HyperEnclave: An Open and Cross-platform Trusted Execution Environment

Yuekai Jia[1], Shuang Liu[2], Wenhao Wang[3,4(✉)], Yu Chen[1], Zhengde Zhai[2], Shoumeng Yan[2], and Zhengyu He[2]

[1]*Tsinghua University*
[2]*Ant Group*
[3]*SKLOIS, Institute of Information Engineering, CAS*
[4]*School of Cyber Security, University of Chinese Academy of Sciences*

Abstract

A number of trusted execution environments (TEEs) have been proposed by both academia and industry. However, most of them require specific hardware or firmware changes and are bound to specific hardware vendors (such as Intel, AMD, ARM, and IBM). In this paper, we propose HyperEnclave, an open and cross-platform process-based TEE that relies on the widely-available virtualization extension to create the isolated execution environment. In particular, HyperEnclave is designed to support the *flexible enclave operation modes* to fulfill the security and performance demands under various enclave workloads. We provide the enclave SDK to run existing SGX programs on HyperEnclave with little or no source code changes. We have implemented HyperEnclave on commodity AMD servers and deployed the system in a world-leading FinTech company to support real-world privacy-preserving computations. The evaluation on both micro-benchmarks and application benchmarks shows the design of HyperEnclave introduces only a small overhead.

1 Introduction

In recent years, trusted execution environments (TEEs) are emerging as a new form of computing paradigm, known as confidential computing, due to the high demand for privacy-preserving data processing technologies that can handle massive data samples. TEEs provide hardware-enforced memory partitions where sensitive data can be securely processed. Existing TEE designs support different levels of TEE abstractions, such as process-based (Intel's Software Guard eXtensions (SGX) [55]), VM-based (AMD SEV [45]), separate worlds (ARM TrustZone [16]), and hybrid (Keystone [49]). Currently, the most prominent example of TEEs is Intel SGX, which is widely available in commercial off-the-shelf (COTS) desktop and server processors.

Motivations. Most of today's TEE technologies are close-sourced and require specific hardware or firmware changes that are difficult to audit, slow to evolve, and thus are inferior to cryptographic alternatives (such as homomorphic encryption), which are based upon public algorithms and widely available hardware. Moreover, most existing TEE designs restrict the enclaves (i.e., the protected TEE regions) to run only in fixed mode.[1] It is difficult to support the performance and security requirements of various types of applications that need to be protected by TEEs. For example, Intel SGX enclaves run in the user mode and cannot access privileged resources (such as the file system, the IDT, and page tables) and process privileged events (interrupt and exceptions). As a result, running I/O-intensive and memory-demanding tasks leads to significant performance degradation.

To fill the gap, in this paper we propose the design of HyperEnclave to support *confidential cloud computing* that can run securely on both legacy servers readily available in the cloud, and on the rising ARM (or RISC-V in the future) servers, without requiring specific hardware features. For this purpose, our design provides a process-based TEE abstraction using the widely available virtualization extension (for isolation) and TPM (for root of trust and randomness etc.). To better fulfill the needs for specific enclave workloads, HyperEnclave supports the *flexible enclave operation modes*, i.e., the enclaves can run at different privilege levels and can have access to certain privileged resources (see Sec. 4 for more details).

Design details. In our design, the system runs in three modes. A trusted software layer, called RustMonitor (security monitor written in Rust), runs in the *monitor mode*, which is mapped to the VMX root mode. RustMonitor is responsible for enforcing the isolation and is part of the trusted computing base (TCB). The untrusted OS (referred to as the *primary OS*) provides an execution environment for the untrusted part of applications; the untrusted OS and application parts run in the *normal mode*, which is mapped to the VMX non-root mode. The trusted part of application (i.e., *enclave*) runs in the *secure mode*, which can be mapped *flexibly* to ring-3 or ring-0 of the VMX non-root mode, or ring-3 of the VMX root mode.

*Corresponding author: Wenhao Wang (wangwenhao@iie.ac.cn).

[1]An exception is CURE [20], which however requires hardware changes to the CPU core and the system bus to support the flexible enclaves (Sec. 9).

Memory isolation is enforced with hardware-based memory protection of the memory-management unit (MMU). As we observe that existing process-based TEEs (e.g., Inktag [38] and Intel SGX [55]) are vulnerable to page-table-based attacks [74], our memory isolation scheme chooses to manage the enclave's page table and page fault events entirely by the trusted code, removing the involvement of the primary OS. The design also prevents certain types of enclave malware attacks (Sec. 3.2).

To minimize the attack surface, we adopt an approach called *measured late launch*: the primary OS kernel is first booted; then a chunk of special kernel code, implemented as a kernel module in the primary OS, runs to initiate RustMonitor in the most privileged level (i.e., the monitor mode) and demotes the primary OS to the normal mode. All booted components during the booting process are measured and extended to the TPM Platform Configuration Registers (PCRs). Since the TPM attestation guarantees that PCRs cannot be rolled back, the design ensures that RustMonitor is securely launched; otherwise, a violation of the TPM *quote* would be detected during remote attestation.

We have implemented HyperEnclave on commodity AMD servers. In total RustMonitor consists of about 7,500 lines of Rust code. The APIs of our enclave SDK are compatible with the official SGX SDK. As a result, code written for SGX could be easily ported to run on HyperEnclave by recompiling the code with little (or no) source code changes. We have ported a number of SGX applications, as well as the Rust SGX SDK [71] and the Occlum library OS [64] to HyperEnclave. The micro-benchmarks show that the overheads for ECALLs and OCALLs are < 9,700 and < 5,260 cycles respectively (14,432 and 12,432 cycles respectively on Intel SGX). The evaluation on a suite of real-world applications shows that the overhead is small (e.g., the overhead on SQLite is only 5%).

Contributions. In summary, the paper proposes the design of HyperEnclave, with the following contributions:

- An open[2] and cross-platform processed-based TEE with minimum hardware requirements (virtualization extensions and TPM) that can run existing SGX programs with little or no source code changes, which enables the reuse of the rich toolchains and ecosystem for Intel SGX.
- Supporting the flexible enclave operation modes to fulfill the diverse security and performance requirements of enclave applications without hardware or firmware changes.
- A memory isolation scheme that the enclave's page table and page fault are managed entirely by the trust code, which mitigates the page-table-based attacks and the enclave malware attacks.
- A measured late launch approach, combined with the TPM-based attestation to reduce the attack surface.
- An implementation on commodity servers (mostly) using the memory safe language Rust, and an evaluation on real hardware and applications, demonstrating that the proposed design is practical and only has a small overhead.

[2]The code will be available at https://github.com/HyperEnclave.

2 Background

2.1 Trusted Execution Environment

A Trusted Execution Environment (TEE) is designed to ensure that sensitive data is stored, processed, and protected in an isolated and trusted environment. The isolated area could be a separate system apart from the normal operating system (such as the TrustZone [16] secure world), a part of a process address space (such as an Intel SGX [55] enclave), or a stand-alone VM (such as a virtual machine protected by AMD SEV [45] or Intel TDX [41]). To resist the privileged attacker, TEE needs to thwart not only the OS-level adversary but also the malicious party who has physical access to the platform. To this end, it offers hardware-enforced security features including isolated execution, integrity, and confidentiality protection of the enclave, along with the ability to authenticate the code running inside a trusted platform through remote attestation.

Isolation. At the core of a TEE is the memory isolation scheme, which guarantees that code, data, and the runtime state of the enclave cannot be accessed or tampered with by untrusted parties. For Intel SGX, the protected memory (i.e., the *enclave*) is mapped to a special physical memory area called Enclave Page Cache (EPC), which is encrypted and cannot be directly accessed by other software, firmware, BIOS, and direct memory access (DMA).

Attestation. The goal of remote attestation is to generate an attestation *quote*, which includes the measurement of the software state, signed with the attestation key embedded in the hardware. The remote user verifies the validity of the quote by checking the signature (which reflects the hardware identity) and the measurement (which proves the software state).

2.2 Trusted Platform Module

Trusted Platform Module (TPM) is both an industry-standard [36] and an ISO/IEC standard [4] for a secure cryptoprocessor. It is used by nearly all PC and server manufacturers. Firmware TPMs (fTPMs) are firmware-based (e.g. UEFI) TPM implementations. At the time of this writing, Intel, AMD, and Qualcomm all have implemented fTPMs.

TPM has a set of Platform Configuration Registers (PCRs), which can be used for the measurement of the booted code during the boot process. PCRs are reset to zero on system reboot or power on-off. During every boot process, the PCRs can only be extended with the new measurement (called PCR extend), and thus cannot be set to arbitrary values.

Every TPM ships with a unique asymmetric key, called the Endorsement Key (EK), embedded by the manufacturer as the root of trust. The TPM can generate a quote of the PCR values, signed using the TPM Attestation Identity Keys (AIK), while the AIK is generated inside TPM and certified using EK. Any modifications of the booted code would be reflected in the quote. Upon receiving the quote, the remote party can validate the signing key comes from an authentic TPM and can be assured that the PCR digest report has not been altered.

2.3 Threat Model

Like the other TEE proposals [23, 49], we trust the underlying hardware, including the processor establishing the virtualization-based isolation, the System Management Mode (SMM) code, as well as the TPM. We assume that the Core Root of Trust for Measurement (CRTM) is trusted and immutable. HyperEnclave mitigates certain physical memory attacks, such as cold boot attacks and bus snooping attacks with the hardware support for memory encryption. We don't fully trust the operator and assume the attacker cannot mount physical attacks during the boot process, i.e., we assume that the system is initially benign (during system boot), and the early OS during the boot stage is part of the TCB. This can be achieved in two ways.

- *Firstly*, the power-on event can be secured with a hardware device, such as an HSM (i.e., hardware security module). The platform enters the boot process only with the engagement and supervision of a trusted party, who owns the HSM. After that, the operators for maintenance are not trusted.
- *Secondly*, the boot process can be enhanced to defend against adversaries with physical accesses. To prevent I/O attacks, we can harden the OS to remove unnecessary devices and disable the DMA capability of peripherals before IOMMU is enabled. We can enable memory encryption at an early stage (e.g., in the BIOS, before any off-chip memory is used) to prevent physical memory attacks.

However, after RustMonitor is launched, the primary OS is demoted to the normal mode, and can be under the control of the attacker, who may try to compromise the RustMonitor or enclaves, e.g., try to access the protected memory directly or through DMA. We consider the enclave code may be malicious or controlled by an attacker due to memory bugs. Our design needs to prevent a compromised enclave from contaminating the other enclaves or the RustMonitor. We also prevent the attacks against the primary OS or the application code, such as those presented in [63]. Similar to other TEEs, in this paper we do not focus on the prevention of denial of service (DoS) attacks or side channel attacks, such as cache timing and speculative execution attacks [48].

3 Design

HyperEnclave is designed to support confidential cloud computing without requiring specific hardware features. Therefore, HyperEnclave is built upon the widely available virtualization extension. In particular, HyperEnclave is designed to support the process-based TEE model (similar to Intel SGX) for the following reasons.

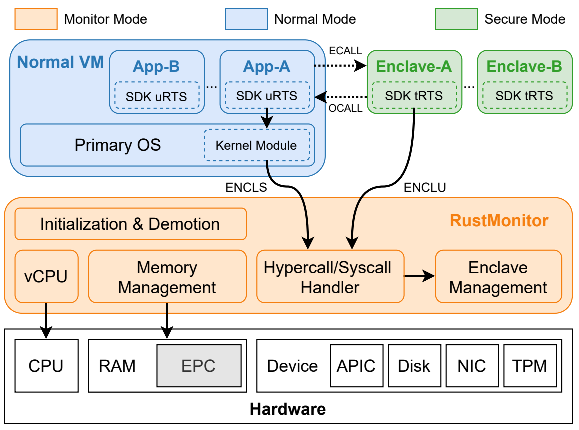


Figure 1: System Overview.

- *Minimized TCB*. To protect an application using the process-based TEE, the TCB includes only the protected code itself, while in the other forms of TEE, much more code must be included, such as the guest operating system for VM-based TEEs.
- *Established ecosystem*. Since Intel SGX is currently the most prevalent TEE supported in the cloud (major CSPs, including GCP, Azure, and Aliyun, provide SGX-based instances [9, 62]), a rich set of toolchains and applications have been developed. Supporting the SGX model reduces the porting effort and makes it easy to deploy confidential computing tasks in the cloud.
- *Cloud computing trends*. We have witnessed a clear trend towards running container-based serverless applications in the cloud. Protecting these applications against untrusted clouds using TEEs is important. Considering that such computing tasks are typically short-lived, and favor a short start-up time, maintaining a VM seems to be too heavy-weight.

In this section, we introduce HyperEnclave using x86 notations, as we prototyped HyperEnclave on AMD servers.

3.1 System Overview

HyperEnclave supports the following modes: the monitor mode, i.e., VMX root operation mode; the normal mode for the primary OS and untrusted part of applications, i.e., ring-0 and ring-3 of the VMX non-root operation mode respectively; and the secure mode for the enclave, which could be ring-3 and ring-0 of the VMX non-root operation mode, or ring-3 of the VMX root operation mode, depending on the *enclave operation mode*. We will introduce the flexible operation mode supported by HyperEnclave in Sec. 4. As illustrated in Figure 1, HyperEnclave consists of the following components:

- *RustMonitor* is a lightweight hypervisor running in the monitor mode that manages the enclave memory, enforces the memory isolation, and controls the enclave state transitions. It works as a resource monitor, while complicated tasks are offloaded to the primary OS.
- RustMonitor creates a *unique* guest VM (referred to as the *normal VM*) that runs the *primary OS* (such as Linux) and hosts the untrusted part of applications in the normal mode. The primary OS is still in charge of process scheduling and I/O devices management, but it is not trusted by the RustMonitor and enclaves.
- *Application* is the untrusted part of the application which runs in the primary OS.
- *The kernel module.* We provide a kernel module in the primary OS to load, measure, and launch RustMonitor, as well as to invoke the emulated privileged operations.
- To ease development, HyperEnclave provides an *enclave SDK* with APIs compatible with the official Intel SGX SDK [12], including both the untrusted runtime and trusted runtime (i.e., SDK uRTS and SDK tRTS). As such, most SGX programs can run on HyperEnclave with little or no source code changes.
- *Enclave* is the trusted part of the application running in the secure mode.

3.2 Memory Management and Protection

Challenges. For process-based TEEs, the enclave runs in the user mode and is not able to manage its own page table. Existing designs (e.g., Intel SGX, TrustVisor [54]) allow the untrusted OS to manage the enclave's page table. To prevent memory mapping attacks (i.e., attacks by manipulating the enclave's address mappings, as shown in Figure 9, Appendix A.1), the design of SGX extends the Page Missing Handler (PMH) and introduces a new metadata called EPCM for additional security checks on TLB misses [32]. Without secure hardware support, a prevalent software solution [19, 54, 75] is to make the page tables write-protected by setting the page table entries (PTEs) for pages holding the page tables, i.e., any update to the page table traps to the hypervisor and then be verified. However, on x86 platforms the updates of access and dirty bits of the PTEs also trap into the hypervisor, leading to non-negligible overhead. Even-worse, since the enclave page fault is also processed by the OS, the above designs are still vulnerable to the page table-based-attacks, such as the controlled-channel attacks [74].

The design becomes more challenging to support enclave dynamic memory management (i.e., EDMM on SGX2 platforms [34]), i.e, dynamically adding or removing enclave pages, or changing the enclave page attributes or types after the enclave is initialized. Without EDMM, all physical memory that the enclave might ever use must be committed before enclave initialization. Therefore, EDMM reduces enclave build time and enables new enclave features, such as on-demand stack and heap growth, and on-demand creation of code pages to support just-in-time (JIT) compilation. On SGX2 platforms, the enclaves need to send the EDMM request to the SGX driver through OCALLs, who then makes the requested changes. Since the driver is untrusted by the enclaves, the changes need to be explicitly checked and accepted by the enclaves to take effect, which involves heavy enclave mode switches.

HyperEnclave memory management. We observe that the above challenges are rooted in the fact that the enclave's page table and page faults are both managed by the primary OS. In HyperEnclave, though the enclave is still part of the application's address space, we create a separate page table for the enclave and let RustMonitor manage the enclave's page table and page fault without the involvement of the primary OS[3], while the page tables in the normal VM are still managed by the primary OS. However, the design faces new challenges: since the enclave can access the application's entire address space, upon a change to the mapping of the page tables in the applications, e.g, due to page swapping, the updated mapping needs to be synchronized to the enclave's page table managed by RustMonitor.

To eliminate the overhead for synchronization, we pre-allocate a *marshalling buffer* in the application's address space, which is shared with the enclave. The mappings of the marshalling buffer are fixed during the entire enclave life cycle by pre-populating the physical memory and pinning it in the memory. All data exchanged between the enclave and the application must be passed through the marshalling buffer. The application's memory mappings (except those for the marshalling buffer) are not needed by the enclave and are not included in the enclave's page table. Such a design also mitigates the known enclave malware attacks [63], as the enclave cannot access the application's address space but the marshalling buffer (Sec. 6 for more details). We remind the attacker may manipulate the marshalling buffer, however it does not cause additional security issues, since the buffer is untrusted by design where the developer is responsible to ensure that the data transmitted through the buffer is authentic and protected (same as the SGX model).

When the enclave accesses a virtual address that is not committed with a physical page (e.g., due to page swapping or EDMM), a page fault is raised and the enclave traps to RustMonitor. RustMonitor picks up a free page from the enclave memory pool, inserts a new mapping to the enclave's page table, and resumes the enclave's execution. When the enclave requests changing the page permissions, the enclave issues a hypercall to RustMonitor to update the permissions in the enclave's page table and clear the corresponding TLB entries.[4]

HyperEnclave memory isolation. Figure 2 shows the mem-

[3]P-Enclave can manage its own guest page table (Sec. 4.3).
[4]P-Enclave can change the page permissions by itself (Sec. 4.3).

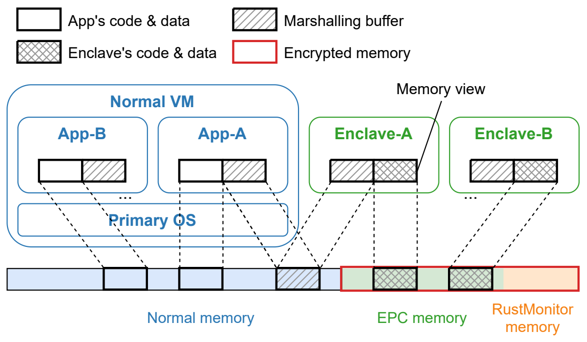


Figure 2: Memory isolation.

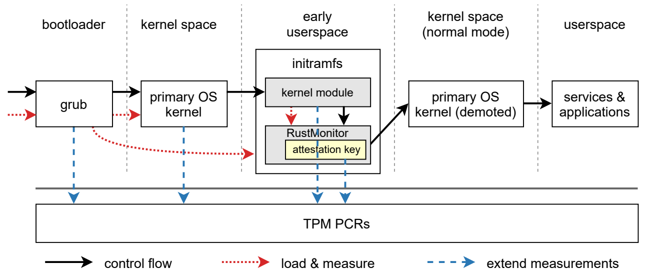


Figure 3: Measured Late Launch.

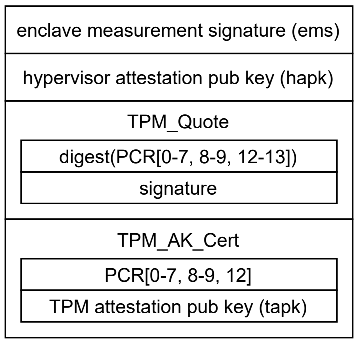

enclave measurement signature (ems)
hypervisor attestation pub key (hapk)
TPM_Quote
digest(PCR[0-7, 8-9, 12-13])
signature
TPM_AK_Cert
PCR[0-7, 8-9, 12]
TPM attestation pub key (tapk)

Figure 4: The HyperEnclave quote structure.

ory mappings of the applications within the normal VM and the enclaves. The application's memory within the normal VM is managed with nested paging, while the enclave's memory could be managed through nested paging or through normal 1-level address translation, determined by the corresponding operation mode (Sec. 4). As a result, HyperEnclave enforces the following security requirements.

- **R-1:** The primary OS and applications are not allowed to access the physical memory belonging to RustMonitor and the enclaves.
- **R-2:** The enclave is not allowed to access physical memory belonging to RustMonitor and other enclaves. It is designed to have access to only a specific memory region shared with the untrusted application for parameter passing (i.e., the marshalling buffer).
- **R-3:** DMA accesses from malicious peripherals to the physical memory belonging to RustMonitor and the enclaves are not allowed. In order to prevent such attacks, HyperEnclave restricts the physical memory used by the peripherals with the support of the Input-Output Memory Management Unit (IOMMU) in modern processors.

Memory encryption. To thwart physical memory attacks, such as cold boot and bus snooping attacks, HyperEnclave may leverage hardware memory encryption (such as AMD SME [44] and Intel MKTME [42]) to encrypt partial physical memory at the page granularity. If the platform does not support hardware memory encryption, HyperEnclave may consider to apply software approaches [76] to encrypt the isolated memory. This approach, however, may impose substantial overhead compared with hardware based solutions.

3.3 Trusted Boot, Attestation and Sealing

Measured Late Launch. The boot process of HyperEnclave is shown in Figure 3. On system boot, a static and immutable piece of code, known as the Core Root of Trust for Measurement (CRTM), executes first to bootstrap the process of building a measurement chain for subsequent firmware and software, including the BIOS, grub, the primary OS kernel, and initramfs. The measurements are stored to TPM PCRs for each boot component, so that any modification will be reflected in the attestation quote.

To reduce the attack surface from the primary OS, we put the RustMonitor image into the initramfs. The kernel measures the RustMonitor image and extends the value to TPM PCRs, then it launches RustMonitor in *early userspace*, i.e., before any userspace program that relies on the disk file system starts to run. Along with the measured boot, it ensures that the software state when RustMonitor is loaded is trusted.

After RustMonitor is loaded, the execution continues at the pre-defined entry. RustMonitor sets up its own running context (such as the stack, page table, IDT, etc.) and prepares the virtual CPU (vCPU) configurations for each CPU. Then RustMonitor launches the normal VM and demotes the primary OS to the normal mode. Returning to the kernel module, the kernel continues to boot in the normal mode and is unaware of the existence of RustMonitor.

HyperEnclave applies the above approach (referred to as *measured late launch*) so that RustMonitor is loaded as a type-2 hypervisor (like KVM) while runs as a type-1 hypervisor (like Xen). In this way, RustMonitor does not need to trust the primary OS anymore after the primary OS is demoted to the normal mode.

Remote Attestation. With the measured late launch, all booted components are measured and extended to the TPM. After RustMonitor is booted, it needs to extend the trust to the enclaves. For this purpose, RustMonitor derives an attestation key pair which is used to sign the enclave measurement. Then RustMonitor extends the derived public key to the TPM

PCR, and the private key never leaves RustMonitor which is protected by memory isolation and encryption.

During enclave creation, all pages added to the enclave (including the corresponding page content, page type, and RWX permissions) are measured by RustMonitor to generate the enclave measurement. The (intermediate) measurement is stored in RustMonitor's memory, which is invisible to the enclaves and the primary OS.

Similar to TPM and Intel SGX, HyperEnclave adopts a SIGn-and-MAc (SIGMA) attestation protocol for the remote attestation flow. As shown in Figure 4, we denote the public key of RustMonitor's attestation key by the hypervisor attestation public key (hapk). The enclave measurement is signed using RustMonitor's attestation key to form the enclave measurement signature (ems). The TPM quote TMP_Quote, which is signed using the TPM attestation key, includes the PCRs for the measurement of all booted code, and the measurement of hapk. Upon receiving the attestation report, the remote user can verify the report by comparing the measurement of booted code (including the CRTM, BIOS, grub, kernel, initramfs, and hypervisor) and the enclave, as well as verifying the certificate chain for generating the signature.

Secret key generation. When RustMonitor is initialized for the first time, it generates a root key K_{root} from the random number generator (RNG) module of the TPM. K_{root} is stored outside the TPM using TPM's seal operation. During the booting process on system reset, RustMonitor decrypts K_{root} using TPM's unseal operation, which guarantees that K_{root} can only be unsealed with the exactly same TPM chip with matching PCR configurations. Furthermore, RustMonitor floods the PCRs with a constant before transferring control to the primary OS to prevent it from retrieving K_{root}. All other key materials, including the enclave's sealing key and report key are derived from both K_{root} and the enclave's measurement.

3.4 The Enclave SDK

Porting existing applications to the enclaves can be cumbersome since TEEs usually expose limited hardware and software interfaces and provide additional security services (e.g., attestation and sealing). For process-based TEEs, the applications need to be partitioned into the trusted and untrusted parts, and the interfaces need to be carefully designed to avoid various security pitfalls [27, 46, 69]. A lot of effort has been spent and many tools have been developed for Intel SGX, due to its dominant position in the market, including library OSes [64, 67], containers [18], automatic partition and protection tools [50, 68], WebAssembly Micro Runtime [57], and interface protection [65]. Consequently, Intel SGX has supported securely running applications written in C/C++, Rust, Java, Python, etc., without expensive code refactoring.

We provide the enclave SDK with APIs compatible with the official Intel SGX SDK to ease the development of applications on HyperEnclave. The enclave SDK is retrofitting the official SGX SDK. By replacing the SGX user leaf functions (e.g., EENTER, EEXIT, and ERESUME) with hypercalls, SGX programs can run on HyperEnclave with little or no source code changes. Once the enclave executes these user leaf functions, it traps to RustMonitor and RustMonitor emulates the functionalities of the corresponding SGX instructions.

The enclave is compiled as a trusted library of the application, while the application itself runs in the primary OS. The enclave life cycle is managed through the emulation of a set of privileged SGX instructions (i.e., ECREATE, EADD, EINIT, etc.). To this end, the kernel module running in the primary OS provides similar functionalities by invoking RustMonitor through hypercalls, and exposes the functionalities to the applications by the `ioctl()` interfaces. By emulating the privileged SGX instructions, RustMonitor is responsible for the management of the enclave's life cycle (Sec. 4).

To be compatible with the official Intel SGX SDK, most data structures involved in HyperEnclave (such as the SIGSTRUCT structure, the SECS page, and the TCS page) are similar to that of SGX. With the HyperEnclave design, it is straightforward to support dynamic enclave management in an enclave, since the enclave memory and page fault are all managed by RustMonitor. Multi-threading within the enclave is supported by associating one TCS page for each enclave thread within the enclave. Exception handling within the enclave is supported by setting more than 1 SSA page for each TCS. The details are omitted due to space constraints and we refer the readers to the SGX manual [11] for more details.

4 Flexible Enclave Operation Mode

A wide range of existing applications can be offloaded to the TEEs, such as computing-intensive tasks (machine learning [60]), input and output (IO)-intensive tasks (such as the Apache and Nginx web server [18]), memory-intensive tasks (Redis and Memcached [18]), and tasks which favor in-enclave exception handling and privilege separation [21]. Most TEEs support running the enclaves only in fixed mode, Intel SGX (also TrustVisor [54] and Secage [51]) enclaves in particular, as part of the application address space, run in user mode. As a result, the user mode enclave is not allowed to access the privileged resources (such as the IDT and page tables) and process the privileged events (interrupt and exceptions). It must switch to the untrusted code to gain access to privileged resources and handle the events. The I/O-intensive and memory-intensive tasks essentially involve the frequent world switches which are expensive and introduce non-negligible performance losses, even though both software and hardware optimizations have been proposed trying to reduce the context switch latencies [61, 66, 73]. In this section, we introduce the three enclave operation modes supported by HyperEnclave, as shown in Figure 5. The world switches in different enclave operation modes are shown in Figure 6.

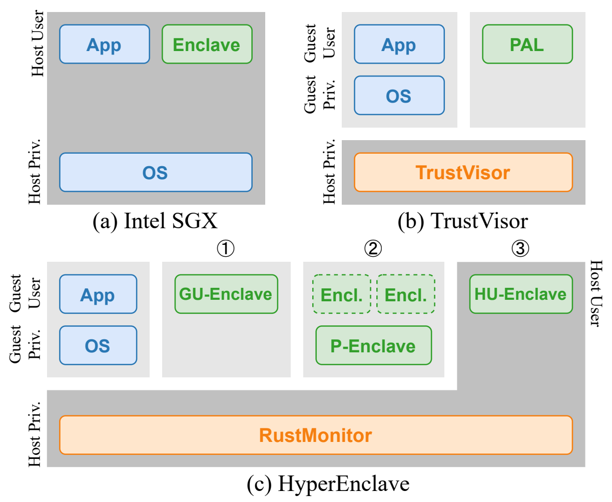
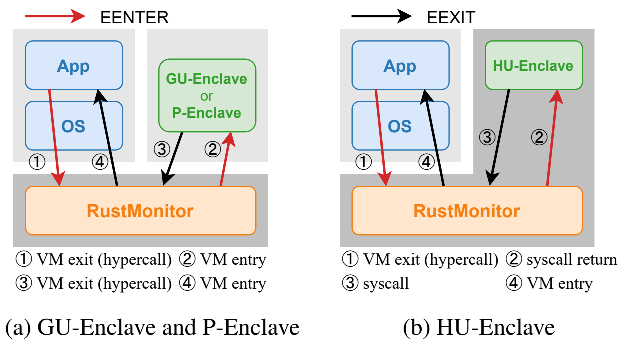


Figure 5: Comparison of the enclave operation modes supported by process-based TEEs. (a) Intel SGX runs enclaves in the host user mode (or guest user mode in the virtualization environment). (b) TrustVisor runs the protected code (Pieces of Application Logic, PALs) in the guest user mode. (c) HyperEnclave supports 3 coexisting enclave operation modes: ① GU-Enclaves running in guest user mode; ② P-Enclaves running in guest privileged mode and optional guest user mode; ③ HU-Enclaves running in host user mode.



EENTER
EEXIT
App
OS
GU-Enclave or P-Enclave
RustMonitor
① VM exit (hypercall) ② VM entry
③ VM exit (hypercall) ④ VM entry
(a) GU-Enclave and P-Enclave
App
OS
HU-Enclave
RustMonitor
① VM exit (hypercall) ② syscall return
③ syscall ④ VM entry
(b) HU-Enclave

Figure 6: World switches for the supported enclave operation modes. (a) Using hypercalls to enter and exit GU-Enclave and P-Enclave. (b) Using syscalls and syscall returns to enter and exit HU-Enclave.

4.1 Guest User Enclaves

Guest user enclave (GU-Enclave) is the basic enclave operation mode which is typically running computing-intensive tasks. The enclave runs in the guest user mode (i.e., guest ring-3 of the VMX non-root operation mode).

During the enclave creation, RustMonitor prepares a vCPU structure which contains a guest page table (GPT) and a nested page table (NPT) for GU-Enclave. On entry and exit between the normal VM and the enclave VM, RustMonitor switches the vCPU states (e.g. the instruction pointer, thread pointer, NPT, and GPT) accordingly.

To handle the interrupts and exceptions during the enclave running, RustMonitor configures the vCPU to trap all interrupts and exceptions to the monitor mode. RustMonitor then saves the enclave's context, forwards the interrupt or exception to the normal VM. After the primary OS completes handling the interrupt or exception, the application invokes the ERESUME hypercall, which traps to RustMonitor to restore the enclave's context and resume the execution of the enclave.

4.2 Host User Enclaves

Host user enclave (HU-Enclave) is running in host user mode. It delivers the optimal world switch efficiency by substituting the mode switch (hypercalls: $\sim$ 880 CPU cycles on our platform) with the ring switch (syscalls: $\sim$ 120 CPU cycles on our platform) (Figure 6). It further eliminates the extra virtualization overhead (e.g. vCPU context switching and two-dimensional page walking) in GU-Enclave. HU-Enclave may benefit the I/O-intensive workload according to our evaluation in Sec 7. By comparison, running enclaves in the guest user mode provides more defensive depth.

When loading the HU-Enclave, RustMonitor prepares a process context, e.g. creates a level-1 page table. On enclave entry, RustMonitor updates the CPU state and invokes the system call return instruction (i.e., `SYSRET` on x86 platforms) to enter the HU-Enclave. Correspondingly, on enclave exit, HU-enclave invokes the system call instruction (i.e., `SYSCALL` on x86 platforms) and traps into RustMonitor. The ENCLU leaf instructions (e.g., EGETKEY, EREPORT) are emulated as a system call. Interrupts and exceptions within the HU-Enclaves also trap into the RustMonitor. The procedures are similar to those for the GU-Enclaves described in Sec. 4.1.

4.3 Privileged Enclaves

Inspired by the VM-based TEEs, such as AMD SEV [45], HyperEnclave supports privilege enclaves (P-Enclaves) which run in guest privileged mode. P-Enclave is permitted to access the GDT, IDT, and level-1 page table which benefits a wide variety of applications, as demonstrated by Dune [21]. One such example is the garbage collector, an essential feature for Java applications (existing works port the JVM to enclaves [26, 43]). The garbage collector frequently changes page permissions to trigger page faults in order to track the page status. For user mode enclaves (e.g., GU-Enclaves and HU-Enclaves), it has to involve the primary OS to update the page table and handle the page fault which suffers huge performance loss due to world switches. P-Enclaves eliminate the world switch by supporting in-enclave exception handling and level-1 page table management. More specifically, P-Enclaves configures its own exception handler to handle certain exceptions (such as page fault). RustMonitor passes through the white-list exceptions to the P-Enclave and forwards others to the primary OS. Furthermore, P-Enclaves can also support page-table-based in-enclave isolation schemes, e.g., sandboxing untrusted third-party libraries.

With the ability to receive interrupts within the enclaves, P-Enclaves may also detect abnormal interrupt events by counting the frequency, before requesting RustMonitor to route them to the primary OS. As such, existing interrupt-based side channel attacks [24, 37, 40, 58, 59, 70] could be detected and mitigated. We leave further exploration in this direction to future work due to space constraints.

5 Implementations

We report our implementation of HyperEnclave on an AMD platform that supports hardware virtualization technology and memory encryption. In the current implementation, RustMonitor consists of about 7,500 lines of code written mostly in Rust, and the kernel module for the primary OS has about 3,500 lines of C code. Also, we made about 2,000 lines of code changes to the official Intel SGX SDK (version 2.13).

5.1 RustMonitor

RustMonitor runs at the highest privilege level and enforces the isolation for the enclaves. To reduce the risks caused by memory corruption or concurrency bugs, we implemented RustMonitor mostly in Rust, a memory-safe language, with only a few lines of assembly code used for context switches. Compared with existing hypervisors such as KVM [47] and Xen [29], RustMonitor is much smaller and thus easier to be formally verified. We are working on the formal verification of RustMonitor and plan to release the result as a separate report.

When the platform is booted, we configure the kernel command line parameters in the grub to reserve regions of physical memory, which are exclusively used by RustMonitor and the enclaves. RustMonitor manages the reserved physical memory by maintaining a list of free pages. When an enclave page is needed, e.g., when adding an enclave page during enclave creation, a free page is retrieved from the pool; when the enclave page is freed, the page is attached to the list again. Moreover, RustMonitor also manages the enclave's page tables and processes the page fault.

5.2 The Kernel Module

The kernel module is loaded by the primary OS during the booting process. Then it loads, measures, and launches RustMonitor, with the measurement extended to the TPM PCR as part of the TPM quote. When the kernel module is loaded, a device file is created and mounted at `/dev/hyper_enclave`. The application can open it and issue the `ioctl()` to invoke the emulated privileged operations.

5.3 The Enclave SDK

HyperEnclave retrofits the official SGX SDK as follows.

Supporting the SGX SDK APIs. We replace the SGX user leaf functions (e.g. EENTER, EEXIT, ERESUME, etc.) in the SGX SDK with hypercalls or system calls. Our implementation retains the same parameter semantics and orders as SGX for compatibility purposes.

Parameters passing with the marshalling buffer. In HyperEnclave, the enclave can only access its own address space and the marshalling buffer shared with the application. The size of the marshalling buffer can be configured in the enclave's configuration file, with a default size. The data needs to be transmitted to the marshalling buffer before invoking edge calls. We modified SGX SDK to handle the transitions, which are thus transparent to the developer.

We modified the untrusted runtime library in the SDK (i.e., `libsgx_urts.so`), such that during enclave initialization a marshalling buffer is allocated using `mmap()` with `MAP_POPULATE` flags set. As a result, the GPAs for the marshalling buffers are pre-populated. Then an `ioctl()` is issued to request the primary OS not to compact or swap out the physical pages of the marshalling buffers during the enclave's lifetime. When the application invokes the emulated EINIT instruction to mark the initialization of the enclave, the base address and the size of the marshalling buffer are passed to RustMonitor, who will add the mapping of the marshalling buffer in the enclave's page table. In this way, the marshalling buffer is now shared between the enclave and the untrusted application. The base address and the size of the marshalling buffer are also passed to the trusted runtime library to transmit data from the marshalling buffer to the enclave.

The current OCALL's implementation in the SGX SDK invokes the `sgx_ocalloc()` within the enclave to allocate a buffer on the stack area of the untrusted application, which is then used for cross-enclave data transmission. As such, we only need to modify the `sgx_ocalloc()` function to allocate a memory area in the marshalling buffer. To support parameter passing through the marshalling buffer for ECALLs, we modified SGX's Edger8r tool to automatically generate code that copies the transmitted data into the marshalling buffer.

The SGX programming model supports passing parameters with the `user_check` attribute. For such parameters, the SDK tool will not generate code to check the address range or perform data movement. Since the enclave code could access the entire process's address space, some enclave programs may use a pointer with the `user_check` attribute to manipulate the data buffer outside the enclave directly, without accounting for the overhead for copying the data across the enclave boundary. To deal with it, we added an interface for the developer to allocate the buffer within the marshalling buffer, in the cases when the developer may use parameters with the `user_check` attribute.

The remote attestation flow is similar to SGX, following the same SIGn-and-MAc (SIGMA) protocol. We extended the `sgx_quote_t` structure in the SDK to include the HyperEnclave quote, and the modification is transparent to the enclave code.

With the above design, most SGX programs could run on HyperEnclave without source code changes. Furthermore, to ease the development of HyperEnclave applications, we have also ported the Rust SGX SDK [71] and the Occlum library OS [64] to HyperEnclave.

6 Security Analysis

Trust Establishment. HyperEnclave relies on measured boot to bootstrap the trust of RustMonitor, a common approach for the design of TEEs (e.g., TrustZone [16] and Keystone [49]). All the components during booting (including the CRTM, BIOS, grub, kernel, and initramfs) are measured and extended to the TPM PCRs. As a result, any tampering of the booted code will be reflected and audited through remote attestation.

We consider HyperEnclave is deployed in a controlled environment (i.e., the data center in the cloud computing scenario) such that the attacker has limited physical access to the platforms. To reduce the attack surface and minimize the TCB, we put the RustMonitor image into the initramfs, and RustMonitor is loaded and measured in early userspace. In this phase, the primary OS kernel does not accept external inputs from the user, and the peripherals such as network connection are disabled. With the measured late launch approach, RustMonitor does not need to trust the primary OS anymore after the OS is demoted to the normal mode.

Enclave memory isolation. As presented in Sec. 3.2, the enclave's memory and page tables are maintained by RustMonitor, which are inaccessible to the primary OS. The TLBs are cleared upon world switches to prevent illegal memory accesses using stale TLB entries. RustMonitor prevents the primary OS to access the reserved physical memory by removing the corresponding mappings from its NPT. RustMonitor also configures the IOMMU to prevent unauthorized device accesses to the reserved physical memory.

Our design prevents the memory mapping attacks since the primary OS cannot interfere with the enclave's address mappings. We introduce the marshalling buffer to support memory sharing between the enclave and the application. The application pre-allocates a marshalling buffer in normal memory and passes the base address and size of the buffer to RustMonitor during enclave initialization. In case the application may pass crafted addresses (e.g., to overwrite the enclave memory), before adding the mapping of the marshalling buffer to the enclave's page table, RustMonitor ensures the address range of the marshalling buffer is outside the enclave address range.

Defense Against Compromised Enclaves. Previous work demonstrates that enclave malware may steal secret data or hijack the control-flow of the application outside the enclave [63]. HyperEnclave is designed to confine potentially malicious enclaves as follows.

- *Preventing arbitrary memory accesses to the application.* The SGX enclave could access the entire address space of the application, and it is possible for attacks such as leaking the secret keys or stack canaries, or tampering with the code pointers for control flow attacks. In HyperEnclave, the enclave can only access its own memory and the marshalling buffer, which is only used for parameter passing.
- *Preventing arbitrary control flows after EEXIT.* The SGX design allows the enclave to jump to arbitrary addresses by setting rbx before executing the EEXIT instruction (i.e., exiting the enclave), opening door to enclave malware attacks [63]. In our design, since the EEXIT instruction is emulated by RustMonitor, it is easy to prevent such attacks by adding the validity check when EEXIT is invoked.

Physical attacks. Hardware memory encryption techniques such as AMD SME could be used to protect the enclave from physical attacks such as cold boot attacks or bus snooping attacks. With memory encryption, data are always encrypted in the memory or on the memory bus, and are only decrypted within the CPU. The memory encryption key is generated randomly on system boot and stored in the CPU, which cannot be accessed explicitly by software.

Side channel attacks. Compared with SGX, HyperEnclave can mitigate certain types of side channel attacks. Since the enclave's guest page tables and page fault events are processed by RustMonitor without primary OS involvement, the latter cannot mount page-table-based attacks [25, 72, 74]. We leave the protection against micro-architectural attacks such as speculative execution attacks as future work.

7 Evaluation

We deployed HyperEnclave on a server with two AMD EPYC 7601 CPUs (2 threads per core, total of 128 logical cores) with 512 GB DDR4 RAM. We configure 2 GB reserved memory for RustMonitor, and 24 GB for EPC memory. The primary OS is Ubuntu 18.04 LTS with Linux kernel 4.19.91. For comparison, we run the same experiments on an Intel Xeon E3-1270 v6 CPU with SGX enabled, with 64 GB DDR4 RAM, running the same OS. The SGX SDK version for both HyperEnclave and the native SGX hardware are 2.13. All programs are compiled with GCC 7.5.0 and the same optimization level.

We tried to rule out differences between hardware. Except where explicitly stated, the evaluations didn't exceed the EPC size, so as not to trigger excessive page swapping. All evaluations were performed in single-threaded mode. For micro-benchmarks (Table 1 and Table 2), we compare HyperEnclave on AMD hardware with SGX on Intel hardware using the same SGX SDK. We measured the core cycles to avoid the influence of CPU frequencies. For real-world workloads, we set the baselines as the counterparts with no security protections on Intel and AMD platforms respectively, and compare

	EENTER	EEXIT	ECALL	OCALL
Intel SGX	-	-	14,432	12,432
HU-Enclave	1,163	1,144	8,440	4,120
GU-Enclave	1,704	1,319	9,480	4,920
P-Enclave	1,649	1,401	9,700	5,260

Table 1: Latency of SGX primitives on HyperEnclave and Intel (in CPU cycles).

	Intel SGX	GU-Enclave	P-Enclave
`#UD`	28,561	17,490	258
`#PF`	–	2,660	1,132

Table 2: Average CPU cycles of handling an `#UD` and `#PF` exception inside the enclaves.

the *relative slowdowns* introduced by SGX and HyperEnclave. Since we only compare the relative slowdowns, we stress that the *absolute* performance results are on dissimilar platforms and are not directly comparable. All HyperEnclave evaluations were measured with memory encryption enabled.

We've been careful in ensuring HyperEnclave implements the TEE functionality correctly. Still, memory encryption between SGX1 and HyperEnclave is different, i.e., Merkel tree and AES-CTR versus AES-XTS (see Figure 11 in Sec. A.3 for the evaluation of memory encryption overhead), which may explain the improvement for memory-intensive workloads. Besides, world switches for HyperEnclave (especially, HU-enclave) are faster, which explains the improvement for I/O-intensive workloads.

7.1 World Switches Performance

We measured the latency of edge calls (i.e., ECALLs and OCALLs) on both HyperEnclave (under different enclave operation modes) and Intel SGX. The test code runs empty edge calls with no explicit parameters 1,000,000 times and takes the median value. We also measured the instruction-level latency for the emulated EENTER and EEXIT instructions on HyperEnclave. We were not able to measure the instruction-level latency on SGX since the `RDTSCP` instruction is not supported within the enclaves on our SGX platform.

The results are shown in Table 1. It shows that HU-Enclave has the optimal edge calls performance as it reduces a mode switch ($\sim$ 880 cycles) to a ring switch ($\sim$ 120 cycles) while P-Enclave is slower than GU-Enclave for it needs to switch more privileged states during the world switches. All of the results are comparable with Intel SGX.

7.2 Enclave Exception Handling

We used the undefined instruction exception (`#UD`) and the page fault exception (`#PF`) to evaluate the enclave exception handling performance. In the `#UD` benchmark, the test code executes an undefined instruction in the enclave to trigger the exception 1,000,000 times. The exception handler advances the instruction pointer and returns. For P-Enclave, the exceptions are captured and handled entirely within the enclaves, without enclave mode switches. For GU-Enclave and SGX, an exception causes an asynchronous enclave exit (AEX) and switches the CPU to the untrusted OS, then executes a two-phase exception handling [7]. The result shows that the exception handling within P-Enclaves is about 68× and 110× faster than GU-Enclave and Intel SGX respectively (Table 2).

We further simulated a typical garbage collector (GC) scenario that the test code first allocated a large memory buffer, then the write permissions to the buffer were revoked by changing the enclave's page table. After that, the enclave accessed the buffer to trigger the page faults. In the exception handler, the write permission is restored. The result (Table 2) shows that P-Enclave is about 2.3× faster than GU-Enclave, for P-Enclave updates the page table and handles the page faults by itself, while GU-Enclave needs to trap into RustMonitor to update the page tables. Note that we did not evaluate GC on Intel SGX, since our SGX1 platform does not support page permission modifications after the enclave initialization.

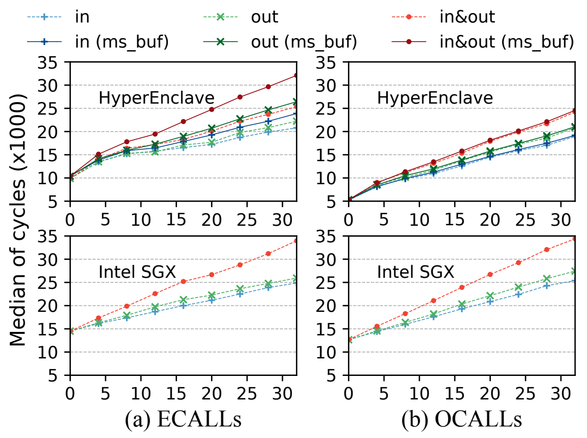
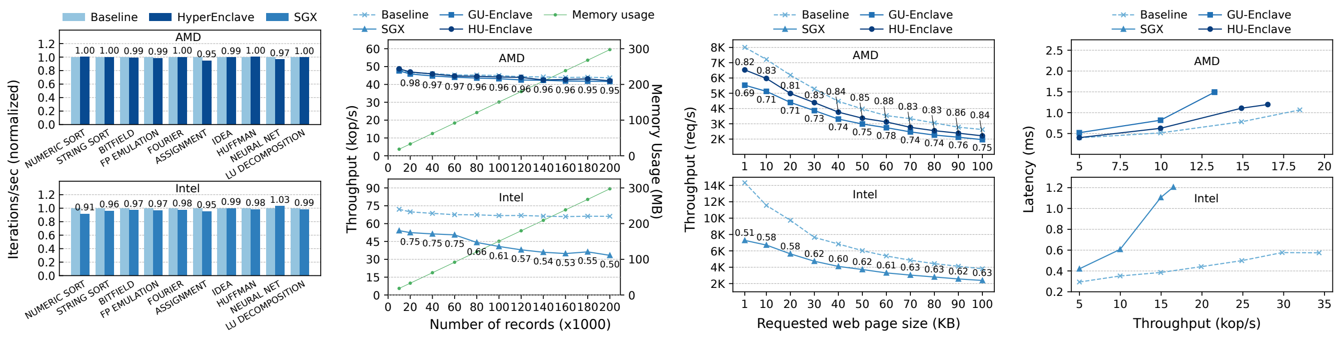


Figure 7: Marshalling buffer overhead for ECALLs and OCALLs with various data size, and with the marshalling buffer (marked as ms_buf) enabled and disabled.

7.3 Marshalling Buffer Overhead

To measure the overhead of introducing the marshalling buffer, we constructed a GU-Enclave variant that does not use the marshalling buffer as the baseline. We measured the overhead for both ECALLs and OCALLs with various sizes of the transferred data while varying the directions for data movement (i.e., "in", "out" and "in&out"). We ensure the data to be transferred is not cached using the `CLFLUSH` instruction. We evaluated the performance for transferring the same data on SGX for comparison.

Figure 7 provides the results for ECALLs and OCALLs respectively. It shows that the overhead increases almost lin-



(a) NBench (b) SQLite (c) Lighttpd on Occlum (d) Redis on Occlum

Figure 8: Performance of NBench, SQLite, Lighttpd, and Redis on AMD (with HyperEnclave) and Intel (with SGX).

early with the data size. For ECALLs, the overhead for "in", "out" and "in&out" directions is 8%, 11% and 21% respectively for transferring 16 KB data, due to the extra memory copy. For OCALLs, the overhead is negligible, since it allocates a buffer on the marshalling buffer without additional memory copy (Sec. 5.3). We remind that data transfers in ECALLs contribute a small portion to the processing time for many real-world computation workloads, especially for computation or memory intensive tasks.

7.4 Real-world Workloads

The evaluations were conducted on four real-world applications: an algorithm benchmark suite **NBench** [53], a lightweight web server **Lighttpd** [13], two popular databases **SQLite** [14] and **Redis** [15], as representations for CPU intensive, I/O-intensive, and memory-intensive tasks. We ported the library OS Occlum [64] (v0.21) to the enclave SDK to reduce the porting effort for Lighttpd and Redis. We measured the performance on both HyperEnclave and Intel SGX, using the same code compiled under the SDK simulation mode as the baseline (providing no security guarantees).

NBench. NBench measures the performance of a system's CPU, FPU, and memory system, without I/O and system calls involved. We used an adaptation of NBench to SGX, i.e., SGX-NBench [8] with no source code modification for our evaluation. As shown in Figure 8a, the overhead introduced by HyperEnclave and SGX is about 1% and 3% respectively.

SQLite. We ported SQLite (v3.19.3) with the enclave SDK, and evaluated it on both Intel SGX and HyperEnclave (GU-Enclave and HU-Enclave) using the YCSB [30] workload A (50% reads, 50% updates). In this evaluation we focused on the memory performance, so we configured the database as in-memory and embedded the client into the enclave to avoid I/O operations. We increased the number of records and measured the time for 100,000 database operations. As shown in Figure 8b, on SGX the throughput is about 75% of the baseline for small memory usage. When the memory usage exceeds the EPC size (about 90 MB), the performance drops to 50% due to page swapping. On HyperEnclave, both GU-Enclave and HU-Enclave have almost the same performance as the baseline (< 5% overhead). We speculate that it's because the memory encryption performance for AMD SME (without integrity protection) is faster than SGX.

Lighttpd. We ran a Lighttpd (v1.4.40) server with Occlum on both SGX and HyperEnclave (GU-Enclave and HU-Enclave modes). We used the Apache HTTP benchmarking tool [10] and ran 100 concurrent clients over the local loopback to fetch various sizes of web pages to evaluate the throughput. In this evaluation, the overhead mainly comes from the frequent enclave mode switches (Table 1). As shown in Figure 8c, HU-Enclave delivers the best performance as expected (81% $\sim$ 88% of the baseline). GU-Enclave achieves 69% $\sim$ 78% of the baseline, while SGX achieves 51% $\sim$ 63% of the baseline.

Redis. We use Redis to evaluate the performance under the comprehensive scenarios where both memory and I/O are intensive. We ran a Redis (v6.0.9) database server with Occlum on both SGX and HyperEnclave (GU-Enclave and HU-Enclave modes). Similar to SQLite, we configured the database as in-memory and used the YCSB workload A. For the evaluation, we first loaded 50,000 records (in total 50 MB data) and then performed 100,000 operations from 20 clients over the local loopback. We increased the request frequency and measured the latency under different throughput. As shown in Figure 8d, HU-Enclave achieves 89% of the maximum throughput of the baseline, while GU-Enclave and SGX are about 72% and 48% of the baseline, respectively.

8 Discussions

HyperEnclave on other platforms. HyperEnclave requires the virtualization extension (specifically, two-level address translation) for isolation and TPM for the root of trust and randomness, etc. Virtualization is supported on many ARM servers (such as the ARMv8 platforms [2]). The RISC-V H-extension specification has evolved to v0.6.1 in 2021. Both ARM and RISC-V virtualization support two-level address translation. Certain TPM products already support ARM servers. Research has been conducted to support firmware TPM on RISC-V [22]. As such, signs are promising that

HyperEnclave can be adapted to run on ARM and RISC-V platforms.

However, porting HyperEnclave to ARM and RISC-V platforms requires non-trivial engineering effort, considering that the instruction set architectures (ISAs) are totally different. Take the ARMv8 architecture as an example. The software modules can be mapped to different exception levels (ELs): The monitor mode for RustMonitor can be mapped to EL2; The normal mode for the primary OS and untrusted part of the applications can be mapped to EL1 and EL0 respectively; The secure mode for enclaves can be mapped flexibly to EL1 or EL0. Memory isolation can be supported similarly with the support of stage 2 address translations. Furthermore, the official Intel SGX SDK only supports x86 platforms. In particular, the transitions across enclave boundaries are handled with platform-dependent assembly code, and need to be rewritten according to the application binary interface (ABI) of the targeted platforms. We leave the further exploration of adapting HyperEnclave to other platforms as future work.

Attack surfaces under different enclave operation modes. The untrusted primary OS still runs within the VM and the attack surface from the primary OS to enclaves does not change. Running the enclaves in privileged mode or in the host, however, may expose more attack surfaces to a malicious enclave. For example, it may make the enclave malware easier to escalate to host ring-0, if the enclave runs in the host already.

9 Related Works

Most existing TEEs require specific hardware or firmware changes [17, 33, 39, 39, 41, 45, 55]. Specifically, CURE [20] changes the CPU core to support the enclave identifier (eid), and modifies the system bus to support the memory and peripheral arbiters for memory and peripheral access control according to the eid. Then the trusted security monitor can configure the hardware primitives to support the flexible enclaves. In contrast, HyperEnclave supports flexible enclave modes on commodity hardwares without hardware or firmware changes.

A line of research has been conducted to build the isolated execution environment (e.g., PAL for TrustVisor and HAP for InkTag) using virtualization extensions, including TrustVisor [54], InkTag [38], Overshadow [28], AWS Nitro Enclaves [3], Microsoft Defender Credential Guard [5] and Hyper-V Shielded VMs [6]. TrustVisor makes the PAL page table pages read-only to prevent memory mapping attacks. The design introduces much overhead during PAL registration and switches from/to PALs. Even worse, it triggers many NPT violations in high memory pressures scenarios, due to the updates to the access and dirty bits of the PAL page tables, introducing huge overhead [52]. In Inktag, the HAP page tables are managed by the hypervisor. It allows the untrusted OS to request the hypervisor to update the HAP page tables. Both TrustVisor and InkTag are susceptible to page-table-based attacks [72, 74]. Furthermore, these designs usually contain a large code base in the TCB, including device drivers, guest IO emulation, network and block device virtualization, etc. For example, AWS Nitro Enclaves [3] are constructed by the host KVM and Linux, and thus the host Linux kernel is always trusted. We made the design choice to minimize the TCB of RustMonitor, which performs basic CPU/memory virtualization and enclave management. The primary OS kernel only needs to be trusted during the boot process and is demoted after RustMonitor starts.

Komodo [35] implements an SGX-like enclave protection model in the TrustZone environment. Keystone [49] supports customizable TEEs on RISC-V platforms. Komodo enclaves run in secure user mode, while Keystone enclaves run in U-mode and S-mode. In comparison, HyperEnclave supports flexible enclave mode (guest ring-3, guest ring-0/ring-3, and host ring-3). Both Komodo and HyperEnclave use page-table-based memory isolation, while Keystone uses PMP (physical memory protection) for memory isolation.

Recently, ARM introduced the Realm Management Extension (RME) in the forthcoming ARMv9-A architecture [1]. The monitor (running at EL3) enforces physical memory isolation among the secure, non-secure, and Realm worlds. The trusted Realm Management Monitor (RMM), which executes at EL2 in Realm security state (R-EL2), isolates the Realms from each other through the stage 2 page tables. Similar to HyperEnclave, the RMM is much simpler than a typical hypervisor and relies on the non-secure hypervisor for device emulation etc. RME requires architectural extensions, while HyperEnclave does not. Moreover, HyperEnclave decouples its trust chain from the CPU as much as possible to construct an open and cross-platform TEE.

10 Conclusion

In this paper, we proposed HyperEnclave, an open TEE that can run on various platforms with minimum hardware requirements. It supports the process-based TEE model, and SGX programs can run on HyperEnclave with little or no code changes. Moreover, HyperEnclave supports the flexible enclave operation modes to fulfill various enclave workloads. We implemented HyperEnclave on commodity AMD servers and deployed the system internally for real-world computations. We are working on the formal verification of the implementation and plan to open source to the community.

Acknowledgments

We would like to thank our shepherd David Cock and the anonymous reviewers for their invaluable feedback. This work was supported in part by the National Key R&D Program of China (Grant No. 2020YFB1805402) and the National Natural Science Foundation of China (Grant No. 61802397 and No. U19A2060).

References

[1] Arm Realm Management Extension (RME) System Architecture. https://developer.arm.com/documentation/den0129/latest.

[2] Armv8 white paper. https://community.arm.com/docs/DOC-10896.

[3] AWS Nitro Enclaves User Guide. https://docs.aws.amazon.com/enclaves/latest/user/nitro-enclave.html.

[4] ISO/IEC, P.D.: 11889: Information technology – Security techniques – Trusted platform module.

[5] Manage Windows Defender Credential Guard. https://docs.microsoft.com/en-us/windows/security/identity-protection/credential-guard/credential-guard-manage.

[6] Microsoft Hyper-V Shielded VM. https://www.techtarget.com/searchwindowsserver/definition/Microsoft-Hyper-V-Shielded-VM.

[7] SGX two phase exception handling. https://github.com/MWShan/linux-sgx/blob/master/docs/DesignDocs/IntelSGXExceptionHandling-Linux.md.

[8] The nbench benchmark ported to SGX. https://github.com/utds3lab/sgx-nbench, 2017.

[9] Google. Asylo. https://asylo.dev/, 2019.

[10] ab - apache http server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html, 2021.

[11] Intel 64 and IA-32 architectures software developer's manual, combined volumes:1,2A,2B,2C,3A,3B,3C and 3D. https://software.intel.com/content/dam/develop/external/us/en/documents-tps/325462-sdm-vol-1-2abcd-3abcd.pdf, 2021. Order Number: 325462-075US, June 2021.

[12] Intel SGX for Linux. https://github.com/intel/linux-sgx, 2021.

[13] Lighttpd. https://www.lighttpd.net, 2021.

[14] SQLite. https://www.sqlite.org, 2021.

[15] Redis. https://redis.io, 2022.

[16] T. Alves and D. Felton. Trustzone: Integrated hardware and software security. *white paper*, 2004.

[17] ARM. Arm Confidential Compute Architecture. https://www.arm.com/why-arm/architecture/security-features/arm-confidential-compute-architecture, 2020.

[18] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, 2016.

[19] Ahmed M. Azab, P. Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, G. Ganesh, Jia Ma, and Wenbo Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[20] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. CURE: A security architecture with customizable and resilient enclaves. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021.

[21] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 335–348, Hollywood, CA, October 2012. USENIX Association.

[22] Marouene Boubakri, Fausto Chiatante, and Belhassen Zouari. Towards a firmware TPM on RISC-V. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 647–650. IEEE, 2021.

[23] F. Brasser, D. Gens, P. Jauernig, A. R. Sadeghi, and E. Stapf. Sanctuary: Arming trustzone with user-space enclaves. In *Network and Distributed System Security Symposium*, 2019.

[24] 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)*, 2017.

[25] 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 *26th USENIX Security Symposium (USENIX Security 17)*, pages 1041–1056, Vancouver, BC, August 2017. USENIX Association.

[26] Chia che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca A. Popa, and Donald E. Porter. Civet: An efficient java partitioning framework for hardware enclaves. In *USENIX Security Symposium*, 2020.

[27] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. *ACM SIGARCH Computer Architecture News*, 41(1):253–264, 2013.

[28] Xiaoxin Chen, Tal Garfinkel, E Christopher Lewis, Pratap Subrahmanyam, Carl A Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan RK Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. *ACM SIGOPS Operating Systems Review*, 42(2):2–13, 2008.

[29] David Chisnall. *The definitive guide to the xen hypervisor*. Pearson Education, 2008.

[30] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[31] Standard Performance Evaluation Corporation. SPEC CPU 2017. https://www.spec.org/cpu2017/, 2021.

[32] Victor Costan and S. Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016:86, 2016.

[33] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, Austin, TX, August 2016. USENIX Association.

[34] McKeen F., Alexandrovich I., Anati I., Caspi D., Johnson S., Leslie H. R., and Rozas C. Intel software guard extensions (intel sgx) support for dynamic memory management inside an enclave. *Hardware and Architectural Support for Security and Privacy*, 2016.

[35] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *SOSP'17*, 2017.

[36] Trusted Computing Group. TPM Library Specification 2.0, 2016.

[37] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *2017 USENIX Annual Technical Conference (USENIXATC 17)*, pages 299–312, 2017.

[38] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. Inktag: secure applications on an untrusted operating system. *ASPLOS ... proceedings. International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 253–264, 2013.

[39] Guerney DH Hunt, Ramachandra Pai, Michael V Le, Hani Jamjoom, Sukadev Bhattiprolu, Rick Boivie, Laurent Dufour, Brad Frey, Mohit Kapur, Kenneth A Goldman, et al. Confidential computing for openpower. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 294–310, 2021.

[40] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. Bluethunder: A 2-level directional predictor based side-channel attack against sgx. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 321–347, 2020.

[41] Intel. Intel Trust Domain Extensions. https://software.intel.com/content/dam/develop/external/us/en/documents/tdxwhitepaper-v4.pdf, 2020.

[42] Intel. Intel Architecture Memory Encryption Technologies Specification. https://software.intel.com/content/dam/develop/external/us/en/documents-tps/multi-key-total-memory-encryption-spec.pdf, 2021.

[43] Jianyu Jiang, Xusheng Chen, Tsz On Li, Cheng Wang, Tianxiang Shen, Shixiong Zhao, Heming Cui, Cho-Li Wang, and Fengwei Zhang. Uranus: Simple, efficient sgx programming and its applications. *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020.

[44] David Kaplan. AMD x86 memory encryption technologies. Austin, TX, August 2016. USENIX Association.

[45] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. *White paper*, 2016.

[46] Mustakimur Rahman Khandaker, Yueqiang Cheng, Zhi Wang, and Tao Wei. Coin attacks: On insecurity of enclave untrusted interfaces in sgx. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 971–985, 2020.

[47] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230. Dttawa, Dntorio, Canada, 2007.

[48] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[49] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[50] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. Glamdring: Automatic application partitioning for intel SGX. In *2017 USENIX Annual Technical Conference (USENIXATC 17)*, pages 285–298, 2017.

[51] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1607–1619, 2015.

[52] Andrei Lutas, Daniel Ticle, and O. Creţ. Hypervisor based memory introspection: Challenges, problems and limitations. In *ICISSP*, 2017.

[53] Uwe F. Mayer. Linux/Unix nbench. https://www.math.utah.edu/~mayer/linux/bmark.html, 2017.

[54] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, V. Gligor, and A. Perrig. Trustvisor: Efficient tcb reduction and attestation. *2010 IEEE Symposium on Security and Privacy*, pages 143–158, 2010.

[55] 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. *Hasp, isca*, 10(1), 2013.

[56] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *USENIX 1996 Annual Technical Conference (USENIX ATC 96)*, San Diego, CA, January 1996. USENIX Association.

[57] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. Twine: An embedded trusted runtime for webassembly. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 205–216. IEEE, 2021.

[58] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How sgx amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 69–90. Springer, 2017.

[59] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. Copycat: Controlled instruction-level attacks on enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 469–486, 2020.

[60] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 619–636, 2016.

[61] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless os services for sgx enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 238–253, 2017.

[62] Mark Russinovich. Introducing Azure confidential computing. *Seattle, WA: Microsoft*, 2017.

[63] M. Schwarz, S. Weiser, and D. Gruss. Practical enclave malware with intel sgx. In *DIMVA 2019*, pages 177–196, 2019.

[64] 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 Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 955–970, 2020.

[65] Shweta Shinde, Shengyi Wang, Pinghai Yuan, Aquinas Hobor, Abhik Roychoudhury, and Prateek Saxena. Besfs: A POSIX filesystem for enclaves with a mechanized safety proof. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 523–540, 2020.

[66] Hongliang Tian, Qiong Zhang, Shoumeng Yan, Alex Rudnitsky, Liron Shacham, Ron Yariv, and Noam Milshten. Switchless calls made practical in Intel SGX. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, pages 22–27, 2018.

[67] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-sgx: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIXATC 17)*, pages 645–658, 2017.

[68] Chia-Che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E Porter. Civet:

An efficient java partitioning framework for hardware enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 505–522, 2020.

[69] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D Garcia, and Frank Piessens. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1741–1758, 2019.

[70] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 178–195, 2018.

[71] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. Towards memory safe enclave programming with rust-sgx. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[72] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2421–2434, 2017.

[73] Ofir Weisse, Valeria Bertacco, and Todd Austin. Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 81–93. IEEE, 2017.

[74] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656. IEEE, 2015.

[75] Minhong Yun and Lin Zhong. Ginseng: Keeping secrets in registers when you distrust the operating system. In *NDSS*, 2019.

[76] Shijun Zhao, Qianying Zhang, Yu Qin, Wei Feng, and Dengguo Feng. Sectee: A software-based approach to secure enclave architecture using TEE. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

A Supplementary Materials

A.1 Mapping Attacks

See Figure 9.

A.2 Virtualization Overhead

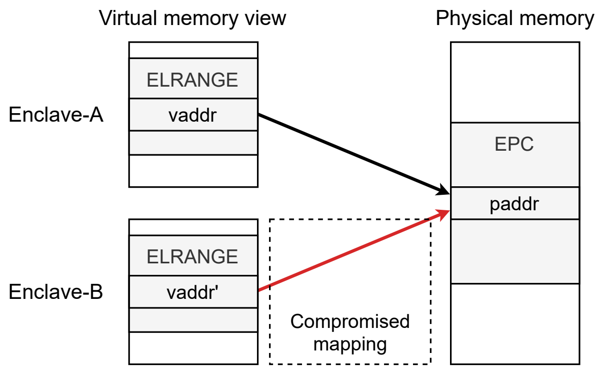
We'd like to evaluate the virtualization overhead on the normal VM. We ran SPEC CPU 2017 INTSpeed benchmarks [31], LMBench [56], and Linux kernel (v5.15) building when enabled and disabled RustMonitor. The result shows that the virtualization overhead is less than 1% in most benchmarks (see Figure 10 and Table 3). HyperEnclave avoids massive VM-exits by pass-through most devices to the normal VM and installs huge pages in NPT when possible to relieve the TLB pressure.

	LMbench (μs)						Kernel Build (s)
	null call	fork	ctxsw	mmap	Page Fault	AF UNIX	
Native	0.1195	196.3	3.13	66,125	0.2433	5.73	1,410
Normal VM	0.1192	197.9	3.22	66,407	0.2461	5.69	1,417
Overhead	-0.25%	0.82%	2.88%	0.43%	1.15%	-0.70%	0.50%

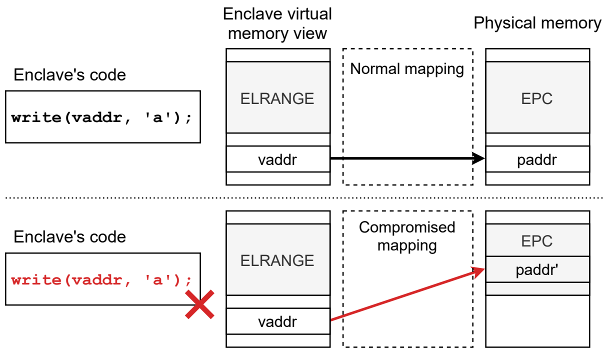
Table 3: Virtualization overhead on LMBench (null syscall, fork, context switches among 16 processes with 64KB working set, mmap, page fault, and unix socket) and building the Linux kernel.

A.3 Memory Encryption Overhead

We evaluate the memory encryption overhead by measuring the memory access latency with and without encryption in sequential and random access patterns. The buffer size is varied from 16 KB to 256 MB. Figure 11 illustrates the result on HyperEnclave and SGX. When the buffer size is smaller than the LLC size (8 MB), the overhead on both platforms is negligible. When the buffer size is over the LLC size, the overhead for sequential accesses and random accesses can be over 2.4× and 25× respectively on HyperEnclave, while on SGX is 3× and 30× respectively. When the buffer size exceeds the EPC size (93 MB), the overhead for sequential accesses and random accesses is 45× and 1000× slow on SGX, due to the EPC page swapping, while on HyperEnclave the overhead is still less than 30× since we reserve 24GB as enclave memory in our test.



(a) Alias mapping attack



(b) Remapping attack

Figure 9: Mapping attacks. (a) Two guest virtual addresses within the enclaves are mapped to the same guest physical address; (b) A non-enclave virtual address is mapped to the physical address belonging to the enclave.

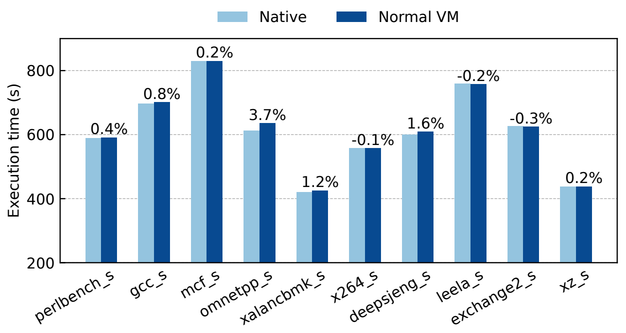


Figure 10: Virtualization overhead on SPEC CPU 2017.

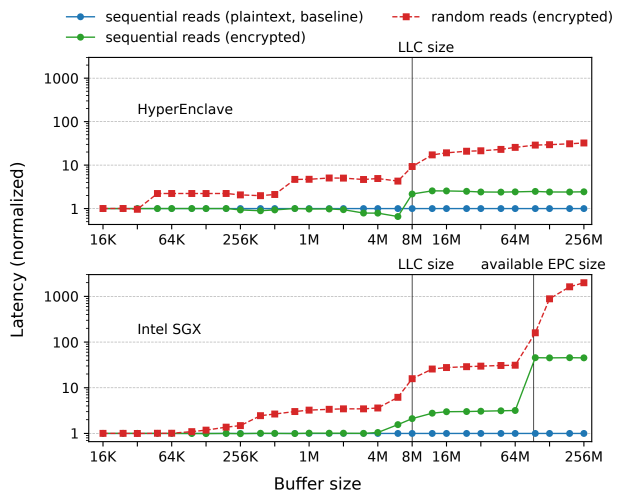


Figure 11: Memory encryption overhead for sequential and random memory accesses on HyperEnclave (with AMD SME) and Intel SGX (with Intel MEE). The LLC size is 8 MB, and the available EPC size on Intel SGX is about 93 MB.

B Artifact Appendix

Abstract

HyperEnclave can support existing SGX toolchains and run SGX applications on AMD CPU with security guarantees. This artifact contains the binaries of the RustMonitor, and documentations on how to setup the HyperEnclave environment. We provide two containers to reduce the environment configuration efforts. Specifically, the server container includes the pre-installed enclave SDK, the Occlum LibOS, and the benchmarks along with their dependencies[5]. The client container includes pre-installed client side benchmark scripts for Lighttpd and Redis.

Scope

The artifact includes benchmarks for edge calls (i.e., ECALLs and OCALLs), and benchmarks for the real-world workloads, including NBench, SQLite, Lighttpd and Redis. We provide scripts to reproduce the results in the paper (summarized in Table 4).

Contents

- `README.md` describes the artifact and provides a road map for evaluation.
- `host/` contains RustMonitor binary, the Linux kernel module binary, and the scripts to install and enable HyperEnclave.
- `server/` contains the source code (or patches) and scripts of all experiments to run within the enclaves. We also provide a docker container with all dependencies installed.

[5]The artifact is based on SGX SDK v2.15, Occlum LibOS v0.27, and GCC 9.4.0. The versions have little effect on the performance results.

Experiments	Figure/Table	Which container	Estimated time	Description
edge-calls	Table 1	server	10s	The latency of EENTER/EEXIT and ECALLs/OCALLs.
exception	Table 2	server	20s	Handling exceptions inside the enclaves.
NBench	Figure 8a	server	10m	Performance scores of NBench inside the enclaves.
SQLite	Figure 8b	server	15m	Throughput of in-memory SQLite database with different number of records, under YCSB A workload.
Lighttpd	Figure 8c	server/client	10m	Throughput of Lighttpd web server inside Occlum LibOS with different request sizes.
Redis	Figure 8d	server/client	20m	Latency-throughput curve of Redis in-memory database server inside Occlum LibOS with increasing request frequencies. The client uses YCSB A workload.

Table 4: Summary of the benchmarks included in the artifact.

- `client/` contains the benchmark scripts for network-based experiments (Lighttpd and Redis) to run on the client side. We also provide a docker container with all dependencies installed.
- `plots/` contains plotting scripts to generate figures from the experiment results.
- `paper-results/` contains the results shown in the paper.

Hosting

Check out https://github.com/HyperEnclave/atc22-ae (tag: `atc22-ae`, commit ID: `d1be8ab`).

Requirements

Hardware requirements:

- A 64-bit AMD platform with SVM enabled. Optionally, we recommend that the platform should support SME for the protection against physical memory attacks.
- RAM $\geq$ 16 GB.
- Free disk space $\geq$ 30 GB.

We disabled TPM and IOMMU features in RustMonitor binary for artifact evaluation to minimize the hardware requirements. These features do not affect the performance results.

Software requirements:

- Linux with the specified kernel version (i.e., `5.3.0-28-generic`) to match our given kernel module binary. We recommend Ubuntu 18.04.4 LTS which uses this version of kernel as the default.
- Docker.
- Git.
- GCC and Linux kernel headers (for building the `enable_rdfsbase` kernel module).