, unless noted otherwise, we used the exitless (or *switchless*) system call setting for all experiments. This setting is now widely-supported (e.g., even the relatively basic SGX SDK supports it [17]) and is known to improve performance by avoiding expensive enclave exits using background request handling threads and system call batching. Other work also evaluates SGX enclaves using this option [22, 68]. Second, we configured both machine kernels as *tickless* [15] to reduce enclave exits due to frequent timer interrupts [66].

Terminology. eOPF refers to the base framework with all interposition and cloud orchestration features but no runtime side-channel defense. eOPF+PM refers to the system with the paging module enabled for paging side-channel defense. eOPF+CM refers to caching module enabled on a system to prevent cache attacks, while eOPF+BM refer to the system enabled with BPU defenses. Hyper-threading in enclave-running cores is disabled for both CM and BM. When all side-channel defenses are enabled, we refer to the system as eOPF+PM+CM+BM. Our baseline in all experiments was a non-virtualized system.

8.1 Microbenchmarks

This section describes our experiments to find the raw cost of eOPF's enclave interposition and side-channel protection at different events through two benchmarks.

Enclave event interposition benchmark. We created a test program that executes 100k barebones enclave entry and exit tests. In the entry test, the application enters the enclave while providing current (pre-entry) time as argument. The enclave measures the time it took to enter and returns to the appli-

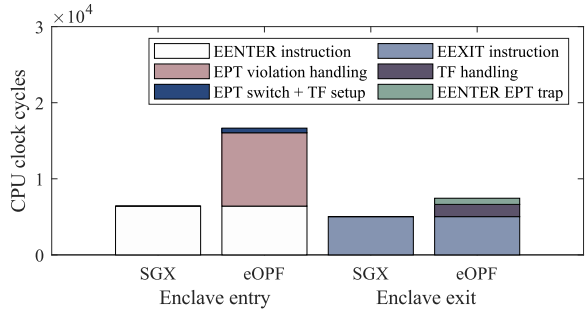


Figure 6: Overheads at enclave entry and exit for eOPF on our server machine. TF means *trap flag*. SGX's `EENTER` and `EEXIT` instruction cost was comparable to existing work [68].

Invalidation	Time (kcycles) Min	Time (kcycles) Max	Time (ms) Min	Time (ms) Max
Desktop				
CM (L1/L2 + LLC)	247	10560	0.08	3.35
BM (PHT)	120	844	0.04	0.30
Total	367	11404	0.12	3.65
Server				
CM (L1/L2 + LLC)	3240	19454	1.25	7.48
BM (PHT)	373	494	0.14	0.19
Total	3613	19947	1.39	7.67

Figure 7: Overheads due to resource invalidation at enclave exits for CM and BM.

cation. In the exit test, the same set of operations occur but from enclave to the application. To get a detailed picture, we also measured the time to perform eOPF-specific tasks (e.g., switch EPT during entry) from inside the framework. The time was measured everywhere using `RDTSCP`. Finally, we disabled *exitless* mode for this experiment to get the full cost of exits. Fig. 6 shows the performance overhead incurred in our experiment with native SGX and eOPF.

*e*OPF *overhead.* eOPF adds 59% overhead to enclave entries, while it adds 71% overhead to enclave exits.

On each entry, eOPF incurs a virtual machine exit to handle the EPT violation, which takes 9639 cycles on our machine. We believe some of this cost is because eOPF is implemented on Bareflank, which is designed for modularity instead of performance; hence, it can be further optimized. Once the violation is handled, eOPF sets the trap flag to trap subsequent exits and switches the EPT to allow enclave execution. These tasks take 1067 cycles.

On each enclave exit, eOPF incurs a virtual machine exit for the trap flag, which only takes 1608 cycles on our machine. Afterwards, eOPF switches the EPT to trap subsequent enclave entries which takes 812 cycles on our machine.

Side-channel protection benchmark. We ran a benchmark enclave program that continuously writes to a large 256 MB buffer on both machines. We ran the enclave continuously for 60 seconds and measured incurred performance overheads (using `RDTSCP`) while enabling different side-channel protection modules. Since the resources affected by the caching and branching modules (CM and BM) have different sizes on each of our test machines, we ran their experiments on each machine. Fig. 7 shows the runtime performance overhead.

*e*OPF+PM *overhead.* Paging defenses introduce a one-time cost, during the enclave's lifetime, at enclave creation. The paging module performs the following steps: (a) maps guest page tables to eOPF's address space, (b) scans entire page tables (including non-enclave entries) to find the enclave regions and sets access/dirty bits, and (c) write-protects enclave page table entries. On the server machine, these steps adds an additional ∼1.9 seconds to our enclave's creation. We expect this cost is negligible for longer-running enclaves.

*e*OPF+CM *overhead.* At enclave exits, eOPF's cache defenses (§5.2.1 and §5.2.2) require (a) partitioning the last-level cache (LLC) and switching partitions during enclave execution and (b) writing back and invalidating the caches at enclave exits. Please refer to §8.2 for the runtime overhead.

Partitioning the LLC and switching partitions is fast: it takes ∼200 cycles to update a model-specific register (using `WRMSR`). Cache write-back and invalidation time depends on the state and size of the cache. On the desktop machine, we noticed that it took up to 3.35 ms, whereas its lower bound (through consecutive invalidations) was 0.08 ms. Cache invalidation took from 1.25 ms to 7.48 ms on the server.

Despite a smaller cache, invalidation on the desktop is not that much faster than the server. The reason is that, unlike server machines where SGX does not implement hardware memory integrity [41], the desktop enforces integrity using a Merkle tree. This tree is updated on each cache-line that is flushed to DRAM [46], incurring 6 additional memory accesses per-cache-line. Notably, invalidating non-enclave memory on the desktop machine took only up to 0.81 ms.

*e*OPF+BM *overhead.* We executed our custom branch predictor flush to invalidate the PHT (§5.2.2). The lower bound for invalidation was 0.04 and we saw an upper bound of 0.30 milliseconds. Since typically branch misprediction adds a 5 nanoseconds latency [36], our evaluation results indicate that all branches were being mispredicted.

Benchmark result summary. eOPF interposition and side-channel protection cost is only incurred at enclave entry or exits. Although the cost can be high, these events are only a small fraction of the program's execution and can be significantly reduced with widely-available optimizations like switchless enclaves and tickless kernels (§8). For instance, in our experiments with SGX SDK's switchless benchmark [14], we noticed only 3k exits for 2 million enclave calls. Hence, as the next section will demonstrate, eOPF's overhead on real-world programs using software optimizations is typically modest, even with all side-channel protections enabled.

8.2 Real-world Enclave Programs

Common settings and results. While partitioning sensitive functionality of a program to run inside an enclave was the initial SGX intent, it has evolved over the years to run entire programs inside enclaves using Library OSs [25, 76, 80, 83], particularly for convenience reasons. In fact, even Intel has officially adopted (and continuously supports) the Gramine Library OS (formerly Graphene [83]) as an SDK for running Linux programs inside SGX [5]. Hence, we also used the Occlum and Gramine Library OSs [76, 83] to run Linux programs inside enclaves for evaluation.

Unless noted otherwise, we ran programs on the server machine using an enclave partition size of 1/12*LLC, the smallest allowed CAT-based partition on the server machine. This setting allows the machine to be shared amongst the most number of users, highly desirable in cloud machines. We ran each program 10 times and report the average.

Since eOPF+PM only incurs a one-time performance overhead during enclave creation, in our long-running programs, its performance impact was negligible. Hence, we do not illustrate its overhead in Fig. 8 and Fig. 9

We also evaluated the overhead incurred by the base framework alone (i.e., no side-channel modules were enabled) during each real-world program's execution inside enclaves. The base framework must interpose all enclave events, which increases the runtime cost (§8.1). However, our evaluation shows that this cost is very small during execution. On the server machine, the framework's cost is less than 2% on average during execution of each real-world program described in this section, primarily because exits are low and CPU virtualization (required by the framework) is lightweight.

Assorted (SPEC). SPEC is a collection of well-known CPU and memory-intensive programs that are useful to assess real-world system performance. It has been used for evaluation by the Occlum LibOS (which we used for this experiment) and includes real-world programs (e.g., compiler toolchains and compression libraries) that are evaluation targets for other SGX systems [75]. The Occlum LibOS is designed to support SPEC 2006 integer benchmarks out-of-the-box, unlike the latest SPEC 2017. Hence, we decided to evaluate our system with SPEC 2006 integer benchmarks.

Fig. 8 illustrates eOPF's performance across SPEC using reference datasets on the server machine. Encouragingly, even with all protections enabled, most programs incurred a modest performance overhead—7 out of 11 incurred less than 20% slowdown, and the geometric mean slowdown was 17%. Across all programs, the biggest slowdown factor was cache protections. Since our experiments used switchless system calls and tickless kernels (§8), most programs incurred very few enclave exits and showed modest performance overhead. Nevertheless, the smaller enclave LLC partition had a considerable effect (e.g., 311%) on the performance of highly memory-intensive programs like gcc and omnetpp.

We also ran SPEC programs on the desktop machine using test datasets to estimate performance in worst-case scenarios with many enclave exits. Since reference workloads require significant memory, it is infeasible to run them on the desktop machine. Fig. 9 illustrates eOPF's performance on the desktop using 1/8*LLC, the closest to fair sharing for each core. With both CM and BM enabled, the geometric mean overhead was 34% on the desktop machine. Since the desktop machine only has a 128 MB EPC, demand paging was inevitable. Thus, the programs incurred many more enclave exits because of page faults and performance was (expectedly) lower than the server machine. We noticed two programs, mcf and sjeng, incurred a very high overhead. We found that their test datasets required up to 1 GB of memory, hence their enclaves incurred the most exits (due to page faults) per-second.

Key-value store (Redis). Key-value stores like Redis [12] are widely used in cloud environments. We evaluated Redis using default settings and its official redis-benchmark, which tests 20 different key-value store operations including GET, SET, MSET, and POP. We ran each operation for 100,000 iterations using the default settings of 50 parallel clients. In its default state, Redis only keeps the key-value store *in-memory* for performance and does not write to disk. Additionally, note that while Redis ran inside an enclave, client network socket connections were received by user-space code outside the enclave, since enclaves cannot receive network packets directly. This code then sent the packets to the Redis enclave. In typical scenarios, client request in the packets would be protected using TLS that terminates inside the enclave, but redis-benchmark does not support TLS. Hence, requests were unencrypted and we used this for benchmarking purposes.

In our experiments, eOPF+CM+BM reduced throughput by 4–21% (geometric mean was 11%) across these operations. We only observed 27 enclave exits per-second during the benchmark's 119s execution. These exits were few due to switchless optimizations (§8). Given a low number of enclave exits and the fact that Redis is highly memory-intensive, the major factor behind its throughput reduction was the program executing on a restricted LLC partition.

Web server (Lighttpd). Webservers like Lighttpd [8], handle sensitive queries to fetch webpages, and hence are a good fit for SGX. We ran Lighttpd with 8 worker threads because this setting maximized throughput. From a separate server machine (average latency between machines was 0.09 ms), we used ApacheBench to send 10,000 HTTP requests for a 10 KB file from up to 256 concurrent clients. We sent HTTP requests like prior research [76] for stress benchmarking. In real-world cases, HTTPS ensures a request is end-to-end protected with TLS connections terminating inside the enclave.

In our experiments, eOPF+CM+BM's geometric mean throughput reduction across the test was only 5%. Interestingly, requests from a single client incurred a 65% throughput reduction, while requests from 256 concurrent clients incurred only 1% reduction. The reason is that the worker threads go

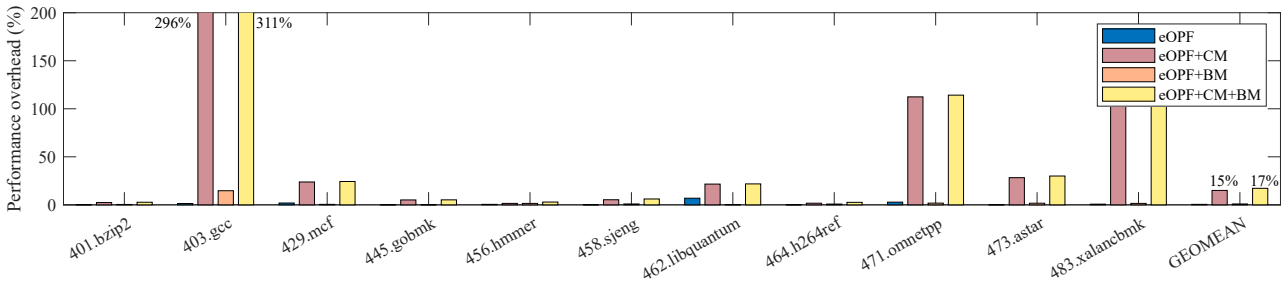


Figure 8: SPEC CPU 2006 performance with eOPF using the reference dataset on the server machine. The enclave partition was 1/12*LLC. For this test, the enclave exits per-second were: 3, 105, 4, 3, 3, 2, 2, 1, 11, 1, 8, from left to right.

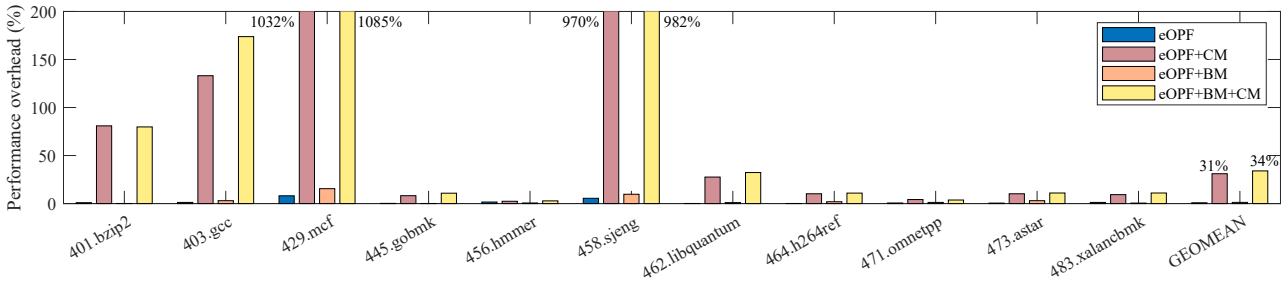


Figure 9: SPEC CPU 2006 performance with eOPF using the test dataset on the desktop machine. The enclave partition was 1/8*LLC. For this test, the enclave exits per-second were 29, 2184, 3071, 965, 25, 3444, 417, 298, 60, 22, 91, from left to right.

to sleep when there are no requests and they are awakened through inter-processor-interrupts, hence they incur additional enclave exits. With greater concurrency, the workers are always busy handling requests, thus they do not go to sleep.

8.3 Key takeaways

T1. The base eOPF (which enables secure enclave orchestration) incurs a low performance overhead (<2%) on real-world programs because (a) it leverages lightweight techniques and (b) its overhead is incurred at infrequent enclave exits.

T2. While eOPF's overhead expectedly increases with principled side-channel defenses, especially for highly memory-intensive programs (e.g., 311% for gcc on the server machine), it remains modest for the vast majority of programs (e.g., 17% geomean for SPEC programs on the server machine).

T3. eOPF's side-channel defense overhead is comparable to defenses that detect attacks using heuristics (e.g., Varys [66] incurs 15% overhead). However, through invalidation and isolation, eOPF provides strong protection akin to cryptographic techniques that obfuscate *all* side-channel leakage with high costs (e.g., Raccoon [70] incurs 21.8× overhead).

T4. Given the modest defense cost and the fact that eOPF allows users to flexibly decide when defenses are applied, eOPF can be practically adopted in today's cloud machines.

9 Discussion

Virtual machine support. In addition to containers, eOPF can orchestrate and protect enclaves running in different virtual machines (VMs) without a design change. This is because eOPF executes at the hypervisor layer, where it has the ability to distinguish between enclaves in different VMs [50, 52] during enclave lifecycle interposition (§4.1). In particular, when the hypervisor starts or resumes a VM, eOPF tracks this using the virtual machine control structure (VMCS). Subsequently, at any exit to the VMM during enclave creation or asynchronous enclave exits, eOPF determines which VM encountered this event by checking the VMCS again. Finally, eOPF interposes on (a) enclave entries using the per-VM EPT and (b) synchronous enclave exits using the single-step interception bit, which is also set in the per-VM VMCS.

Co-attestation without premeasurement. While provisioning an enclave with a secret identifier (*eid*) during co-attestation, the requirement is that *eid* is not disclosed to an attacker-controlled enclave (§4.2). In principle, this can be achieved without premeasurement if eOPF (a) installs the *eid* in any recently-created enclave and (b) restricts *eid* enclave page permissions using EPT to prevent the enclave from accessing it, unless SGX measurement is called at which time the user will verify. We leave the study of alternate co-attestation primitives for eOPF to future work.

Supported enclave count. By default, eOPF is a thin orchestration layer that collects statistics and enforces properties for cloud providers; hence, it supports as many enclaves as the platform originally can. If *all* side-channel protections are enforced (§5.2), eOPF can support (a) as many enclaves as can be kept resident in the EPC (i.e., within 512 GB in modern systems [41]) and (b) as many *distrusting* containers (each with unlimited enclaves) as CAT partitions allow (currently

15 partitions). The first limit can be removed using oblivious page swapping mechanisms (discussed in the *future eOPF extensions* paragraph in this section). The second limit can be addressed if future hardware iterations increase the number of isolated cache partitions (e.g., using recent proposals [37]). eOPF can be easily extended to leverage new functionality when it is made available by developers or hardware vendors.

Future eOPF extensions. eOPF is designed to be an extensible framework that flexibly provides several guarantees to cloud providers and enclave users. One possible extension would be to support *oblivious swapping* of enclave pages at page faults [67]. In particular, when a page fault happens during enclave execution, eOPF can clear the `CR2` register to shield the faulting page from the OS. Instead, eOPF can provide a list of candidate pages for the OS to bring into the EPC. To create a secure candidate list, eOPF can rely on a cryptographically-secure algorithm like the Oblivious RAM (ORAM) [82]. eOPF can also enable the use of efficient per-thread hardware memory protection (using MPK) for enclaves and enable memory protection use-cases [49]. In particular, currently enclaves cannot securely use MPK since it requires setting protection keys in page tables [20], which are controlled by the OS. Instead of relying on the OS, the enclave can rely on eOPF to set correct protection keys.

10 Related Work

Privileged software monitors for TEEs. A lot of research has been done to design software security monitors that are more privileged than the OS and leverage them to create protected process contexts with strong isolation guarantees. Many systems [31, 47, 48, 62] rely on hardware memory protection capabilities (e.g., EPT) of virtualization layers (e.g., VMX) for such security monitors. Other systems [35, 38] rely on compiler instrumentation (e.g., software fault isolation) to deprivilege the OS and execute a security monitor at ring-0. On non-x86 systems, several designs [34, 41, 42, 59] leverage architectural privileged layers like ARM TrustZone or RISC-V machine mode and their protection features (e.g., physical memory protection). While our design of eOPF takes inspiration from all these systems, eOPF remains unique for several reasons. First, eOPF only offers complementary protection to enclaves, ensuring that even if its monitor is compromised, user computations retain SGX protections. Second, by leveraging SGX and its extensive industry support, eOPF can be readily-adopted by cloud providers without hardware changes or designing extensive software development kits.

SGX digital side-channel defenses. Researchers have proposed both software and hardware solutions to address digital side-channels in enclaves. Software solutions implemented inside enclaves cannot prevent memory access patterns from being disclosed since that is a hardware limitation. Therefore, many software protection schemes rely on cryptographic protocols like ORAM [18, 70, 71] to obfuscate all memory access patterns (i.e., make all access patterns indistinguishable). However, since ORAM is expensive, these defenses incur significant slowdown (e.g., 21.8× [70]). Other software solutions [45, 66, 78, 79] leverage heuristics to detect certain attack vectors. On the hardware front, Autarky [67] implements strong and efficient protection against page table attacks. One of Autarky's ideas is to set all access and dirty bits for enclave page table entries, which is also adopted by eOPF's page table defense. In contrast to these defenses, eOPF offers a more comprehensive protection against several side-channels with low performance impact and minimal user effort.

Running programs inside enclaves. Haven [25] runs Microsoft Windows programs in enclaves with minimal changes. Graphene [83] and Panoply [80] implement library OSs to run Linux applications inside enclaves, while VC3 [73] allows developers to protect data analytics. Ryoan [49] provides trusted client-server application processing in SGX and Scone [22] enables SGX-protected containers. Eleos [68] designs user-level paging to reduce enclave exits and improve performance. eOPF is orthogonal to this line of research and can improve the security guarantees provided by these systems.

Enclave and platform attestation. Windows 11 machines leverage the TPM [23] for secure boot and platform attestation. SGX's remote enclave attestation is also inspired by the TPM. Recently, MAGE [30] demonstrated how to extend SGX enclave to attest mutually-trusted enclaves together by leveraging a *premeasurement* of enclave memory regions. eOPF's use of premeasurement (pM_e) is inspired by MAGE, but eOPF uses it to enable platform-enclave co-attestation.

11 Conclusion

eOPF provides a trusted privileged environment for cloud providers to enable protected services on their SGX-capable confidential computing platform. In this paper, we overcome several challenges to design eOPF, implement secure cloud orchestration and complementary side-channel defense as services enabled by eOPF, and provide a detailed security and performance analysis of the framework. Our results indicate that eOPF provides strong protection with very low performance impact on average (<2% for the framework alone) and it can be readily-adopted in today's clouds.

12 Acknowledgment

We would like to thank the anonymous reviewers and our shepherd, Bryan Parno, for their insightful reviews which significantly improved the paper's evaluation and presentation. This work was partly supported by the National Science Foundation (NSF) under grants CNS-2145888 and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2023-00209093).

References

[1] AWS Lambda. https://aws.amazon.com/lambda/.

[2] AWS Nitro Enclaves. https://aws.amazon.com/ec2/nitro/nitro-enclaves/.

[3] Bareflank/hypervisor. https://github.com/Bareflank/hypervisor.

[4] Intel 3rd Gen Xeon Scalable Processors (Ice Lake). https://www.storagereview.com/news/intel-3rd-gen-xeon-scalable-processors-ice-lake.

[5] Intel(r) Software Guard Extensions. https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html.

[6] Key concepts and Definitions for Burstable Performance Instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-credits-baseline-concepts.html.

[7] L1 Terminal Fault / CVE-2018-3615 , CVE-2018-3620,CVE-2018-3646 / INTEL-SA-00161. https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault.

[8] Lighttpd - Fly Light. https://www.lighttpd.net/.

[9] Nginx. https://www.nginx.com/.

[10] OpenBSD: HyperThreading Disabled by Default on Install. https://marc.info/?l=openbsd-cvs&m=152943660103446.

[11] Qubole Announces Apache Spark on AWS Lambda. https://www.qubole.com/blog/spark-on-aws-lambda/.

[12] Redis. https://redis.io/.

[13] Standard Performance Evaluation Corporation. https://www.spec.org/cpu2006/.

[14] Switchless Enclave Example. https://github.com/intel/linux-sgx/tree/master/SampleCode/Switchless.

[15] Tickless Kernel. https://en.wikipedia.org/wiki/Tickless_kernel.

[16] 23andme: DNA Genetic Testing and Analysis, 2017.

[17] 01org. Intel(r) software guard extensions for linux* os (source code). https://github.com/01org/linux-sgx, 2016.

[18] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. Obfuscuro: A Commodity Obfuscation Engine for Intel SGX. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2019.

[19] Adil Ahmad, Juhee Kim, Jaebaek Seo, Insik Shin, Pedro Fonseca, and Byoungyoung Lee. Chancel: Efficient Multi-client Isolation Under Adversarial Programs. In *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS)*, 2021.

[20] Adil Ahmad, Sangho Lee, Pedro Fonseca, and Byoungyoung Lee. 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, USA, April 2021.

[21] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida García, and N. Tuveri. Port Contention for Fun and Profit. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[22] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, 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.

[23] Will Arthur and David Challener. *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*. Apress, 2015.

[24] Microsoft Azure. Azure confidential computing. https://azure.microsoft.com/en-us/blog/azure-confidential-computing/, 2018.

[25] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, October 2014.

[26] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, 2017.

[27] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD.

[28] Ramaswamy Chandramouli, Ramaswamy Chandramouli, Anoop Singhal, Duminda Wijesekera, and Changwei Liu. *Methodology for Enabling Forensic Analysis Using Hypervisor Vulnerabilities Data*. US Department of Commerce, National Institute of Standards and Technology, 2019.

[29] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *Proceedings of IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[30] Guoxing Chen and Yinqian Zhang. MAGE: Mutual Attestation for a Group of Enclaves without Trusted Third Parties. In *Proceedings of the 31st USENIX Security Symposium (Security)*, August 2022.

[31] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*.

[32] Yuan Chen, Jiaqi Li, Guorui Xu, Yajin Zhou, Zhi Wang, Cong Wang, and Kui Ren. SGXLock: Towards Efficiently Establishing Mutual Distrust Between Host Application and Enclave for SGX. In *Proceedings of the 31st USENIX Security Symposium (Security)*, August 2022.

[33] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[34] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proceedings of the 25th USENIX Security Symposium (Security)*, 2016.

[35] John Criswell, Nathan Dautenhahn, and Vikram Adve. Virtual ghost: Protecting applications from Hostile Operating Systems. *ACM SIGARCH Computer Architecture News*, 2014.

[36] Jeff Dean. Latency numbers every programmer should know. https://gist.github.com/jboner/2841832.

[37] Ghada Dessouky, Alexander Gruler, Pouya Mahmoody, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Chunkedcache: On-demand and Scalable Cache Isolation for Security Architectures. In *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS)*, Virtual Event, USA, February 2021.

[38] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L Cox, and Sandhya Dwarkadas. Shielding Software from Privileged Side-Channel Attacks. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug 2018.

[39] Alan M Dunn, Owen S Hofmann, Brent Waters, and Emmett Witchel. Cloaking Malware with the Trusted Platform Module. In *Proceedings of the 20th USENIX Security Symposium (Security)*, San Francisco, CA, August 2011.

[40] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[41] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Scalable Memory Protection in the PENGLAI Enclave. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual Event, USA, July 2021.

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

[43] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In *EUROSEC*, pages 2–1, 2017.

[44] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug 2018.

[45] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *Proceedings of the 26th USENIX Security Symposium (Security)*, 2017.

[46] Shay Gueron. A Memory Encryption Engine Suitable for General Purpose Processors. Cryptology ePrint Archive, Report 2016/204, 2016. https://eprint.iacr.org/2016/204.

[47] Alexander Van't Hof and Jason Nieh. 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.

[48] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. InkTag: Secure Applications on an Untrusted Operating System. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*.

[49] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, November 2016.

[50] Intel. Intel Trusted eXecution Technology–Software Development Guide. *Document number 315168-005*.

[51] Intel. Intel® processors voltage settings modification advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00289.html.

[52] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual. *Volume 3A: System Programming Guide*, 2016.

[53] Intel. Intel 64 and ia-32 architectures software developer's manual. *Volume 3D: System Programming Guide*, 2022.

[54] Intel Jason Chen. Supporting TEE on x86 Client Platforms with pKVM. https://www.youtube.com/watch?v=EP9ps_h-WeI.

[55] Pratheek Karnati. Data-in-use Protection on IBM Cloud using Intel SGX. https://www.ibm.com/cloud/blog/data-use-protection-ibm-cloud-using-intel-sgx.

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

[57] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Sgxbounds: Memory safety for shielded execution. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*.

[58] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-Che Tsai, and Raluca Ada Popa. An off-chip attack on hardware enclaves via the memory bus. In *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, Aug 2020.

[59] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*, 2020.

[60] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Aug 2017.

[61] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing. In *IEEE international symposium on high performance computer architecture (HPCA)*, 2016.

[62] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of the 31th IEEE Symposium on Security and Privacy (Oakland)*, May 2010.

[63] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proceedings of the 3rd European Conference on Computer Systems (EuroSys)*, Glasgow, Scotland, March 2008.

[64] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on HASP*, 2013.

[65] Shintarou Okada. a header-file-only, sha256 hash generator in c++. https://github.com/okdshin/PicoSHA2.

[66] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, June 2018.

[67] Meni Orenbach, Andrew Baumann, and Mark Silberstein. Autarky: Closing Controlled Channels with Self-Paging Enclaves. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*, 2020.

[68] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, Belgrade, Serbia, April 2016.

[69] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella-Béguelin. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2020.

[70] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, August 2015.

[71] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. ZeroTrace: Oblivious Memory Primitives from Intel SGX. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2018.

[72] Sarah Schlothauer. Serverless platform Apache OpenWhisk graduates to Top Level Project. https://jaxenter.com/serverless-openwhisk-top-level-160417.html, 2019.

[73] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy Data Analytics in the Cloud using SGX. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2015.

[74] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard eXtension: Using SGX to Conceal Cache Attacks. In *Proceedings of the 14th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, July 2017.

[75] Jaebaek Seo, Byounyoung Lee, Seongmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2017.

[76] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[77] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. Deconstructing Xen. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2017.

[78] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 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, February 2017.

[79] S Shinde, ZL Chua, V Narayanan, and P Saxena. Preventing your Faults from Telling your Secrets. In *Proceedings of the 11th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Xi'an, China, May–June 2016.

[80] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. PANOPLY: Low-TCB Linux Applications With SGX Enclaves. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2017.

[81] Splunk. How Good is ClamAV at Detecting Commodity Malware? https://www.splunk.com/en_us/blog/security/how-good-is-clamav-at-detecting-commodity-malware.html.

[82] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, October 2013.

[83] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, June 2017.

[84] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets Without Page Faults: Stealthy Page Table-based Attacks on Enclaved Execution. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Aug 2017.

[85] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland), 2019*.

[86] Zhi Wang and Xuxian Jiang. Hypersafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Proceedings of the 31th IEEE Symposium on Security and Privacy (Oakland)*, May 2010.

[87] Richard Wilkins and Brian Richardson. UEFI Secure Boot in Modern Computer Security Solutions, 2013.

[88] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2015.

[89] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building Verifiable Trusted Path on Commodity x86 Computers. In *2012 IEEE Symposium on Security and Privacy (S&P))*.