The Road to Trust: Building Enclaves within Confidential VMs

Wenhao Wang*§, Linke Song*§, Benshan Mei*§, Shuang Liu†, Shijun Zhao*,
Shoumeng Yan†✉, XiaoFeng Wang‡, Dan Meng*, Rui Hou*§✉

*Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS
†Ant Group
‡Indiana University Bloomington
§School of Cyber Security, University of Chinese Academy of Sciences

***Abstract*—Integrity is critical for maintaining system security, as it ensures that only genuine software is loaded onto a machine. Although confidential virtual machines (CVMs) function within isolated environments separate from the host, it is important to recognize that users still encounter challenges in maintaining control over the integrity of the code running within the trusted execution environments (TEEs). The presence of a sophisticated operating system (OS) raises the possibility of dynamically creating and executing any code, making user applications within TEEs vulnerable to interference or tampering if the guest OS is compromised. To address this issue, this paper introduces NestedSGX, a framework which leverages virtual machine privilege level (VMPL), a recent hardware feature available on AMD SEV-SNP to enable the creation of hardware enclaves within the guest VM. Similar to Intel SGX, NestedSGX considers the guest OS untrusted for loading potentially malicious code. It ensures that only trusted and measured code executed within the enclave can be remotely attested. To seamlessly protect existing applications, NestedSGX aims for compatibility with Intel SGX by simulating SGX leaf functions. We have also ported the SGX SDK and the Occlum library OS to NestedSGX, enabling the use of existing SGX toolchains and applications in the system. Performance evaluations show that context switches in NestedSGX take about 32,000 – 34,000 cycles, approximately $1.9\times$ – $2.1\times$ higher than that of Intel SGX. NestedSGX incurs minimal overhead in most real-world applications, with an average overhead below 2% for computation and memory intensive workloads and below 15.68% for I/O intensive workloads.**

I. Introduction

Confidential computing enhances cloud security by allowing tenants to control the trusted components of their workloads. It minimizes the need for trust in hardware, software, and services, while providing strong protection against attacks from other tenants and the cloud provider. This empowers tenants to develop and deploy confidential applications for their most sensitive data.

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240385
www.ndss-symposium.org

At the core of the confidential computing stack is the trusted execution environment (TEE), which isolates the code and data of a confidential workload from other code running on the system, even at the highest privilege levels. In particular, Intel SGX [5] provides protection for the trusted components of an application, known as enclaves, allowing only measured code to run within the TEEs[1]. On the other hand, VM-based TEEs, such as AMD SEV [47], Intel TDX [30], and ARM CCA [4] offer a fully backward-compatible confidential virtual machine (CVM), enabling the execution of existing applications without modifications. However, placing entire VMs within TEEs is generally considered less desirable: a VM image is far more than just a kernel and an application – it includes a large number of system services. It remains uncertain whether it is more secure than running the software on premises or on existing cloud infrastructure. Therefore, an interesting question still remains open for VM-based TEEs:

How can users attest and establish trust in applications running within a CVM, given the dynamic nature of building, loading, modification, and execution of arbitrary code within the CVM, if the guest OS might be compromised?

A seminal work called vSGX [62] was developed to virtualize Intel SGX on AMD SEV, ensuring full binary compatibility and enabling the execution of unmodified SGX programs in CVMs. vSGX adopts a two-VM approach where one VM handles the untrusted application while the other hosts the enclave. However, this emulation of SGX leaf instructions incurs significant overhead, particularly for control flow transfers across the boundary of the two VMs (e.g., EENTER and EEXIT). For instance, an empty ECall operation on vSGX takes approximately 1.5ms, which is around $160\times$ slower than on native Intel SGX. This slow context switch severely limits the service throughput achievable within the enclave. In practical scenarios, this limitation becomes a critical performance bottleneck. To illustrate, vSGX is $6\times$ slower than the baseline on the cURL benchmark, and launching a 256 MB size vSGX

[1]Intel SGXv2 [39] enables dynamic changes to the page attribute and dynamic loading of code, but with strict controls in place.

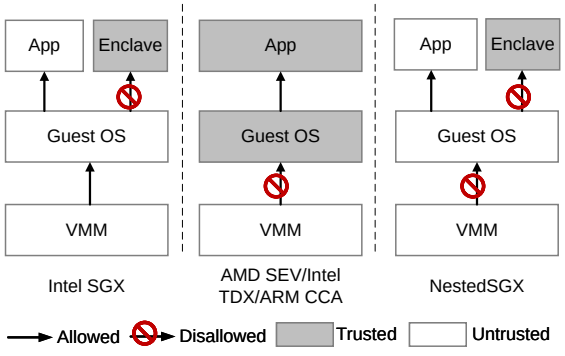


Fig. 1: Comparison between NestedSGX and other TEEs. NestedSGX offers a layered protection mechanism against both the host VMM and the guest OS within the CVM.

enclave takes about 5 minutes.

Design. We take a further step in this direction by introducing our system, called NestedSGX. NestedSGX follows a defense-in-depth approach to attesting applications (as depicted in Fig. 1), where both the enclaves and the guest OS run *within the same CVM*. Specifically, the trusted portion of an application (i.e., the enclave) is loaded and executed within an isolated environment, separated from the feature-rich OS, similar to the enclave abstraction of Intel SGX. During the loading process, the trusted portion is measured and can be attested to a remote user. The core root of trust for measurement (CRTM) and root of trust for reporting (RTR) rely on a small segment of *trusted code*, known as the security monitor. This security monitor enforces memory isolation within the CVM, and its integrity is also measured and attested with the assistance of hardware-based TEE. This approach empowers us to dynamically measure and attest the application code, enabling us to bypass the complexities associated with the guest OS. To facilitate the protection of existing applications, NestedSGX prioritizes compatibility with Intel SGX. This involves simulating SGX leaf functions with a trusted SGX emulation layer to manage the entire life cycle of enclaves, including creation, memory management, teardown, and context switches.

In more details, our approach leverages the privilege separation offered by AMD SEV-SNP, particularly the virtual machine privilege level (VMPL), to establish isolated guest physical space exclusively occupied by the security monitor and the enclaves. With VMPL, the vCPUs within the CVM are configured to operate at different privilege levels. Each privilege level is associated with specific access permissions to the guest physical memory pages. This capability enables us to assign distinct privilege levels to different software components running within the CVM. Particularly, the security monitor, the SGX emulation layer and the enclave operate at the higher VMPL (e.g., VMPL0), in the kernel mode and user mode respectively. On the other hand, the guest OS and the untrusted part of the application (App) are required to run in the kernel mode and user mode at a lower VMPL (e.g., VMPL1). This design ensures that the guest OS does not have access to the memory allocated for VMPL0.

We implemented NestedSGX on top of AMD's Linux Secure VM Service Module (SVSM) framework [16], without necessitating any modifications of the host software, such as KVM. Furthermore, we have successfully ported the Intel SGX SDK [29], the Rust SGX SDK [56] as well as the Occlum library OS [48] to the NestedSGX framework, allowing for smooth integration of existing SGX toolchains and applications. In addition, we have implemented the HotCalls [58] optimization in Occlum, which allows specific OCALLs to be handled asynchronously within the enclave without exiting it. This optimization has led to improved performance for applications running on Occlum without requiring any modifications to their source code. Unlike vSGX, we stress that *full binary compatibility with SGX is a non-goal of our paper*.

We conducted a series of benchmarking tests on commercial AMD SEV-SNP hardware and observed that the cost of world switches is about $1.9\times$ – $2.1\times$ higher than Intel SGX, which is nearly *two orders of magnitude faster than vSGX*. Moreover, NestedSGX introduces negligible overhead in most real-world applications, with less than 2% overhead for memory and computation-intensive tasks and less than 15.68% overhead for I/O-intensive tasks. Generally, NestedSGX exhibits moderate costs that are comparable to other enclave systems, such as Intel SGX. Consequently, we view it as a promising solution for addressing the critical challenge of performing secure computations within CVMs.

Contributions. The contributions are summarized as follows.

- *Design*. We present the design of NestedSGX, incorporating the VMPL feature within AMD SEV-SNP. NestedSGX addresses the challenge of establishing trust for applications while minimizing reliance on the potentially compromised guest OS.
- *Compatibility with Intel SGX*. NestedSGX offers compatibility with Intel SGX, allowing for similar management of enclaves. It significantly reduces the effort required to port existing applications onto the NestedSGX framework.
- *Implementations and evaluations*. We implemented NestedSGX on commercial hardware. The evaluation demonstrates the overhead incurred by NestedSGX is comparable to that of Intel SGX, affirming its efficiency and viability.

Roadmap. The rest of the paper is organized as follows. We provide necessary background information about Intel SGX and SEV-SNP, along with the threat model in Sec. II. We present the detailed design of NestedSGX in Sec. III, followed by implementation details in Sec. IV. The security analysis and performance evaluation of NestedSGX are provided in Sec. V and Sec. VI, respectively. We discuss possible extensions and related works of NestedSGX in Sec. VII and Sec. VIII. Sec. IX concludes the paper.

II. Background

A. Intel SGX

Intel Software Guard Extensions (SGX) is a hardware-based security technology designed to provide a trusted execution environment (TEE) called an enclave for secure processing of sensitive computations. Enclaves are isolated regions within the application's virtual address space, sharing the same page table with the untrusted part of the application. The enclave page cache (EPC) is a dedicated pool of physical memory reserved for enclaves. All enclave pages are encrypted by the hardware, safeguarding against unauthorized access even if an adversary gains physical access to the memory. To mitigate memory mapping attacks by manipulating the page table, the hardware manages the enclave page cache metadata (EPCM). This metadata stores essential information for each EPC page, such as allocation state, owner, and access permissions. During page table walk, the EPCM is consulted to ensure that only authenticated operations are permitted.

The SGX hardware enables user space applications to set aside private memory regions of code and data. The allocation of the private memory is managed through a set of privileged (ring-0) ENCLS leaf functions, while a set of unprivileged (ring-3) ENCLU leaf functions allow applications to enter and execute within these regions. Additionally, the hardware records meta-information for the enclave and its thread to effectively manage the enclave's life cycle. In the event of interrupts or exceptions, the hardware triggers an Asynchronous Enclave Exits (AEX) event. During an AEX, the enclave's context is saved and the application context is restored. Upon handling the interrupt or exception, the enclave context is restored when the execution resumes within the enclave.

SGX attestation. During enclave creation, the hardware measures the integrity of the enclave code and data. This allows for remote attestation, enabling external entities to verify the trustworthiness of the enclave. In most cases, *every component of enclave code can be measured and attested*. The root of trust for reporting resides in the attestation key, which is exclusively accessible by the Intel signed quote enclave (QE).

B. SEV-SNP and VMPL

AMD provides confidential computing capabilities through secure encrypted virtualization (SEV), a technology that builds upon virtualization. SEV utilizes a dedicated security processor – AMD Security Processor (AMD-SP) – with independent firmware, distinct from the primary x86 cores. SEV-ES, an enhancement to SEV, goes beyond encrypting guest memory and also encrypts the guest CPU registers. This ensures that in-flight values are protected from leakage and prevents malicious manipulation of registers by a compromised hypervisor.

SEV-SNP, introduced in 2020 [47], is the third generation of SEV and provides enhanced security against malicious manipulation of page mappings by the host. One of its key features is the Reverse Map Table (RMP), a structure located in secure memory that maps system physical addresses (sPAs) to guest physical addresses (gPAs). The RMP serves as a metadata table managed by the AMD-SP and plays a critical role in tracking the ownership of each system physical page to prevent the host from writing to encrypted guest pages. It establishes a global one-to-one mapping between sPAs and gPAs, ensuring that a page cannot be simultaneously mapped into multiple guests or multiple times within a single guest.

Virtual Machine Privilege Level (VMPL) is an optional feature within the SEV-SNP architecture that allows a CVM to divide its address space into four distinct levels. These levels serve as hardware-isolated abstraction layers for the CVM, with VMPL0 representing the highest privilege level and VMPL3 representing the lowest. Each hardware context, known as a Virtual Machine Save Area (VMSA), is associated with a specific VMPL. Moreover, different memory pages assigned to a guest can have varying permissions based on the VMPL. VMPLs are utilized for additional page permission checks and are independent of other x86 security features.

The RMP maintains records of the read, write, and execution permissions for each guest physical page across different VMPLs. During the translation of a virtual address to a physical address by the CPU or IOMMU, an RMP check is typically conducted to determine the relevant permissions and ownership of each physical memory page. The guest has the ability to modify VMPL permissions using the `RMPADJUST` instruction. This instruction allows a higher privilege VMPL to adjust the permissions of a lower privilege VMPL. It grants the guest the flexibility to modify and manage the permissions of different VMPLs based on its specific requirements. In addition to SEV-SNP, AMD provides the Secure Virtual Machine Service Module (SVSM) framework [50], which is a small piece of code running at the highest privilege level (VMPL0) to provide security services (such as virtual TPM) to the rest of the guest.

Switching VMPLs. VMPL switching requests are triggered by the CVM through the `VMGEXIT` instruction and captured by the hypervisor. The hypervisor switches the execution to the targeted VMPL through the `VMRUN` instruction, with the VMSA associated with the targeted VMPL as the parameter. There are two methods for the CVM to communicate the desired VMPL to the hypervisor, either by using the MSR protocol with a value of the requested VMPL in the GHCB MSR (MSR 0xc0010130), or by setting the shared GHCB with the exit code `GHCB_NAE_RUN_VMPL` (0x80000018) and the desired VMPL as the exit information.

SEV-SNP attestation. The SEV-SNP CVM is initialized from an unencrypted initial image. This image contains the boot code necessary for the CVM but does not include any confidential information. During the launch process, the hypervisor requests the AMD-SP to install this initial set of pages in the CVM. The AMD-SP cryptographically measures the content of these pages, along with the associated metadata, to ensure an accurate measurement of the initial guest memory layout. Following that, the boot code triggers the bootstrapping of the OS image, which is also a part of the measured initial image. The attestation report, signed by the AMD-SP firmware, allows

a third party, such as the guest owner, to verify the state of the CVM running on an authentic AMD platform.

C. Threat model

NestedSGX is built on top of SEV-SNP and follows a similar threat model, which considers the host software as a potential source of harm or compromise. We place our trust in the underlying SEV-SNP hardware to protect the CVM from direct observation or tampering by the host software or other CVMs. Specifically, we do not consider the recent attacks on SEV-SNP, such as the CacheWrap attack [60], the WeSee attack [44] and the Heckler attack [45].

Since the user can attest the load-time correctness of the CVM with remote attestation, we assume that the attacker initially only controls all software and hardware external to the CVM. However, considering the substantial code base of the guest OS within the CVM, it is not considered entirely trustworthy. Therefore, the primary goal of NestedSGX is to strongly maintain the integrity and confidentiality of the enclave, effectively protecting it from potential security compromises originating from both the host software and the untrusted guest OS. These entities may collaborate in their attempts to breach the enclave's security.

Furthermore, this paper does not cover the security risks associated with unsafe code, such as memory safety vulnerabilities within the enclave [19], enclave malwares [46], [52], or attacks originating from the interface between the enclave and the application (e.g., COIN [32] or IAGO [22] attacks). However, it is important to highlight that NestedSGX ensures that the enclave cannot compromise the operations or integrity of the guest OS, even in the presence of vulnerabilities. The security monitor is responsible for enforcing memory isolation between mutually distrustful components. The SGX emulation layer serves as the core root of trust for measurement (CRTM). To establish a foundation of trust, it is assumed that the security monitor and the SGX emulation layer are trusted.

This paper does not cover side channel attacks, including traditional cache side channel attacks [35], page table based attacks [54], [57], [59] or transient execution attacks [33], [53], [55]. We consider protection against these types of attacks as orthogonal to our design. Furthermore, since the host and the guest OS are free to not schedule the enclave for execution, Denial-of-Service (DoS) attacks are also out of scope.

III. Design

A. Overview

As shown in Fig. 2, NestedSGX encompasses the following components within the CVM. The application is divided into the trusted part (***enclave***) and the untrusted part (***App***). The ***security monitor*** is responsible for enforcing security isolation within NestedSGX. The ***SGX emulation layer*** emulates SGX instructions, and serves as the root of trust (RoT) for measurement and reporting of the enclave. The ***guest OS*** offers essential features to applications, such as file systems, networking, device drivers, and language runtime.

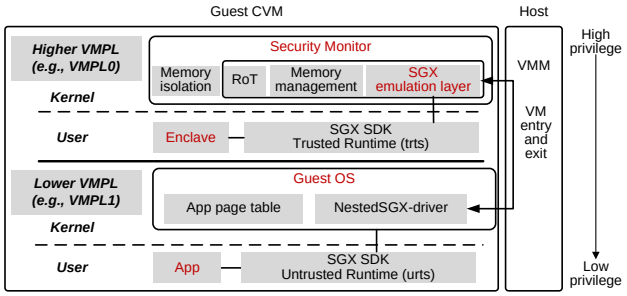


Fig. 2: NestedSGX overview. The security monitor, the SGX emulation layer and the enclave operate in the kernel and user mode at VMPL0, while the guest OS and App operate in the kernel and user mode at VMPL1. The transitions between VMPL0 and VMPL1 occur via the untrusted host.

To enforce memory isolation, the security monitor runs at the highest privileged VMPL (i.e., VMPL0). On the other hand, we cannot place the SGX emulation layer in a VMPL lower than that of the guest OS (e.g., guest OS in VMPL1 and SGX emulation layer in VMPL2). Doing so would grant the guest OS the ability to generate attestation reports on behalf of the SGX emulation layer, potentially compromising the trust chain (please refer to Sec. III-D for details). Within our threat model, we consider the SGX emulation layer to be trusted. For the sake of brevity, throughout the remainder of the paper, we regard *the SGX emulation layer as an integral component of the security monitor*, both running at VMPL0. while the guest OS and App operate at a lower privileged VMPL (i.e., VMPL1). Within this design, we also place the enclaves within the same VMPL as the SGX emulation layer, in the user mode and kernel mode respectively, since if they were placed at different VMPLs, a context switch between the enclave and App would necessitate 2 costly VMPL switches. In practice, however, if running both the SGX emulation layer and the enclaves in VMPL0 raises security concerns, we can adapt NestedSGX slightly as follows: the security monitor still runs in VMPL0; the SGX emulation layer and the enclaves run in the kernel and user mode at VMPL1, and the guest OS and the App run in the lowest VMPL (e.g., VMPL3).

To let the users maintain control over the code running inside the CVM and bootstrap trust to the enclave within a feature-rich guest OS, NestedSGX is designed to fulfill the following requirements.

Firstly, NestedSGX ensures that any transitions between the trusted and untrusted components are mediated by the security monitor. This guarantees that sensitive information is properly sanitized and safeguarded. To leverage existing SGX toolchains, NestedSGX employs the SGX programming model to manage the enclave's life cycle. The details for secure enclave state transitions will be presented in Sec. III-B.

Secondly, the system establishes isolated memory regions exclusively used by the security monitor, the enclaves and the guest OS respectively. This ensures that the enclave and

Table I: Comparison with competing approaches.

	Unlimited enclaves	Multi-threading	Exception handling	SGX ecosystem	No changes to hypervisor	Low-overhead context switches
vSGX [62]	✗	✓	✓	✓	✗	✗
Veil [13]	✓	✗	✗	✗	✗	✓
NestedSGX	✓	✓	✓	✓	✓	✓

the security monitor are secure even in scenarios where the guest OS operates in kernel mode. Specifically, the entire gPA space is divided to two parts: a continuous and fixed region exclusively used by VMPL0, and the remaining portion used by VMPL1. Additionally, our design guarantees that the enclave cannot access the memory of the guest OS, despite running at a higher VMPL. The details of the memory isolation scheme will be presented in Sec. III-C.

Lastly, in NestedSGX, only authenticated code and data can be loaded and processed within the enclave. To achieve this, NestedSGX enforces a mechanism where the enclave can only be loaded by the trusted security monitor, which also performs the measurement of the enclave's integrity. While the guest OS retains the freedom to load and execute any code within its own address space, these code segments are not affirmed by the security monitor and therefore will not be attested. The details of enclave measurement and attestation will be presented in Sec. III-D.

Comparison with competing approaches (Table I). vSGX [62] effectively virtualizes Intel SGX on AMD SEV, ensuring complete binary compatibility and the execution of unmodified SGX programs. vSGX employs two separated CVMs to accommodate the untrusted application and the enclave. In contrast, NestedSGX adopts a more intuitive design that utilizes different VMPLs for memory isolation. The NestedSGX design offers the following benefits.

Firstly, vSGX cannot support directly sharing encrypted memory between the enclave VM (EVM) and application VM (AVM). To provide confidentiality, integrity and replay protection, a dedicated communication protocol and encrypted channel are essential. Within NestedSGX, the security monitor can access the entire guest physical address space, and communicate with the guest OS through shared memory. This approach circumvents the overhead associated with memory encryption and movement, and prevents the hypervisor from observing traffic patterns. As a comparison, an empty ECALL costs 1.5 ms on vSGX, and only 12 us on NestedSGX, which is two orders of magnitude faster than vSGX.

Secondly, with vSGX, only one enclave can be hosted within the EVM, which restricts the maximum number of concurrently-running enclaves on the hardware platform. This limitation arises because SEV associates ASIDs with VMs' memory encryption keys, and the ASID bit is limited. In contrast, NestedSGX has the ability to support an unlimited number of enclaves.

Concurrent to our work, Ahmad et al. proposed Veil [13] as a service framework running at VMPL0. As one use case, Veil offers protection to the entire applications within a separated VMPL, facilitating redirection of system calls and interrupts. However, its design has some limitations: (1) Exception handling requires enclave context information and is not supported; (2) On context switches (e.g., syscall and interrupt), VMPL switches to the guest OS are triggered by the enclave, and don't follow the standard GHCB protocols, necessitating changes to the hypervisor; (3) Veil does not support SGX ecosystems. Addressing these limitations requires an SGX emulation layer to emulate SGX instructions and fulfill certain OS functionalities, such as mediating all enclave-App context switches and forwarding exceptions to the guest OS, within the user-land enclave's VMPL. Designing and implementing this layer is challenging because its state – including general-purpose and segment registers, stack pointers – changes during context switches when handling various requests (e.g., forwarding exceptions or emulating SGX instructions), so when the system re-enters the enclave again, its state cannot be fully restored due to the change of the state in the emulation layer that handles the context switch. As a result, the handling of subsequent requests can lead to failures, such as faults and stack exhaustion.

Note that this challenge cannot be resolved by vSGX or Veil. vSGX runs the App and enclave in different VMs and can simply pause or activate one of them for context switching, but also incurs significant overheads for cross-VM communications during the switch. Context switches in Veil rely on hardware, which automatically saves and restores the contexts of the enclave and the guest OS within their respective VMSAs. But this mechanism cannot handle the switches triggered by exceptions.

Our solution introduces a unique design that models the states of the SGX emulation layer using a finite state machine (FSM). After each context switch, the layer restores its original state under the FSM's guidance. We achieve this by pairing asynchronous and synchronous enclave entry and exit requests. We save the state of the SGX emulation layer before the entry request (EENTER/AEX) and restore it after processing the exit request, through a carefully designed restoration strategy based upon the FSM (e.g., restoring gs before rsp to balance the stack). In this way, the enclave state can be guaranteed to be correct no matter how complicated the request sequence could become (e.g., arbitrary exceptions between EENTER and EEXIT). This design, along with our thorough implementation of exception handling and multithreading, differentiates NestedSGX from both Veil and vSGX.

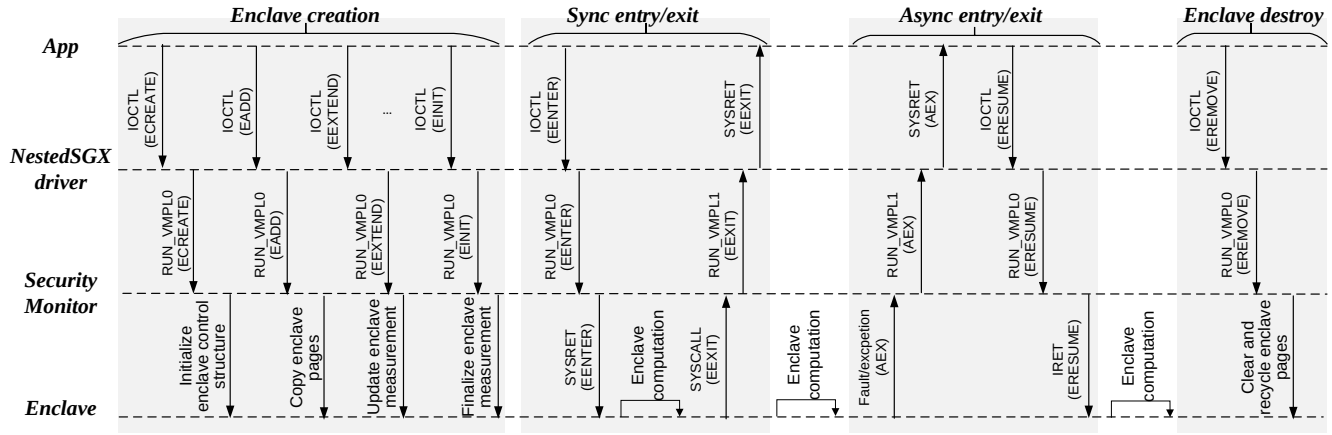


Fig. 3: The management of enclave life cycles, e.g., handling synchronous and asynchronous enclave entry and exit events.

B. Enclave life cycle management

In NestedSGX, the life cycle of an enclave closely resembles that of SGX. Initially, the application invokes the ECREATE leaf function to create the enclave, which initializes the SGX Enclave Control Structure (SECS) page. Subsequently, EPC pages necessary for the enclave, such as code sections, data sections, and Thread Control Structure (TCS) pages, are added using EADD and measured using EEXTEND leaf functions. Once all the required EPC pages are added, the enclave is initialized with EINIT, and the final measurement is performed. Upon completion of the enclave's execution, the EPC pages are reclaimed using the EREMOVE leaf function, and the enclave is destroyed.

NestedSGX achieves compatibility with SGX by transitioning to the security monitor on these leaf functions. Since VMPL switches can only be executed in the privileged mode, NestedSGX provides the NestedSGX-driver within the guest OS, which accepts the request from the App and facilitates the VMPL switch. During this process, the NestedSGX-driver receives the type of SGX leaf functions, while the parameters are saved on the App's stack. The NestedSGX-driver then switches the execution to VMPL0 and transfers control to the security monitor, where the security monitor emulates the instructions strictly adhering to the SGX specifications.

Synchronous enclave entry and exit. Once an enclave is initialized, the App can enter the enclave using the EENTER leaf function and jump to the enclave's code for execution. As shown in Fig. 3, during the emulation of EENTER, the security monitor switches to the enclave's page table and enters the enclave entry point specified by the TCS using the `sysret` instruction. The enclave can then return to the App using the EEXIT leaf function, which is also emulated by the security monitor. Specifically, the enclave returns to the security monitor using the `syscall` instruction. Subsequently, the security monitor switches the execution to the NestedSGX-driver running at VMPL1. The NestedSGX-driver restores the App's page table and returns control to the App using the `sysret` instruction.

Asynchronous enclave entry and exit. During enclave execution, faults and exceptions may occur. According to the SGX model, these faults and exceptions are supposed to be handled by the guest OS first. In NestedSGX, when a fault or exception occurs within the enclave, it traps to the fault and exception handler within the security monitor (Fig. 3). The security monitor then emulates the Asynchronous Exit (AEX) event by saving the context of the enclave in the state saving area (SSA). The security monitor switches the vCPU to the NestedSGX-driver, which operates at VMPL1. The NestedSGX-driver fills the instruction pointer (RIP) to the trampoline area (AEP) and invokes the handler of the guest OS. After the OS handles the fault or exception, it uses the `iret` instruction to return to the AEP. The AEP, in turn, resumes the enclave execution with the ERESUME leaf function. The emulation of the ERESUME leaf function is similar to that of EENTER, except that the enclave's execution context is restored from the SSA. As a result, NestedSGX is able to handle exceptions that occur during enclave execution. For example, NestedSGX supports the emulation of the CPUID instruction, enabling the execution of unmodified applications. [2]

If the untrusted NestedSGX-driver chooses not to fill RIP with AEP, this decision does not introduce any new security issues. This is because the control flow outside the enclave is already vulnerable to manipulation within the SGX model and is not inherently trusted. In NestedSGX, faults and exceptions are firstly trapped to the security monitor, enabling easy detection of suspicious behaviors such as abnormal interrupts caused by page fault-based attacks. Conceptually, NestedSGX can also support the recent AEX-notify feature [23], although the implementation is planned for future work.

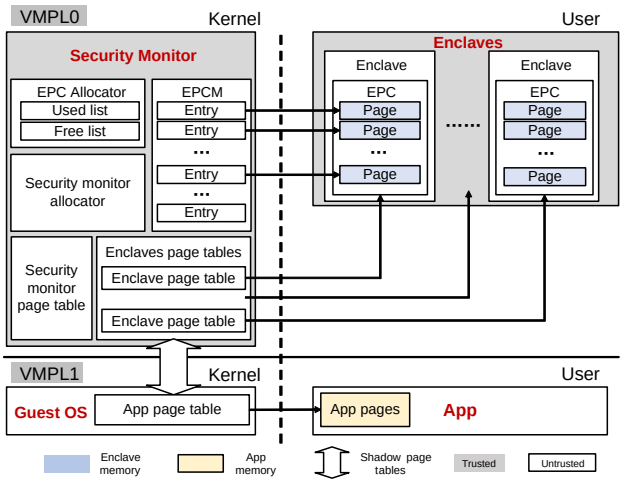


Fig. 4: The guest physical address space is divided into 3 parts: the secure memory used by the security monitor, the EPC memory, and the normal memory.

C. Memory isolation

The entire guest physical address space of the CVM is divided into 3 parts (Fig. 4): the secure memory used by the security monitor, the secure memory used by the enclaves (i.e., EPC), and the normal memory used by the guest OS and untrusted applications. The access to the secure memory is restricted solely to VMPL0, whereas the normal memory can be accessed by both VMPL0 and VMPL1. Notably, the SGX control structures are stored in the memory of the security monitor, while the enclave's code and data pages are stored in the memory reserved for the EPC.

Booting NestedSGX. Upon receiving a request to launch a CVM, the platform proceeds to load the VM image and cryptographically measures its contents. During the initialization process, the entire gPA is divided into two distinct parts. By configuring the RMP attributes, we establish a dedicated region of guest memory exclusively reserved for VMPL0, while the remaining gPA regions are reserved for VMPL1. The memory for VMPL0 is further divided into two parts, one for the security monitor and one for the enclaves.

Once the guest image is loaded, the hypervisor places the vCPU in VMPL0 mode and transfers control to the security monitor, which is positioned at the predetermined gPA location. The security monitor takes responsibility of initializing the guest CPU, memory, and setting up a page table for execution. It subsequently hands over control to the BIOS code to initiate the booting process of the guest OS. The security monitor advertises its presence and the memory range reserved for VMPL0. This prevents the guest OS from attempting to utilize any memory within that range. Any such attempts would be detected and blocked by the VMPL permission check, leading to RMP faults. In this manner, NestedSGX ensures that the guest OS cannot access or interfere with the memory of the security monitor or the enclaves.

EPC memory management. In Intel SGX, the enclave and application share the same page table, and the hardware prevents memory mapping attacks by performing additional security checks during a page table walk, using the EPCM to ensure that the memory mappings are correct. Without hardware support, it is necessary to prevent the guest OS from manipulating the mappings of the enclave. NestedSGX employs a shadow page table scheme: the application's page table is managed by the guest OS, while the enclave's page table is managed by the security monitor. In situations where the application and enclave need to share memory, such as for parameter passing, NestedSGX maps the shared memory to both page tables with the same gVA-to-gPA mapping. The mapping of the parameter buffer remains fixed throughout the entire lifetime of the enclave. Consequently, we do not need to maintain the synchronization of the two page tables.

In NestedSGX, the security monitor is responsible for managing the EPC memory. It maintains two lists: a free list and a used list, containing all the EPC pages. Similarly, an EPCM data structure is utilized to track the state of each EPC page. During enclave initialization, when an EPC page needs to be added to the enclave (via EADD), the security monitor selects a page from the free list and constructs the corresponding page table entry (PTE).

In cases when the enclave's memory accesses result in a page fault, the CPU triggers a trap to the security monitor. The security monitor forwards the page fault information, including the address of the faulting page, to the NestedSGX-driver. If the NestedSGX-driver determines that a page frame needs to be allocated for the enclave, it requests the security monitor to allocate a physical page within the EPC. Additionally, the security monitor constructs the corresponding PTE in the enclave's page table. Once the page fault is successfully handled, the security monitor restores the enclave's execution context. Currently, NestedSGX does not support swapping the enclave's pages to regular memory or disk storage.

Isolation of enclaves. It is also important to prevent an enclave from tampering with the security monitor or the guest OS. This ensures protection against potentially malicious enclave (i.e., enclave malware) or enclaves that may have vulnerabilities that could be exploited. One approach to address this concern is to place the enclave at a distinct VMPL level, separate from VMPL0 or VMPL1, and enforce the appropriate VMPL permissions to contain the enclave code. However, the transition is mediated by the security monitor, necessitating at least two VMPL switches. As a result, transitioning between the application and the enclave becomes costly.

Instead, the enclave is positioned in user mode at VMPL0 and isolated using traditional page table-based isolation. Specifically, NestedSGX ensures that the enclave's page table does not map to any gPA belonging to the security monitor or the guest OS. As a result, transition between the application

[2]In comparison, vSGX [62] requires modifying the application code in order to bypass the CPUID check.

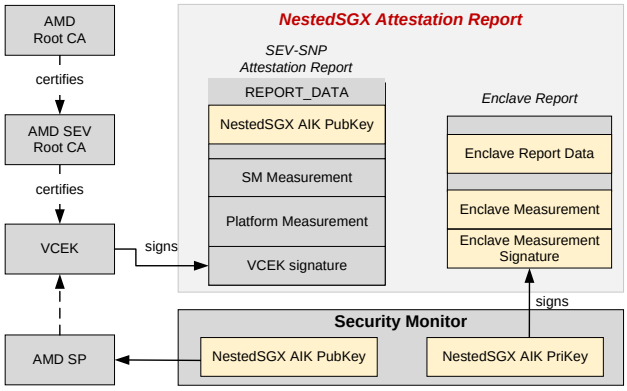


Fig. 5: The NestedSGX attestation report consists of two parts: the SEV-SNP attestation report signed by VCEK, and the enclave report signed by NestedSGX AIK. The NestedSGX AIK is generated within the security monitor and subsequently bound to the SEV-SNP hardware by placing its public key as part of the SEV-SNP attestation report.

and the enclave only requires one VMPL switch.

D. Measurement, attestation and sealing

When the CVM image is loaded and measured, the page attributes, including the access permissions of all VMPLs, are included in the measurement. Any attempts to modify the code and data in VMPL0 and VMPL1, or to manipulate the access permissions of guest pages during CVM initialization, will be reflected in the measurement. After the CVM is initialized, software operating at any VMPL level can initiate an attestation report request by sending a message to the AMD-SP firmware. The request structure comprises the VMPL level and 512 bits of space for user-provided data, which will be incorporated into the attestation report signed by the AMD hardware. The local/remote attestation and sealing mechanisms of NestedSGX closely mirror those employed by Intel SGX. This is accomplished by emulating the corresponding instructions (i.e., EGETKEY and EREPORT) within the security monitor.

Chain of trust. When the CVM is first initialized, the security monitor randomly generates a key pair, which serves as the attestation identity key (AIK) of NestedSGX. The AIK is only required to be generated once and can be reused across different CVM boot cycles. To establish the binding between NestedSGX's AIK and the CVM, we include the digest of the public part of the attestation key in the user-data field of the attestation report request (Fig. 5), while the private key never leaves the memory of the security monitor.

The security monitor initiates the generation of an *SEV-SNP attestation report* by dispatching a SNP_REPORT_REQ message to the AMD-SP hardware. The request message between the security monitor and AMD SP is encrypted using the appropriate VM Platform Communication Key (VMPCK) for VMPL0. Upon receiving the encrypted message, the AMD-SP hardware decrypts it, verifies its integrity, and subsequently responds with an attestation report, which is signed with the SEV-SNP attestation key (Versioned Chip Endorsement Key, VCEK). The report includes the platform measurement, the security monitor measurement, the VMPL level, and the public key of NestedSGX's AIK. The NestedSGX AIK is exclusively accessible to the quoting enclave (QE) and is subsequently used to sign the report of the enclave. This signed enclave report, along with the SEV-SNP attestation report, collectively constitutes the final attestation report in NestedSGX.

Upon receiving the final attestation report, the remote party verifies the certificate chain to ensure that the SEV-SNP attestation report is signed by VCEK. This process confirms that the report originates from an authentic AMD processor, has been executed within a CVM, and that the measurements of both the platform and the security monitor are accurate. Furthermore, he gains confidence that the AIK generated by NestedSGX is produced and retained within the security monitor. Finally, he verifies that the enclave report, which includes the enclave measurement, is signed with NestedSGX's AIK, concluding the attestation process.

Sealing. The security monitor can request keys from the AMD-SP by sending a MSG_KEY_REQ message to the AMD-SP. Upon receiving the request, the AMD-SP derives a key for the guest from a root key, which is responded to the security monitor and can be used as a sealing key. The sealing key can then be extended to support sealing of the enclave's secret, binding the key to the enclave's measurement. As part of the MSG_KEY_REQ message, the security monitor specifies that the derived key is used by VMPL0 and the AMD-SP mixes the VMPL information into the derived key. Although the guest OS running at VMPL1 can also request keys with the MSG_KEY_REQ message, it is not allowed to specify VMPL0 in the message, because the hardware ensures that the specified VMPL is greater than or equal to the current VMPL. Therefore, the guest OS cannot derive the same key as the one derived by the security monitor.

Table II: NestedSGX software stack: the security monitor extends the SVSM framework and is mostly written in Rust and assembly. The NestedSGX-driver and Intel SGX SDK are mostly written in C/C++ and assembly.

Component	Description	Line of Code
Security monitor	Emulation of SGX data structures, instructions and AEX	5,500
NestedSGX-driver	Handling switches between App and security monitor	800
SGX SDK & Occlum	Replacing SGX instructions with IOCTL and system calls, HotCalls	1,200
Total		**7,500**

IV. Implementations

Our implementation of NestedSGX is based on the software stack provided by AMD, which is open-sourced on GitHub [14]. The code consists of several components, including QEMU, Open Virtual Machine Firmware (OVMF), and Linux kernels for both host and guest environments. To implement NestedSGX, we extend the existing open-source Linux Secure VM Service Module (SVSM) framework [16], [50], adding around 5,500 lines of code to the SVSM implementation. For VMPL switches, NestedSGX employs the standard Model Specific Register (MSR) protocol outlined in the Guest-Host Communication Block (GHCB) specification [15]. The MSR protocol notifies the KVM to switch to the targeted VMPL, without any modification of the KVM.

Security monitor. We based our implementation on the main branch of SVSM without CPL3 support, and added our own support for running user-space enclaves. The security monitor emulates SGX leaf functions based on requests from the NestedSGX-driver. To accommodate these requests, a new type of SVSM protocol called SGX protocol was introduced within the SVSM framework [50]. The security monitor effectively enters a request loop within the SVSM framework, allowing it to continue execution upon the arrival of subsequent requests. The security monitor handles a memory pool acting as the EPC, and faithfully emulates SGX data structures and operations in strict adherence to the SGX specification. We registered customized fault and exception handlers for the enclave, emulating the AEX event. Furthermore, we configured the system call handler by setting the LSTAR MSR to handle the EEXIT leaf function.

To enter the enclave, the security monitor performs a series of actions. It switches to the enclave's page table and stack, and then executes the `sysret` instruction with RCX pointing to the entry address, effectively emulating the behavior of EENTER and ERESUME leaf functions. The EEXIT leaf function, on the other hand, is replaced with the `syscall` instruction. In this case, the system call handler within the security monitor manages the VMPL switches. In order to handle AEX events, the security monitor directly accesses the enclave's State Saving Area (SSA) by disabling Supervisor Mode Access Prevention (SMAP). This allows the security monitor to handle AEX events efficiently. Since the security monitor and the enclaves have separate page tables, we put the privilege switching code in a trampoline page that is mapped in both page tables, which is similar to the implementation of kernel page table isolation (KPTI) [26].

NestedSGX-driver. The NestedSGX-driver is responsible for managing the transitions between the App and the security monitor. When transitioning to the security monitor, it handles the SGX instruction emulation request and switches to VMPL0 using the SVSM protocol. The driver exposes the /dev/NestedSGX device to the App, which allows the App to invoke the request using the IOCTL interface, e.g., when entering the enclave with EENTER or ERESUME. In cases where the execution is returned from the enclave to the application, such as after an AEX, OCALL request, or after processing the ECALL request, the driver uses the `sysret` instruction to resume the execution of the application.

Intel SGX SDK. We made modifications to the official Intel SGX SDK to replace the SGX instructions. Our implementation retains the same parameter semantics and orders as SGX for compatibility purposes. Specifically, instructions intended for use within the enclave, such as EEXIT, EREPORT and EGETKEY, were replaced with the `syscall` instruction, which allows for invocation of the security monitor. On the other hand, instructions meant for use within the App and guest OS, such as EENTER, ECREATE and EINIT were replaced with IOCTL calls, which trigger the appropriate functionality within the NestedSGX-driver.

With the aforementioned design, the majority of SGX programs can run on NestedSGX without requiring source code modifications. All the sample code projects provided by the official Intel SGX SDK, except SampleAEXNotify, ran without any issues on NestedSGX. Furthermore, we have adapted the Rust SGX SDK [56] and the Occlum library OS [48] to the NestedSGX platform. Building upon Occlum, we implemented the HotCalls optimization [58], allowing certain OCALLs to be processed asynchronously within the enclave without exiting it.[3]

Intra-enclave isolation. Since Intel SGX only supports the monolithic model, attempts have been made to support intra-enclave isolation, such as Nested Enclave [42] and LIGHTEN-CLAVE [27]. However, these solutions necessitate hardware modifications that go beyond Intel SGX's capabilities. We incorporated support for intra-enclave isolation, leveraging the hardware capabilities of memory protection keys for userspace (PKRU). We offer four PKRU APIs as syscalls for enclaves: `pkey_set`, `pkey_alloc`, `pkey_free`, and `pkey_mprotect`. Additionally, we provide two syscalls that allow enclaves to enable or disable PKRU mechanisms.

Initially, the enclaves invoke the `pkru_enable` function, informing the security monitor to establish the key manager. Subsequently, the enclaves can use the `pkey_alloc` and `pkey_set` functions to obtain a key from the manager and make use of it through `pkey_mprotect`. On the `pkey_mprotect` function, the security monitor checks with the key manager to verify if the key has been assigned to the enclaves. If valid, the monitor updates the PKRU-bits of corresponding PTEs by traversing the enclaves' page tables. The key manager faithfully records and updates key information, including key attributes and the protected virtual address of enclaves. When the enclaves invoke the `pkru_free` function, the security monitor clears the PKRU-bits of the relevant PTEs. Lastly, the enclaves can call the `pkru_disable` function to instruct the security monitor to release the key manager.

[3]We did not cover all OCALLs but focused on optimizing 15 frequently used ones, such as `occlum_ocall_sendmsg`, `occlum_ocall_recvmsg`, `occlum_ocall_clock_gettime`, `occlum_ocall_posix_memalign`, `occlum_ocall_free`, `u_sgxprotectedfs_fread_node` and `u_setsockopt_ocall` etc.

V. Security Analysis

We conduct the security analysis by enumerating the attack vectors under the threat model outlined in Section II-C, and describing how NestedSGX defends against them.

Untrusted App, NestedSGX-driver and guest OS. The App and enclave have separated page tables, where the enclave's page table is managed by the security monitor and not accessible to the guest OS. This prevents various page table based attacks such as address mapping attacks (i.e., manipulating the enclave's address mappings) and side channels based on page fault or the access/dirty bits[4].

The NestedSGX-driver is solely tasked with the switching of the VMPL for the emulation of SGX instructions. It may attempt to modify the emulated instruction and its parameters, which however is equivalent to running a different SGX instruction. As the instruction is expected to be run by the untrusted App, this does not introduce additional attack surfaces. On the other hand, it may try to hijack the control flow after returning from the enclave, e.g., after EEXIT or AEX, however the control flow can be similarly manipulated by the malicious OS in SGX as well.

Untrusted host VMM. The VMPL switching requests are delivered to the host VMM via the MSR protocol. The host VMM is expected to switch to the targeted VMPL using the VMRUN instruction with the VMSA corresponding to the targeted VMPL as the parameter. In this process, the SEV-SNP hardware prevents it from tampering with the VMSA, which contains the state of the VM and the VMPL level. Upon request, the host VMM may try not to switch the VMPL or switch to a different VMPL, causing denial-of-service attacks, which is out of the scope of our paper.

During VMPL switches, the host VMM observes the patterns of ECALLs, OCALLs and AEXes. These leakages also exist in Intel SGX, if the App and the host OS are untrusted. Mitigating the side channels associated with enclave switches are also out of the paper's scope.

Untrusted enclave. Although the enclave operates at the highest VMPL, it is not allowed to access the gPA belonging to lower VMPLs, such as the lower VMPL's VMSA. This is achieved by page table based isolation. Only the enclave's pages are mapped in its page table which is being managed by the security monitor. The hardware also prevents the enclave from overwriting the VMPL permissions of guest pages, since the instruction to do so (RMPADJUST, Sec. II-B) is privileged and can only be executed by the security monitor.

In any case, the enclave is not allowed to bypass the security monitor. For example, it cannot directly transmit data to the host without the intervention of the security monitor. Specifically, the instruction used for the MSR protocol (i.e., `wrmsr`) is a privileged instruction and can only be executed by the security monitor. Another venue of triggering VMEXIT by the enclave is through the unprivileged `vmgexit` instruction. However, the enclave cannot explicitly pass any data to the host since the security monitor does not allow shared memory between the enclave and the host. The side channel through intentionally triggering VMEXIT is out of the paper's scope.

For the enclave to enter the security monitor, the control flow is directed to a fixed location as specified by the LSTAR MSR of VMPL0. On faults and exceptions, the control flow is directed to the fault handler as specified in the Interrupt Descriptor Table (IDT). This prevents the enclave from arbitrarily diverting the control flow within the security monitor.

The enclave may try to mount DoS attacks on the guest OS, e.g., by a busy loop that never returns, since the guest OS is not allowed to inject an interrupt into VMPL0. NestedSGX prevents the attack by preempting the enclave execution using timer interrupts and triggering AEX events. We leave the design of proactively receiving interrupt request from the guest OS as future work.

Chain of trust analysis. The NestedSGX report includes the SEV-SNP attestation report and enclave report. Any attempt to tamper with the security monitor will result in a change to the SEV-SNP attestation report, thereby ensuring the security monitor is reliable and performs as expected. Specifically, the security monitor generates NestedSGX's AIK and maintains its private key inaccessible to other components. It associates NestedSGX's AIK with the platform by including the digest of the AIK's public key in the SEV-SNP attestation report.

While the enclave operates at VMPL0, it is prohibited from directly obtaining the SEV-SNP attestation report. Otherwise, the enclave may incorporate a counterfeit AIK into the report, which could compromise the integrity of the trust chain, particularly if the enclave deliberately exposes the private key of the AIK. This protection is enforced by the fact that SNP_REPORT_REQ can only be initiated from the kernel mode, which is the security monitor. More specifically, SNP_REPORT_REQ necessitates shared memory with the host VMM for passing request and response messages, but we intentionally disable shared memory between the enclave and the host VMM. The hardware also prevents the guest OS (operating at VMPL1) from obtaining an SEV-SNP attestation report for VMPL0. Specifically, the desired VMPL is provided by the guest in the `SNP_REPORT_REQ` message, and the guest can only generate attestation reports for VMPLs that are *greater than or equal to the current VMPL* [17, Sec. 7.3].

The security monitor employs the SNP_GUEST_REQUEST command to solicit the SEV-SNP attestation report. This command establishes a trusted channel between the security monitor and the AMD-SP firmware, encrypted with AES-GCM authenticated encryption using the VMPCK associated with VMPL0. Each message contains a sequence number. Although the VMM mediates the communication between the CVM and the firmware, it is unable to modify or drop these messages without being detected, nor can it access the plaintext of the messages.

[4]Side channel attacks based on nested page tables (NPTs) are still possible, since the NPTs are managed by the untrusted hypervisor.

Table III: Latencies of emulated SGX leaf instructions. The emulation of ENCLS leaf instructions require VMPL switches, which dominates the cost, while most ENCLU leaf instructions are emulated entirely at VMPL0. The slower performance of ENCLU leaf instructions on NestedSGX is attributed to the use of slower Rust implementations in the cryptographic crates.

		vSGX	NestedSGX
ENCLS	ECREATE	3,719 us	8.4 us
	EADD	1,421 us	8.0 us
	EEXTEND	987 us	33 us
	EINIT	811 us	46 us
	EREMOVE	1,014 us	7.8 us
	EAUG	990 us	7.9 us
	EBLOCK	841 us	9.2 us
	ELDB/ELDU	1,958 us	9.3 us
	EMODPR	1,071 us	9.9 us
	EWB	1,819 us	7.9 us
ENCLU	EGETKEY	5.0 us	17 us
	EREPORT	19 us	30 us

VI. Evaluations

A. Experimental setup

We deployed NestedSGX on a server with two AMD EPYC 7543 CPUs (two threads per core, total of 128 logical cores) with 64 GB DDR4 RAM. Every CVM was allocated 4 vCPUs and 4 GB RAM, and ran Ubuntu 22.04 with kernel version 6.5.0-snp-guest (svsm-preview-guest-v3 branch provided by AMD). We configured 512 MB gPA for the security monitor and the enclaves. The modified Linux SGX SDK was based on version 2.20, and the Occlum version was version 0.29.7. We used the main branch of the linux-svsm framework (commitID: 8d518f1). The SVSM framework and the security monitor were compiled with Rust in the default mode (-O0). All programs were compiled with GCC 11.4.0 and the same optimization level (-O2).

The baseline was conducted on the same AMD server running in the SGX SDK simulation mode (sim mode), which does not provide any security guarantees. Specifically, we used the same code base under the same optimization level and compiling options. Notably, as a result of the HotCalls optimization, applications running on Occlum experienced improved performance without the need for any modifications to their source code. We were not able to reproduce the experiment reported in vSGX, and the vSGX results are taken directly from the published paper [62]. The Intel platform for comparison was equipped with an Intel Xeon Platinum 8369B CPU and 64 GB RAM.

B. Micro-benchmarks

SGX leaf instructions. We measured the emulated SGX leaf instructions in NestedSGX, averaging the results over 10,000 runs. As shown in Table III, all ENCLS leaf instructions on NestedSGX are significantly faster than vSGX. This is attributed to the requirement of VMPL switches for emulating ENCLS leaf instructions on NestedSGX, whereas vSGX necessitates more costly cross-VM communications.

Table IV: Latencies of context switches. The cost of NestedSGX includes the following parts: a VMPL switch round-trip which costs about 19,400 cycles on our platform, the SGX SDK routine which costs 4000–5500 cycles, and the rest is the cost of our unoptimized implementation of the security monitor in emulating SGX instructions (including an EENTER and an EEXIT, which cost about 9,000 cycles).

	Intel SGX	vSGX	NestedSGX
ECALL	10,988 cycles (4.1 us)	$\approx 1,500$ us	33,584 cycles (12 us)
OCALL	9,337 cycles (3.5 us)	-	32,014 cycles (11 us)

Both EGETKEY and EREPORT on NestedSGX are slower than those on vSGX, which we confirmed is due to the use of the Rust `rust-crypto` crate that is slower than the implementation used by vSGX. Notably, EGETKEY and EREPORT are not heavily invoked in practice.

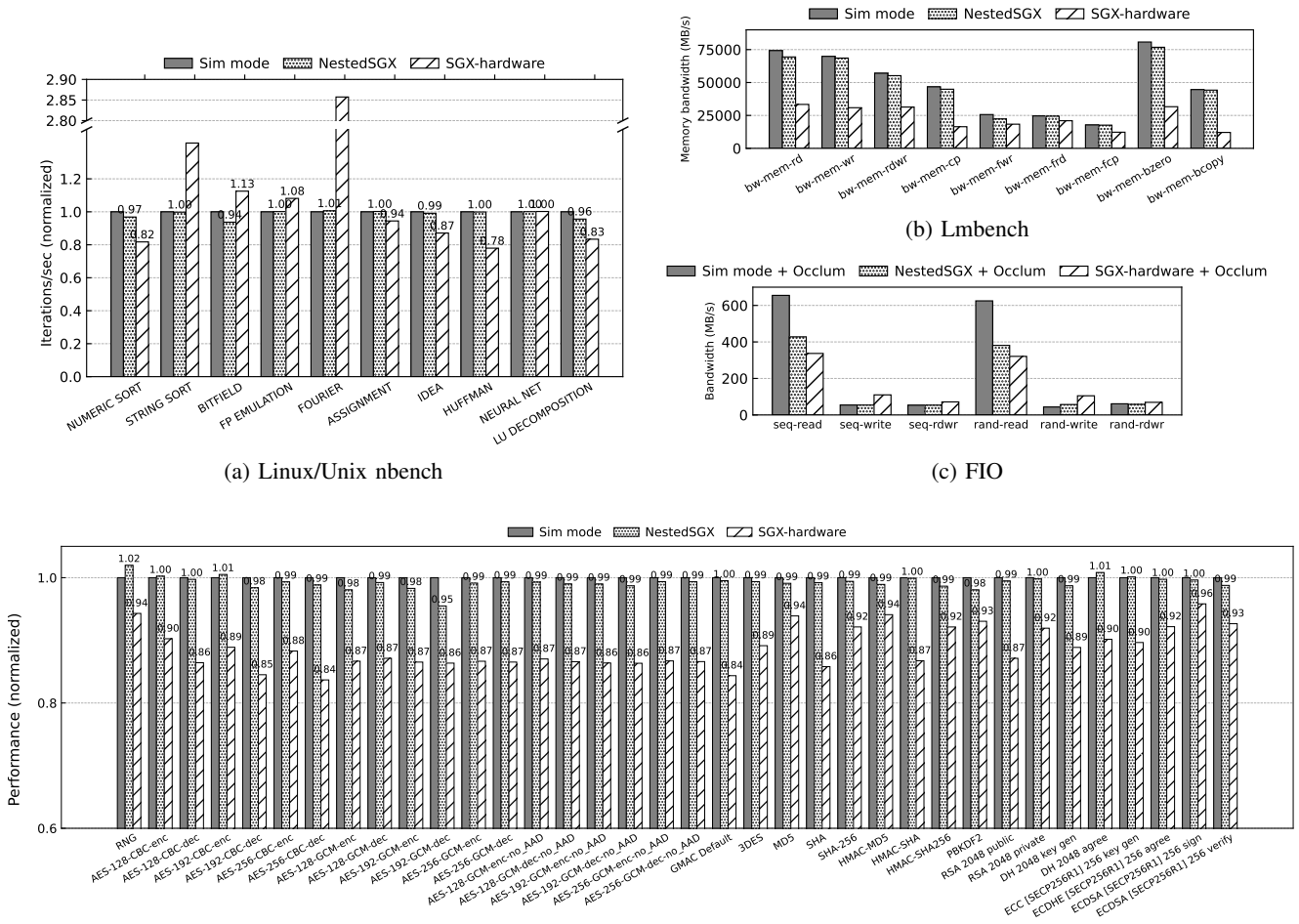
Context switches. We measured the latency of ECALLs and OCALLs in NestedSGX. The test runs empty edge calls with no explicit parameters 100,000 times and takes the average value. As shown in Table IV, the latency of context switches in NestedSGX is $1.9\times - 2.1\times$ higher than Intel SGX, but is still two orders of magnitude faster than vSGX. Notably, existing works on Intel SGX, including EleOS [41], HotCalls [58] and Switchless calls [51] can be further applied to NestedSGX to reduce the cost brought by context switches.

Linux/Unix nbench. Linux/Unix nbench is a benchmark suite that focuses on evaluating the performance of a computer system's CPU, FPU, and memory system [38]. We utilized an adapted version of nbench, namely SGX-NBench [1] to evaluate the computation performance of NestedSGX. Fig. 6a demonstrates that NestedSGX introduces an overhead of approximately 1.3% on average over the baseline, which we believe mostly attributes to the context switches introduced by SGX-NBench during the evaluation.

Lmbench. Lmbench is a benchmark suite commonly employed to assess various aspects of system behavior. In our evaluation, we utilized the SGX version of Lmbench, known as SGX-bench [28], specifically employing the `bw_mem` test to evaluate the memory bandwidth. We conducted 10,000 repetitions of the test and recorded the average values. The memory block size for every access was set to 2 MB. The results, presented in Fig. 6b, indicate that the memory bandwidth achieved is 98.7% of the baseline.

WolfSSL [11]. We performed an evaluation of WolfSSL with Intel SGX [12]. This benchmark primarily focuses on computation-intensive tasks, such as encryption, decryption, digests, and signature verification. As illustrated in Fig. 6d, NestedSGX introduces an average overhead of about 0.78% compared to the baseline.

Flexible I/O Tester (FIO) [10] with Occlum. We conducted an evaluation of the I/O performance of NestedSGX by utilizing FIO (v3.28) with Occlum in the default configuration. Our focus was on benchmarking the bandwidth of randomized



(a) Linux/Unix nbench

(b) Lmbench

(c) FIO

(d) Wolfssl

Fig. 6: Micro-benchmarks.

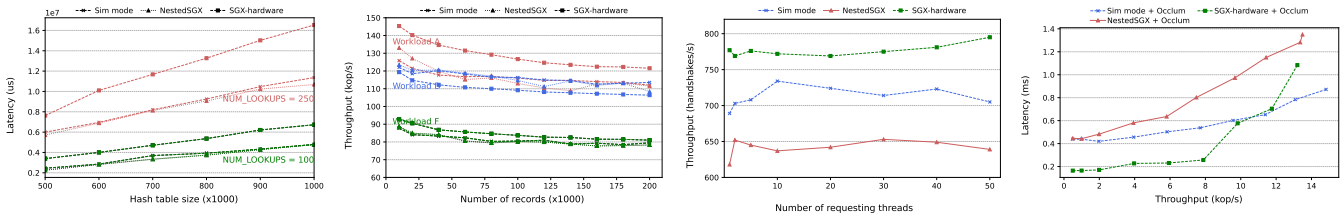
and sequential read and write operations to the disk, utilizing a single thread. We chose the direct I/O mode for sequential operations. Each test involved accessing a 256 MB file with a block size of 256 KB per access. We repeated the test 10 times, with each iteration lasting 100 seconds, and calculated the average bandwidth from all the runs. As depicted in Fig. 6c, the I/O read performance of NestedSGX is approximately 65.3% of the baseline, while the I/O write performance is similar to the baseline, possibly due to Occlum's optimizations to reduce the context switches in write operations.

C. Real-world evaluations

Hash join. Hash join is used to implement the "equivalent-join" operation in modern databases. It involves creating a hash table from rows of the first table and then probes it with rows of the second table. We used the open-source implementation of the algorithm in SGXGauge [8], [34]. We varied the size of the first table (from 500K to 1M records) and fixed the second table (100K), effectively varying the memory and computation intensive nature of the workload. As shown in Fig. 7a, the overhead of NestedSGX is negligible.

SQLite. We utilized the publicly available version of SQLite (v3.19.3) operating on Intel SGX [3]. To evaluate the memory performance of NestedSGX, we configured the database to function in-memory mode and incorporated the client within the enclave. We conducted performance evaluations across various record sizes, measuring the time required for 100,000 database operations. Specifically, we employed three representative workloads within the Yahoo! Cloud Serving Benchmark (YCSB) suite [2]: workload A (update heavy with 50% reads and 50% updates), D (read latest, i.e., delete old ones, insert new ones and read mostly the new ones), and F (short range scan) for the evaluation. As shown in Fig. 7b, NestedSGX introduces about 0.77% overhead on average over the baseline.

TLS server. We utilized the open-source SGX-OpenSSL project and a sample implementation of a TLS server in an enclave [6] for our evaluation. The TLS client operated within the native CVM and sent requests to the server over the local loop-back. To measure the average latency, we employed the tls-perf benchmarking tool [9] and set the number of requesting threads to 1, 2, 5, 10, 20, 30, 40, 50 respectively. As illustrated in Fig. 7c, NestedSGX incurs approximately a



(a) Hash join (b) SQLite (c) TLS server (d) Redis with Occlum

Fig. 7: Real world application benchmarks.

9.75% overhead over the baseline.

Redis. We ran a Redis (v6.0.9) database server with Occlum on NestedSGX. The database was configured as an in-memory setup, and we applied the YCSB workload A for our evaluation. We first loaded 5,000 records (in total 5 MB data), and subsequently executed 10,000 operations from 20 clients over the local loopback. We gradually increased the request frequency and measured the corresponding latency at different throughput levels. As shown in Fig. 7d, NestedSGX achieved a throughput of 84.32% (15.68% overhead) compared to the baseline. The latency overhead was found to be 6.4% under low load conditions and increased to approximately 55% as the system approached its maximum throughput.

VII. Limitations & Future Works

Limitations. Since our implementation is still in the prototype stage, NestedSGX has some limitations to fully support the SGX model. Firstly, the security monitor manages the enclave's page tables, which prevents the guest OS from manipulating the enclave's page tables for potential address mapping attacks. Since the App and enclave have separate page tables, the enclave cannot access the App's memory, as their memory views may differ if the App's page table is updated without synchronizing with the enclave's page table. Therefore, features such as `user_check` and memory sharing are not currently supported by NestedSGX. Secondly, the NestedSGX-driver does not currently support swapping the enclave's memory to disks. Thirdly, the guest OS lacks the capability to inject an interrupt into VMPL0. This limitation imposes constraints on the OS's scheduling of enclave threads. Finally, compatibility is currently provided on top of SGX SDK and Occlum. Thus, applications that are purely written in the SGX instruction set without any SDK or LibOS are not yet supported. We do not think that any of these limitations are inherent to NestedSGX, e.g., memory sharing could be supported if updates to the App's page table were synchronized with the enclave's page table via VMPL switches.

Supporting other VM-based TEEs. Recently, ARM CCA introduces the support of different planes within a CVM, in which each plane is essentially a separated VM but they all share the same guest physical address space. Plane 0 is more privileged and can run a paravisor which manages the switches between other planes and can restrict the memory accessible by other planes [18]. According to ARM's roadmap, the paravisor is designed to support secure services such as vTPM emulation. Similarly, we can re-purpose the paravisor to support running userspace enclaves. TDX supports a similar feature named TD partitioning. Therefore, it is promising that NestedSGX can be extended to other CVM platforms. However, it is worth mentioning that the TDX module and CCA's Realm Management Monitor (RMM) are critical for the memory isolation and secure management of CVMs, and the customization of these components may not be allowed by chip companies. In fact, only the TDX module signed by Intel can run in the Secure Arbitration Mode (SEAM).

Supporting other TEE abstractions. NestedSGX adopts the SGX model as it is arguably the most prevalent TEE with process abstraction. The NestedSGX platform can be enhanced to accommodate other TEE abstractions, such as ARM TrustZone. Specifically, VMPL0 could run the trusted OS (e.g., OP-TEE) and trusted applications (TAs), providing secure services to the guest OS running at VMPL1, which is analogous to the normal world. The SMC instruction is used to handle transitions between the secure world and normal world. In TrustZone, the SMC handler is within the trusted firmware. In our system, the SMC instruction needs to be replaced with a VMPL switching request, which transfers the execution to the trusted OS. This allows porting existing applications built on ARM TrustZone to SEV-SNP CVMs.

VIII. Related Works

Integrity Measurement Architecture (IMA). Integrity plays a vital role in ensuring system security by guaranteeing the exclusive loading of authentic software onto a machine. Measured boot, utilizing the Trusted Platform Module (TPM), securely captures measurements of all boot software, culminating in the loading and execution of the OS kernel. The Linux Integrity Measurement Architecture (IMA) expands upon the concept of measured boot by comprehensively recording all software executions and file accesses within the OS, securely storing them in the TPM [43]. The Container-IMA (C-IMA) extends IMA to containers, enabling the measurement of container images and the runtime integrity measurement of container processes [36].

In lieu of TPM, TZ-IMA extends the storage of measurements in TrustZone [49]. Recently, Intel introduced a solution of IMA in confidential cloud environments, with a focus

on Intel TDX, enabling IMA of programs and applications running in Intel trusted domain (TD) during runtime [7].

However, IMA in general assumes the trust of the boot components, which includes the BIOS, bootloader and guest OS, leading to potentially large TCB and expanded attack surface. For example, it has been shown that IMA can be bypassed with a malicious block device [20]. In contrast, the core root of trust for measurement of NestedSGX (i.e., the security monitor) operates at VMPL0 and we do not place trust on the guest OS. Moreover, the security monitor not only measures the integrity of both enclave code and data but also ensures that the enclave runs in an isolated environment from the guest OS, which is beyond the design goal of IMA.

Establishing the enclave abstraction within TEEs. With the emergence of Intel SGX, endeavors have been undertaken to enhance the interoperability between Intel SGX and other TEE platforms. Notably, Komodo [24], Sanctuary [21], and SecTEE [61] offer support for enclave abstractions on ARM TrustZone. HyperEnclave [31] re-introduces Intel SGX on standard AMD hardware through the use of a small and trusted hypervisor. In contrast to these works, NestedSGX sets itself apart by adopting a distinctive model. While the guest OS remains shielded from the untrusted hypervisor, it is not entirely trusted. Consequently, NestedSGX enables enhanced protection for enclaves within the feature-rich guest OS. Most related to our work are vSGX [62] and Veil [13]. A more detailed comparison with them is provided in Sec. III-A.

Security and performance enhancement using the VMPL feature. Hecate [25] utilizes VMPL0 as an L1 hypervisor, operating within the CVM on top of the untrusted L0-hypervisor (host VMM). Hecate enables the migration of on-premises workloads by maintaining compatibility with unmodified VM images and implementing security policies, such as network firewalls. Honeycomb [37] executes a validator within VMPL0, safeguarded from other software. The validator analyzes the binary code of a GPU kernel to validate that each memory instruction within the GPU kernel can only access specific memory regions, employing static analysis techniques. SVSM-vTPM [40] employs VMPL to isolate the virtual TPM (vTPM) from the guest OS, ensuring the integrity of vTPM's functionalities and the security of the root key for remote attestation. However, all these solutions do not provide isolation for user programs from the untrusted guest OS.

IX. Conclusions

Although confidential computing aims to protect the confidentiality and integrity of code and data within TEEs, it remains challenging to control code within CVMs since the feature-rich guest OS is free to load any code. This paper presents NestedSGX, a secure and efficient solution for maintaining control over the integrity of code and data in TEEs. NestedSGX creates hardware enclaves inside CVMs with VMPL, which ensures remote attestation of trusted and measured code. NestedSGX is compatible with Intel SGX and easily integrated into existing systems. NestedSGX incurs a small overhead in most real-world applications.

Acknowledgments

We would like to express our sincere gratitude to the anonymous reviewers and our shepherd for their insightful and valuable feedback. The authors from Institute of Information Engineering were supported in part by the National Key R&D Program of China (Grant No. 2020YFB1805402), the National Natural Science Foundation of China (Grant No. 62272452), the Strategic Priority Research Program of the Chinese Academy of Sciences (Grant No. XDB0690100) and the research grant from Ant Group. Corresponding authors: Shoumeng Yan (shoumeng.ysm@antgroup.com) and Rui Hou (hourui@iie.ac.cn).

References

[1] "The nbench benchmark ported to SGX," https://github.com/utds3lab/sgx-nbench, 2017.
[2] "Core Workloads (YCSB)," https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads, 2021.
[3] "SQLite," https://www.sqlite.org, 2021.
[4] "Arm confidential compute architecture," https://developer.arm.com/documentation/den0125/0200/?lang=en. Referenced December 2022, 2022.
[5] "Intel software guard extensions overview," https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html. Referenced December 2022, 2022.
[6] "OpenSSL library for SGX application," https://github.com/sparkly9399/SGX-OpenSSL, 2023.
[7] "Runtime integrity measurement and attestation in a trust domain," https://www.intel.com/content/www/us/en/developer/articles/community/runtime-integrity-measure-and-attest-trust-domain.html, 2023.
[8] "SGXGauga comprehensive benchmark suite for Intel SGX," https://github.com/sandeep007734/SGXGauge-Benchmark, 2023.
[9] "TLS handshakes benchmarking tool," https://github.com/tempesta-tech/tls-perf, 2023.
[10] "Flexible I/O Tester," https://github.com/axboe/fio, 2024.
[11] "WolfSSL and wolfCrypt Benchmarks — Embedded SSL/TLS library," https://github.com/wolfSSL/wolfssl, 2024.
[12] "wolfSSL with Intel SGX," https://www.wolfssl.com/wolfssl-with-intel-sgx/, 2024.
[13] A. Ahmad, B. Ou, C. Liu, X. Zhang, and P. Fonseca, "Veil: A protected services framework for confidential virtual machines," in *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (to appear)*, 2024.
[14] AMD, "AMD ESE," https://github.com/AMDESE/, 2023.
[15] ——, "Guest Hypervisor Communication Block Standardization," https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/56421.pdf, 2023.
[16] ——, "Linux SVSM (Secure VM Service Module) for secure x86 virtualization in Rust," https://github.com/AMDESE/linux-svsm, 2023.
[17] ——, "Sev secure nested paging firmware abi specification," https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/56860.pdf, 2023, publication # 56860.
[18] Arm, "Evolution of the arm confidential compute architecture," https://www.youtube.com/watch?v=1AsvIt7bSLY, 2024.
[19] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi, "The guard's dilemma: Efficient Code-Reuse attacks against intel SGX," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1213–1227.
[20] F. Bohling, T. Mueller, M. Eckel, and J. Lindemann, "Subverting linux'integrity measurement architecture," in *Proceedings of the 15th International Conference on Availability, Reliability and Security*, 2020, pp. 1–10.
[21] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "SANCTUARY: ARMing TrustZone with user-space enclaves," in *NDSS*, 2019.
[22] S. Checkoway and H. Shacham, "Iago attacks: Why the system call API is a bad untrusted RPC interface," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 253–264, 2013.

[23] S. Constable, J. Van Bulck, X. Cheng, Y. Xiao, C. Xing, I. Alexandrovich, T. Kim, F. Piessens, M. Vij, and M. Silberstein, "AEX-Notify: Thwarting precise Single-Stepping attacks through interrupt awareness for intel SGX enclaves," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4051–4068.

[24] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 287–305.

[25] X. Ge, H.-C. Kuo, and W. Cui, "Hecate: Lifting and shifting on-premises workloads to an untrusted cloud," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1231–1242.

[26] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "Kaslr is dead: long live kaslr," in *Engineering Secure Software and Systems: 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings 9*. Springer, 2017, pp. 161–176.

[27] J. Gu, B. Zhu, M. Li, W. Li, Y. Xia, and H. Chen, "A Hardware-Software co-design for efficient Intra-Enclave isolation," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3129–3145.

[28] A. Hasan, R. Riley, and D. Ponomarev, "Port or shim? stress testing application performance on intel sgx," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2020, pp. 123–133.

[29] Intel, "Intel SGX SDK," https://github.com/intel/linux-sgx/, 2020.

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

[31] Y. Jia, S. Liu, W. Wang, Y. Chen, Z. Zhai, S. Yan, and Z. He, "HyperEnclave: An open and cross-platform trusted execution environment," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 437–454.

[32] M. R. Khandaker, Y. Cheng, Z. Wang, and T. 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*, 2020, pp. 971–985.

[33] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," *Communications of the ACM*, vol. 63, no. 7, pp. 93–101, 2020.

[34] S. Kumar, A. Panda, and S. R. Sarangi, "A comprehensive benchmark suite for Intel SGX," *arXiv preprint arXiv:2205.06415*, 2022.

[35] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE symposium on security and privacy*. IEEE, 2015, pp. 605–622.

[36] W. Luo, Q. Shen, Y. Xia, and Z. Wu, "Container-IMA: A privacy-preserving integrity measurement architecture for containers," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, 2019, pp. 487–500.

[37] H. Mai, J. Zhao, H. Zheng, Y. Zhao, Z. Liu, M. Gao, C. Wang, H. Cui, X. Feng, and C. Kozyrakis, "Honeycomb: Secure and efficient GPU executions via static validation," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 155–172.

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

[39] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "Intel software guard extensions (Intel SGX) support for dynamic memory management inside an enclave," in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, 2016, pp. 1–9.

[40] V. Narayanan, C. Carvalho, A. Ruocco, G. Almási, J. Bottomley, M. Ye, T. Feldman-Fitzthum, D. Buono, H. Franke, and A. Burtsev, "Remote attestation of SEV-SNP confidential VMs using e-vTPMs," in *Annual Computer Security Applications Conference (ACSAC)*, 2023.

[41] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: Exitless OS services for SGX enclaves," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 238–253.

[42] J. Park, N. Kang, T. Kim, Y. Kwon, and J. Huh, "Nested enclave: Supporting fine-grained hierarchical isolation with sgx," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 776–789.

[43] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn, "Design and implementation of a tcg-based integrity measurement architecture," in *USENIX Security symposium*, vol. 13, no. 2004, 2004, pp. 223–238.

[44] B. Schlüter, S. Sridhara, A. Bertschi, and S. Shinde, "Wesee: Using malicious# vc interrupts to break amd sev-snp," *arXiv preprint arXiv:2404.03526*, 2024.

[45] B. Schlüter, S. Sridhara, M. Kuhne, A. Bertschi, and S. Shinde, "Heckler: Breaking confidential vms with malicious interrupts," *arXiv preprint arXiv:2404.03387*, 2024.

[46] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using sgx to conceal cache attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 3–24.

[47] A. SEV-SNP, "Strengthening vm isolation with integrity protection and more," *White Paper, January*, 2020.

[48] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. 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*, 2020, pp. 955–970.

[49] L. Song, Y. Ding, P. Dong, Y. Guo, and C. Wang, "Tz-ima: Supporting integrity measurement for applications with arm trustzone," in *International Conference on Information and Communications Security*. Springer, 2022, pp. 342–358.

[50] A. SVSM, "Secure vm service module for sev-snp guests," 2022.

[51] H. Tian, Q. Zhang, S. Yan, A. Rudnitsky, L. Shacham, R. Yariv, and N. Milshten, "Switchless calls made practical in Intel SGX," in *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, 2018, pp. 22–27.

[52] F. Toffalini, M. Graziano, M. Conti, and J. Zhou, "SnakeGX: a sneaky attack against SGX enclaves," in *International Conference on Applied Cryptography and Network Security*. Springer, 2021, pp. 333–362.

[53] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient Out-of-Order execution," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 991–1008.

[54] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page Table-Based attacks on enclaved execution," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1041–1056.

[55] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 88–105.

[56] H. Wang, P. Wang, Y. Ding, M. Sun, Y. Jing, R. Duan, L. Li, Y. Zhang, T. Wei, and Z. Lin, "Towards memory safe enclave programming with rust-sgx," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2333–2350.

[57] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. 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*, 2017, pp. 2421–2434.

[58] O. Weisse, V. Bertacco, and T. 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)*. IEEE, 2017, pp. 81–93.

[59] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 640–656.

[60] R. Zhang, L. Gerlach, D. Weber, L. Hetterich, Y. Lü, A. Kogler, and M. Schwarz, "CacheWarp: Software-based Fault Injection using Selective State Reset," in *USENIX Security*, 2024.

[61] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng, "SecTEE: A software-based approach to secure enclave architecture using TEE," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1723–1740.

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

ARTIFACT APPENDIX

A. Description & Requirements

NestedSGX can support existing SGX toolchains (the adapted SGX SDK and Occlum library OS) and run SGX applications atop AMD SEV-SNP confidential VMs (CVMs). This artifact contains the binaries of the security monitor, and documentations on how to setup the NestedSGX environment.

The artifact includes benchmarks for edge calls (i.e., ECALLs and OCALLs), and benchmarks for the real-world workloads, including NBench, SQLite and Redis etc. We offer scripts to replicate the results outlined in the paper, as summarized in Table V.

1) How to access: Check out https://github.com/NestedSGX/nestedsgx-ndss25-ae/. The DOI link of our project is given below.

- Normal version: 10.5281/zenodo.14241711.
- Hotcall version: 10.5281/zenodo.14226886.

2) Hardware dependencies: :

- 3rd Gen AMD EPYC processors with SEV-SNP enabled in the BIOS.
- RAM $\geq$ 16 GB.
- Free disk space $\geq$ 200 GB.

3) Software dependencies:

- Linux with the specified kernel version (i.e., `6.2.0-26-generic`) to build the NestedSGX environment. We recommend Ubuntu 22.04 which uses this version of kernel as the default.
- Git.
- GCC 11.4.0.
- Rust tool chain nightly 1.71.

4) Benchmarks:

- NBench [1].
- Lmbench [1].
- FIO [10].
- Wolfssl [12].
- SGXGauge [8].
- SQLite [3].
- SGX-OpenSSL [6].
- Redis.
- Occlum [48].

B. Artifact Installation & Configuration

The Major steps to set up the environment and reproduce the results are listed below. For more detailed instructions, please refer to the guidelines outlined in the `README.md` file.

- ***Configure the environment.*** Begin by setting up the environment following AMD's Linux SVSM project [16]. This process involves building specific kernels for both the host OS and the guest OS, along with the corresponding QEMU and OVMF files..
- ***Replace the security monitor.*** Substitute the original `svsm.bin` with our customized version to serve as the security monitor for initiating NestedSGX.
- ***Install the guest module.*** Install our guest module on the guest OS. This kernel module facilitates redirecting control flow within NestedSGX.
- ***Prepare the Linux SGX SDK environment.*** Set up the Linux SGX SDK environment to enable simulation mode in the guest VM. Once completed, you can run micro-benchmarks like NBench, Lmbench, Wolfssl, as well as macro-benchmarks including hash join, SQLite, and TLS server.
- ***Set up the Occlum library OS for FIO and Redis.*** Compile and install the modified Occlum to support FIO and Redis. Enhance the performance of these benchmarks by utilizing the HotCalls versions of both the Intel SGX SDK and Occlum.

C. Major Claims

- (C1): The latencies of emulated SGX leaf instructions and edge-calls in NestedSGX are reported in Table III and Table IV, which are about 2$\times$ higher than Intel SGX, but two orders of magnitude faster than vSGX. These are proven by the experiments (E1) and (E2) in Table V.
- (C2): For micro-benchmarks, NestedSGX incurs a low overhead ($< 2\%$) for computation and memory intensive tasks (i.e., Nbench and Lmbench), and a high overhead (up to 34.7%) for I/O intensive tasks (i.e., FIO). These are reported in Fig. 6 and are proven by the experiments (E3), (E4), (E5) and (E6).
- (C3): For real-world evaluations, NestedSGX incurs a low overhead ($< 1\%$) for computation and memory intensive tasks (i.e., Hash join and SQLite), and a high overhead (up to 15.68%) for I/O intensive tasks (i.e., TLS server and Redis). These are reported in Fig. 7 and are proven by the experiments (E7), (E8), (E9) and (E10).

D. Evaluation

We provide a list of all the experiments included in the artifact in Table V. For detailed steps to replicate our results, please refer to the scripts in the `benchmarks` directory. We also provide instructions to collect the results and the scripts to plot the figures (please see `benchmarks/README.md`).

Table V: Summary of the benchmarks included in the artifact. We do not currently offer scripts to plot the results of FIO, Redis, and the TLS server as these benchmarks necessitate some manual effort during the evaluation. The readers can still use our provided scripts to plot the results of FIO and Redis on the version of NestedSGX that does not support HotCalls.

No.	Experiments	Figure/Table	Estimated time	Description
(E1)	Leaf instructions	Table III	1m	The latency of emulated SGX leaf instructions.
(E2)	edge-calls	Table IV	10s	The latency of ECALLs/OCALLs.
(E3)	NBench	Figure 6a	10m	Performance scores of NBench inside the enclaves.
(E4)	Lmbench	Figure 6b	10m	Lmbench memory bandwidth inside the enclaves.
(E5)	FIO	Figure 6c	10m	Bandwidth of FIO read and write operations inside the enclaves.
(E6)	Wolfssl	Figure 6d	10m	Throughput of Wolfssl algorithms inside the enclaves.
(E7)	Hash join	Figure 7a	15m	Latency of hash join inside the enclaves.
(E8)	SQLite	Figure 7b	15m	Throughput of in-memory SQLite database with different number of records, under YCSB A workload.
(E9)	TLS server	Figure 7c	15m	Throughput of TLS server inside the enclaves.
(E10)	Redis	Figure 7d	20m	Latency-throughput curve of Redis in-memory database server inside the Occlum LibOS with increasing request frequencies.