vSGX: Virtualizing SGX Enclaves on AMD SEV

Shixuan Zhao*, Mengyuan Li*, Yinqian Zhang†‡✉, Zhiqiang Lin*✉

*Department of Computer Science and Engineering, The Ohio State University
†Research Institute of Trust-worthy Autonomous Systems, Southern University of Science and Technology
‡Department of Computer Science and Engineering, Southern University of Science and Technology

***Abstract*—The growing need of trusted execution environment (TEE) has boomed the development of hardware enclaves. However, current TEEs and their applications are tightly bound to the hardware implementation, hindering their compatibility across different platforms. This paper presents vSGX, a novel system to virtualize the execution of an Intel SGX enclave atop AMD SEV. The key idea is to interpose the execution of enclave instructions transparently to support the SGX ISA extensions, consolidate encrypted virtual memory of separated SEV virtual machines to create a single virtualized SGX-like address space, and provide attestations for the authenticity of the TEE and the integrity of enclave software with a trust chain rooted in the SEV hardware. By design, vSGX achieves a comparable level of security guarantees on SEV as that on Intel SGX. We have implemented vSGX and demonstrated it imposes reasonable performance overhead for SGX enclave execution.**

I. Introduction

Over the past few years, we have witnessed a tremendous growth of the use of trusted execution environments (TEEs), such as Intel Software Guard Execution (SGX) and AMD Secure Encrypted Virtualization (SEV). TEEs have great promise in protecting both confidentiality and integrity of program code and data from malicious system software and operators, which are extremely valuable for clouds where the computing platform is not fully trusted by its customers. Existing cloud deployment of TEEs includes Alibaba Cloud's SGX VM instances [7], Microsoft Azure's confidential computing [3], Google's confidential virtual machines [4], and so on.

As a prominent TEE platform from Intel, a dominating player in the general-purpose CPU market, SGX has once become the de facto standard for building TEE-based applications. A rich ecosystem with abundant open source projects and commercial products has been built atop SGX, including SGX-based password manager [40], SGX-based anonymity network [38], privacy-preserving data analytics (e.g., [57], [60]) and machine learning (e.g., [41], [50]), SGX-based game protection (e.g., [16], [54]), privacy-preserving contact-tracing (e.g., SafeTrace [1]) and blockchains [19] using SGX, and SGX-based IoT network [48], etc.

However, the ISA extension of SGX mandates a clear separation of software applications into trusted and untrusted components, such that the trusted software components are isolated inside the protected *enclave* regions that are only accessible from code executed in a new CPU mode (i.e., enclave mode). As such, developers need to refactor an existing application or build a new one in accordance with Intel's SGX software specification, and compile it using SGX SDKs, such as Intel SGX SDK [34], and Rust SGX SDKs (e.g., [25], [69]). As a result, applications developed for SGX can only run on SGX processors, resulting in a vendor lock-in situation.

Decoupling TEE software applications from the underlying TEE hardware is a strong desire of the cloud providers. For instance, Google's Asylo project [8] aims to provide a unified SDK interface so that the same TEE source code developed with Asylo can be compiled and run on any TEE hardware; Amazon's Nitro Enclaves [5] use virtualization technology to form secure enclaves, so that confidential workloads can run without SGX. However, neither of these methods can achieve binary compatibility. Ideally, the cloud providers would offer their customers an option to build their applications once, in accordance with the SGX semantics, for instance, given the large volume of existing SGX-based projects, and deploy them on a variety of cloud servers, which may or may not have the hardware capabilities of SGX.

Moreover, the customers should be provided the freedom of choosing the level of trust they have on the cloud providers. For instance, for users who fully trust the cloud providers, hypervisor-based enclaves (e.g., Nitro Enclaves [5]) can be used. But for other users who do not, either SGX or SEV can be chosen from two different levels of trust: SGX features small user-space enclaves with all other software components exposed to the untrusted hypervisor, while SEV protects the entire VM and allows flexible deployment of existing applications, at the cost of a larger attack surface. However, to the best of our knowledge, there is no technique that could combine the benefit of both SGX and SEV so that a user can enjoy SEV-protected VMs for a private computation environment while still be able to run existing SGX enclave binaries.

To demonstrate such a feasibility and practicality, in this paper, we present vSGX, a system that provides binary code compatibility of partitioned SGX enclave software and enables its direct execution atop AMD SEV. Conceptually, vSGX can be considered as an SGX hardware module that is plugged into an SEV machine. The key idea behind vSGX is to leverage the VM protection provided by SEV, and execute trusted enclave of a legacy SGX application in a separated

†Part of this work was done when Shixuan Zhao was visiting SUSTech.
✉ yinqianz@acm.org, zlin@cse.ohio-state.edu

VM, and we call it an *enclave VM* (*EVM*). We call the original VM that runs the untrusted part of the app as an *app VM* (*AVM*). The virtualization is achieved through the interposition of the SGX instructions (e.g., `EENTER`, and `EEXIT`) during their executions in the corresponding VMs or cross-VMs, and implementing the corresponding logic in the VM kernels to offer the transparency to both the trusted enclave code and untrusted application code. The enclave code and data secrecy are achieved by using AMD's memory encryption engine (MEE), and the integrity is achieved by building an attestation service with a trust chain rooted by AMD's SEV attestation. Therefore, we achieve a comparable level of security as SGX while preserving the SEV's security when running SGX applications atop vSGX.

While the idea of virtualizing SGX enclaves using SEV might appear to be simple, it in fact faces many non-trivial challenges (§III). These challenges include how to interpose the execution of enclave instructions in AMD SEV; how to handle enclave entrance and exit since with vSGX an enclave is executed in a separate EVM; how to handle cross memory access between the EVMs and AVMs; how to deal with the untrusted code in the AVM's OS or even a malicious hypervisor; and how to perform SGX remote attestation on AMD machines. We have fortunately addressed these challenges when designing vSGX (§IV).

We have analyzed the security of vSGX and discussed that our design has achieved a comparable level of security as with Intel SGX, through the use of the primitives provided by AMD SEV (§V). We have also evaluated its performance overhead with a set of benchmarks and real world applications (§VI). Our experimental results from the benchmarks show that while many of the enclave instruction executions (particularly `EENTER` and `EEXIT`) are indeed slower when running in SEV compared to running in Intel CPU, these overheads will only be observed by ECall or I/O intensive applications. Our evaluation with real world SGX applications shows that the overhead of vSGX is reasonable. Therefore, we believe vSGX represents a practical way of executing SGX enclaves atop AMD SEV.

In short, this paper makes the following contributions:

- **Novel System:** We present vSGX, a new system that allows the SGX execution atop AMD SEV (including SEV-ES), enhancing enclave applications' inter-TEE operability in a virtualized environment.
- **Comparable Security:** Despite the fundamental design differences between SGX and SEV, vSGX achieves comparable security guarantees to SGX to allow secure execution of SGX enclaves, while preserving the benefits of being protected by SEV.
- **Implementation and Evaluation:** We have implemented vSGX, and systematically characterized its performance overhead. Our results show that it has reasonable overhead for enclave execution when running on AMD SEV, and can be used in practice.

II. Background

A. Intel SGX

Intel SGX provides a trusted execution environment (TEE) by isolating the security-critical component of an application in a hardware-protected enclave. Everything outside the enclave including the OS is untrusted. The only trusted computing base (TCB) includes just the underlying hardware and the enclave itself [32]. In particular, Intel SGX provides the following promises for software running inside the enclave:

- **Confidentiality:** The enclave has its memory encrypted, protected and physically isolated in the Processor Reserved Memory (PRM), and such memory is called Enclave Page Cache (EPC). Any software outside the enclave cannot access the code and data in EPC. In addition, a hardware memory encryption engine (MEE) sits in between of the processor and the memory controller, so that all memory accesses to the PRM region are encrypted and decrypted on the fly.
- **Integrity:** When an EPC page is swapped out from the memory, it is encrypted and authenticated with Message Authentication Code (MAC). Thus, it prevents direct tampering from the outside software. An EPC page can be swapped back once passing the decryption and integrity check.

Intel also provides various platform support, including launching service and remote attestation service so that the enclave and the SGX hardware can authenticate themselves to a remote party.

B. AMD SEV and Extensions

AMD SEV provides a VM-based TEE by encrypting VM's memory without trusting hypervisors or hosts (we will use hypervisors and hosts interchangeably when referring to the virtual machine monitor in the rest of this paper) [37]. Particularly, an AES engine in AMD system-on-chip (SOC) is used to protect VM's data in the memory. SEV VM's data is automatically encrypted by the AES engine when it is written to the memory and is automatically decrypted when it is read from the memory. Each SEV VM has a unique 128-bit VM Encryption Key (VEK), which is stored in AMD Secure Processor (AMD-SP) and never exposed to the platform host. SEV-ES further protects the VM states by encrypting its VM control blocks (VMCB) during VMEXITs [36]. The latest SEV-SNP also includes an inverted page table, such that the integrity of the encrypted memory and the nested page tables is preserved [10].

Similar to Intel SGX, everything outside the SEV VM including co-resident VMs and the platform host is untrusted. Even though the hypervisor can access SEV VM's memory, the AES encryption protects the guest VM's confidentiality and integrity from the hypervisor. However, unlike Intel SGX where the whole enclave is encrypted, SEV supports page-level encryption and the guest VM can control which pages in the memory are encrypted. The guest VM can unset the C-bit in the guest page table entry (PTE) to change an encrypted page to an unencrypted page in order to share data with the hypervisor. Another promising feature of AMD SEV is that the SEV VM requires no application software modifications while some OS kernel modifications are necessary to enable SEV in both the guest side and host side. The support for SEV and SEV-ES has been officially patched since Linux kernel 4.16 and 5.10. A remote attestation framework is also provided by SEV to protect VM's integrity and confidentiality during VM setup.

C. TEE Security

The security of TEEs has been taken under scrutiny since their debut. It has been shown that SGX is vulnerable to various side-channel attacks [18], [27], [30], [43], [59], [61], [74] and more recently speculative execution attacks [20], [22], [58], [67], [68]. These attacks are common on Intel processors, and are more severe on SGX, because SGX assumes a stronger adversary—a malicious OS.

Given a strong security assumption that considers a malicious hypervisor, studies have also shown that SEV is vulnerable to various attacks due to its lack of memory integrity [45], [71], ECB mode of AES encryption and weak tweak function [23], [71], unprotected page tables [49], hypervisor-controlled TLB mechanism [47], unrestricted momentary execution [44], I/O operations [45], and Cipherleaks [46]. Accordingly, new versions of SEV hardware, including SEV-ES [36] and SEV-SNP [10], have been released to address these flaws. It is believed that the latest SEV-SNP could defeat most of the known attacks against SEV. Additionally, SEV is also vulnerable to side-channel attacks [45], [49], [70].

D. Virtualization

Virtualization has been a fundamental technology in modern computing infrastructures, which has enabled modern computing from multi-tasking (where multiple tasks can be executed due to the virtualization of memory and CPUs), to multiple operating systems (where multiple OSes can run simultaneously due to the virtualization of machines) [15]. Without virtualization, modern cloud computing would have not been possible. The key to achieve virtualization relies on the interposition and transparency [55]. With the interposition of virtual to physical address translation (e.g., page directory and page tables), OS can virtualize physical memory to multiple processes. With the interposition of interrupt, page translation, and VM enter and exit, a hypervisor can virtualize a physical machine to run multiple virtual machines (VMs) simultaneously. With transparency, applications or guest OSes will not feel any discrepancies (with the illusion of occupying the whole physical resources of a machine) and run as usual.

There are multiple ways to achieve virtualization. One is through the use of binary translation for the interposition (earlier versions of QEMU fall into this category) [56]. The second one is through para-virtualization to interpose only important instructions (in which guest OS is patched first when running in a VM) [14]. The third one is through interposing hardware events such as interrupts with complete virtualization [65]. When designing vSGX, we explore this third approach by interposing the undefined instruction exception handler and emulating the corresponding instructions with the necessary hardware support from SEV.

III. System Overview

vSGX is a virtualization mechanism that enables AMD SEV processors to transparently execute unmodified SGX enclave binaries with a comparable level of security guarantee while preserve the protection from SEV. In this section, we provide an overview of vSGX, by first describing our design goals (§III-A), followed by our key approach (§III-B) and challenges (§III-C), and finally the threat model (§III-D).

A. Design Goals

There are three main goals when designing vSGX:

- **G1: App binary compatibility.** Orthogonal to LibOS or container approaches where legacy applications can be executed, vSGX aims to run unmodified SGX application binaries on AMD SEV machines without any modifications from enclave programmers.
- **G2: SGX-compatible security for enclaves.** vSGX must achieve a comparable level of security as Intel SGX, such that software running inside a vSGX enclave is protected from any software component outside the enclave, including the hypervisor, the OS, and other enclaves.
- **G3: SEV-preserved security for applications.** vSGX must not compromise or weaken the security provided by AMD SEV for an application, such that applications adopting vSGX are protected from attacks from a malicious or compromised hypervisor.

B. Key Approach

Unlike SGX, which provides isolated and encrypted memory regions within the address space of an application, the security boundary enforced by SEV is the physical memory of an entire VM. Intra-VM isolation is not provided by SEV hardware. As such, to protect enclave code and data from the untrusted application code, the only secure and viable approach is *to run the enclave in a separate SEV VM from the application, and properly handle their instruction execution and communications*.

More specifically, in vSGX, the VM, in which the application runs, is called an *app VM* (*AVM*) and the VM where the enclave runs is called an *enclave VM* (*EVM*). To enforce cross-enclave isolation, only one enclave is allowed to occupy an EVM and an EVM is never reused. With SEV's VM isolation and memory encryption, the application and the hypervisor are not able to access the enclave memory and different enclaves are isolated so they cannot access the memory of each other. To transparently support the enclave binary and the application that are originally built for Intel SGX processors, the hypervisor must provide cross-VM communication mechanisms to help the emulation of the Intel SGX instructions, inter-domain memory accesses, exception handling, and so on.

C. Challenges

Under this multi-VM execution model, to satisfy our design goals, the following technical challenges must be addressed:

- **C1: Instruction emulation:** Although both Intel and AMD machines are x86-64 instruction set architecture (ISA), the SGX extension of the x86-64 ISA is Intel specific and not supported by AMD machines. These extended SGX instructions must be intercepted and emulated by vSGX.
- **C2: Memory management:** SGX embeds the enclave memory inside the application's address space and allows the enclave code to access memory both inside and outside the enclave, while prohibiting accesses to the enclave memory from outside (including another enclave). With the multi-VM execution model, vSGX must still satisfy the same requirement.
- **C3: Enclave entrance and exit:** The semantic of SGX's enclave entrance (i.e., `EENTER`) and exit (i.e., `EEXIT`), namely the world change between untrusted space to trusted space and vice versa, must be preserved in vSGX. The control flow of the enclave code must be preserved as in SGX.
- **C4: Multiple enclaves and multi-threading:** vSGX must be able to concurrently run multiple enclaves (such as the Quoting enclave as in SGX, in addition to application enclaves). Each enclave should also support multi-threading.
- **C5: Remote attestation:** SGX provides a measured launch mechanism for enclave binaries and allows the user to perform remote attestation to verify the authenticity of the SGX platform and the integrity of the enclave's initial code and data. Therefore, vSGX must provide similar functionalities to help the enclave users to establish trust with the vSGX platform and the enclave code. However, by default, SEV only provides attestation for a VM's image. This requires the establishment of a new chain of trust that is anchored at SEV's root of trust.

D. Threat Model

vSGX considers two attack scenarios. Security threats in both scenarios have to be considered. First, with regard to the security of software inside enclaves, vSGX follows the same threat model as SGX and does not trust any software outside the enclave. We assume the adversary may take control of the entire AVM, such as the management of enclave thread scheduling, the virtual memory, and I/O operations. We also assume the adversary is able to launch an enclave that executes any code of his choice. Moreover, we assume the adversary may compromise the hypervisor as well. The collusion among the hypervisor, the AVM, and a malicious enclave (as well as its EVM) represents a worst case scenario in the settings of vSGX.

Second, the security of an application inside the AVM must be preserved with respect to the threat model of SEV, where software inside the AVM is trusted but the hypervisor is not. This must hold even when the application uses vSGX to run an enclave. It requires that vSGX does not increase the attack

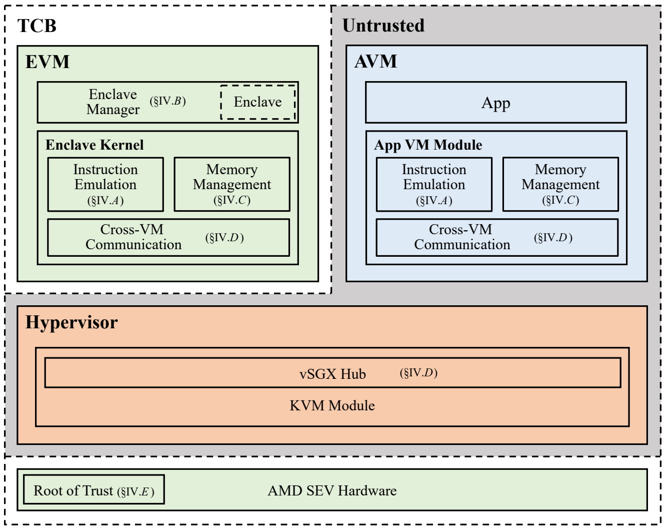


Fig. 1: The vSGX architecture.

surface of SEV. In this setting, vSGX allows the adversary to manage the hypervisor, including the vCPU scheduling, the nested page tables, the interrupt/exception handling, the I/O management, and so on. Nevertheless, vSGX assumes that the variant of SEV it runs on is secure. Notwithstanding the demonstrated attacks against SEV [23], [45], [45], [46], [49], [49], [70], [71], we assume that on SEV-SNP or its successors the integrity and confidentiality of the AVM can be protected from malicious hypervisors.

Out of scope in our threat model are side-channel attacks and powerful physical attacks. While weaker-forms of physical attacks such as DRAM interface snooping [42] and DRAM cold-boot attacks [31] can be thwarted by SEV's on-chip memory encryption engine, powerful attacks such as DDR bus manipulation are not guarded by SEV [37] and thus cannot be prevented by vSGX. We assume transient execution attacks [39] are prevented by hardware countermeasures [6] and pattern-based cache and memory side-channel attacks are mitigated via software hardening.

IV. Detailed Design

The architecture of vSGX is illustrated in Figure 1. There are five components inside vSGX: (1) *Instruction Emulation* (§IV-A), (2) *Enclave Manager* (§IV-B), (3) *Memory Management* (§IV-C), (4) *Cross-VM Communication* (§IV-D), (5) *Remote Attestation* (§IV-E). In this section, we present the detailed design of these components.

A. Instruction Emulation

To address challenge **C1**, vSGX hooks the handler for Invalid Opcode trap (a.k.a., the #UD trap) in the kernels of both the AVM and the EVM, so that SGX instructions can be emulated by vSGX. If the invalid opcode corresponds to one of the `ENCLS` or `ENCLU` instructions, the corresponding functionalities are emulated in the kernel and the relevant registers are modified to reflect the results of execution in

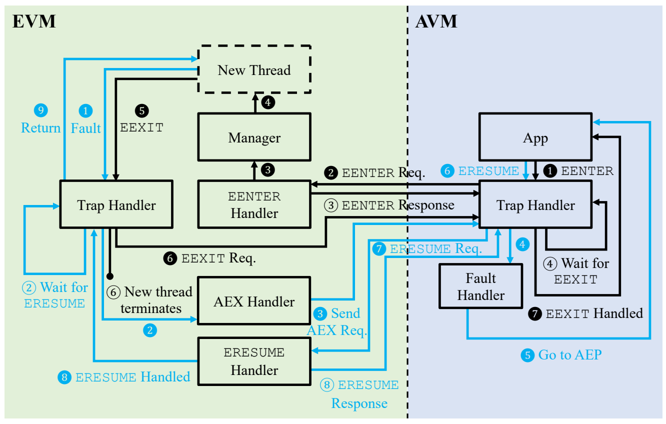


Fig. 2: Entrance and exit of an enclave

the corresponding VMs. vSGX's support of SGX instruction emulation is listed in Table I.

- The emulation of ENCLS instructions are performed across the boundary between the AVM and the EVM. The parameters of an ENCLS instruction, together with necessary data (as in the case of EADD), are packed into a request package, which is sent to the target EVM for execution. On the EVM side, when the emulated instruction finishes, the result is sent back to the AVM as a response, which typically contains an error code and data to be updated in certain registers. One exception is EWB, which encrypts an enclave page and writes it back to untrusted memory. vSGX handles it by packing the encrypted enclave page and its metadata in the response package. To reduce overhead, when EWB fails, the response does not contain the page payload and the metadata.
- The emulation of most ENCLU instructions, such as EREPORT and EGETKEY, can be handled inside the EVM in accordance with the hardware specification, However, the emulation of EENTER, EEXIT, and ERESUME involves control flow transfers across the boundary of the two VMs. The AVM and EVM are instrumented such that only one cross-VM instruction can be executed at a time.

The semantics enclave entrance and exit (per challenge **C3**) are preserved via cross-VM execution (shown in Figure 2). Specifically, the workflow of EENTER is illustrated with black arrows in Figure 2. The #UD trap handler at the AVM intercepts the EENTER instruction (Step ❶), prepares for the corresponding parameters, passes the execution flow to the enclave thread in the EVM (Step ❷). The app thread at the AVM is then paused, waiting for the corresponding EEXIT (Step ④). The execution will be resumed when receiving EEXIT (Step ❼). Upon receiving the EENTER request (Step ❸), the EVM will create a thread or pick up an existing one to handle it (Step ❹). When an EEXIT instruction is executed (Step ❺), the #UD Trap Handler will intercept it, pass the execution flow back to AVM (Step ❻), and terminate the execution thread if necessary (Step ⑥). Steps that can be executed in parallel are labelled with the same index and differentiated with solid and hollow circles in Figure 2.

In Intel SGX, Asynchronous Enclave eXits (AEX) are triggered by interrupts or faults. In vSGX, AEX is supported by vSGX only for faults. Hardware interrupts in the EVM are handled by the trusted enclave kernel without causing AEXs. As illustrated with blue arrows in Figure 2, the trap handlers are hooked to examine if a fault is triggered by an enclave thread (Step ❶). If so, the handler will call the AEX handler (Step ❷) and then put the current thread to sleep, similar to what happens with an EENTER on the AVM side so that vSGX does not need to use XSAVE to save the state of the enclave like Intel SGX. The AEX handler will follow Intel's AEX semantic and generate a synthetic register state then send it to the AVM (Step ❸). The AVM's app thread is then waken up to call the corresponding fault handler with the synthetic state, right inside the #UD trap context of the previous EENTER (Step ❹). When the fault handling is done, either the application is crashed or the fault is handled properly and the control flow goes to the Asynchronous Exit Pointer (AEP), which is specified when executing EENTER by the app [32] (Step ❺), which executes ERESUME (Step ❻) to resume the execution of the EVM (Step ❾).

B. Enclave Manager

An enclave manager is a user-space wrapper process created inside the EVM for hosting enclaves. An enclave binary is loaded as a shared library in a user-space process. When an EVM is launched, an enclave manager is created and then paused to wait to serve enclave creation. Since only one enclave is allowed in an EVM, an enclave manager hosts exactly one enclave (as shown in Figure 1).

Specifically, an enclave manager's workflow starts by creating a new address space (i.e., the enclave context) dedicated for enclave execution that is separated and isolated from its own address space (i.e., the manager context). It then configures two threads: the memory syncing thread (§IV-C) and the dispatcher thread (§IV-D), that run their actions by trapping into the kernel code and then entering the enclave's context. Next, it registers itself to the kernel as a free enclave manager to wait for ECREATE or ELDB/ELDU that creates a new enclave. After an enclave is properly initialized, the manager's main thread then waits for EENTER requests. When an EENTER arrives, it uses pthread to create a new thread for executing the enclave code. The new thread is labelled as an *enclave thread* by a flag in its task_struct in the enclave kernel. Only when the thread is properly setup and out of the control of the manager thread will we swap its context to the enclave context.

To follow SGX's semantics of not allowing an enclave to access any system services by disabling instructions like SYSCALL, vSGX alters the enclave kernel interfaces so that no syscall or software interrupt is available to the enclave thread. The enclave context also does not have vsyscalls mapped. Interrupts will force a thread to be switched back to the manager context, instead of being handled the enclave context.

	Leaf Instruction	Instruction Descriptions	Support	New Enclave	EPCM Modification	Add EPC Page	Remove EPC Page	VM Transmission EVM - AVM	Req. Pkg. Size	Rsp. Pkg. Size
ENCLS	EADD	Add an page to an uninitialized enclave	✓	✗	✓	✓	✗	● ← ◯	4185	19
	EAUG	Add an page to an initialized enclave	✓	✗	✓	✓	✗	● ← ◯	25	19
	EBLOCK	Block an EPC page	✓	✗	✓	✗	✗	● ← ◯	9	19
	ECREATE	Create a SECS page in EPC	✓	✓	✓	✓	✗	● ← ◯	4105	19
	EDBGRD	Read from a debug enclave	✗	-	-	-	-	-	-	-
	EDBGWR	Write to a debug enclave	✗	-	-	-	-	-	-	-
	EEXTEND	Extend uninitialized enclave's measurement	✓	✗	✗	✗	✗	● ← ◯	9	19
	EINIT	Initialize an enclave	✓	✗	✗	✗	✗	● ← ◯	2137	19
	ELDB/ELDU	Load a page to enclave	✓	P**	✓	✓	✗	● ← ◯	8370	4131
	EMODPR	Restrict an EPC page's permission	✓	✗	✓	✗	✗	● ← ◯	12	19
	EMODT	Change an EPC page's type	✓	✗	✓	✗	✗	● ← ◯	12	19
	EPA	Add version array	✓	✗	✓	✓	✗	● ← ◯	9	4131
	EREMOVE	Remove a page from EPC	✓	✗	✓	✗	✓	● ← ◯	9	19
	ETRACK	Block until `EBLOCK` is done	✓	-	-	-	-	-	-	-
	EWB	Write an EPC page to main memory	✓	✗	✓	✗	✓	● ← ◯	4137	8355
ENCLU	EACCEPT	Accept changes to an EPC page	✓	-	✓	✗	✗	◎ -	-	-
	EACCEPTCOPY	Copy a page to a new EPC page	✓	-	✓	✗	✗	◎ -	-	-
	EENTER	Enter an enclave	✓	-	✗	✗	✗	● ← ◯	177	19
	EEXIT	Exit an enclave	✓	-	✗	✗	✗	◎ → ●	153	-
	EGETKEY	Derive a key	✓	-	✗	✗	✗	◎ -	-	-
	EMODPE	Extend permission of an EPC page	✓	-	✓	✗	✗	◎ -	-	-
	EREPORT	Create a cryptographic report	✓	-	✗	✗	✗	◎ -	-	-
	ERESUME	Resume an enclave	✓	-	✗	✗	✗	● ← ◯	33	19
Behavior										
	AEX	Exit an enclave due to interrupt or fault	✓*	-	✗	✗	✗	◎ → ●	166	-

TABLE I: The supported Intel SGX instructions in vSGX: *: Only faults are handled, **: Only when loading a SECS, ●: Emulation done in this VM, ◎: Callee and emulation in the same VM, ◯: Callee in this VM, →: Direction of sending.

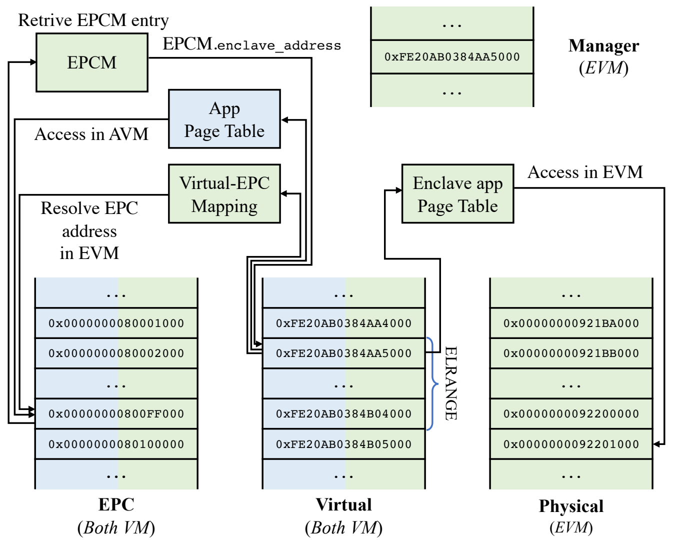


Fig. 3: Virtual memory architecture of vSGX

C. Memory Management

To address challenge **C2**, vSGX incorporates the following components in its design:

(1) Address Spaces: There are four types of address spaces in vSGX: EPC addresses, virtual addresses, enclave physical addresses, and manager addresses. The manager address space is managed by the enclave kernel and the other three address spaces are used and managed by vSGX. The relationship between these address spaces are illustrated in Figure 3. Specifically, the app in the AVM and the enclave in the EVM share the same virtual address space. EPC addresses are used as physical addresses in the AVM and used in the EVM as indices of the Enclave Page Cache Map (EPCM)—the EPC management structure of SGX. Enclave physical addresses are the real backing of the enclave memory which are mapped to virtual addresses in the EVM and used by the CPU to perform addressing. Since enclave apps can use `ENCLU` instructions that consult the EPCM with a virtual address, vSGX uses a virtual-EPC mapping table inside the enclave to help doing this. The entry in this table is added or removed when using `ENCLS` instructions that add or remove an EPC page.

(2) Software EPCM: Intel SGX leverages EPCM to securely manage the virtual address mapping and access permission of a EPC page. To manage EPC pages, vSGX implements a software EPCM, as illustrated in Figure 3. Each entry of the EPCM stores the mapping and permission of each EPC page, just like Intel SGX.

Because the software EPCM is not used by the SEV processor directly, in vSGX, when an SGX instruction modifies an EPCM entry (listed in Table I), we also change the corresponding page table entry in the EVM. By doing so, the EPCM's restrictions can be reflected in regular memory accesses. The page table in EVM is isolated from AVM so it can be trusted.

Some instructions like `ECREATE` and `EPA` can add a page without providing a virtual address. vSGX allocates these pages in the kernel. `ECREATE` is enclave specific so it is allocated within the corresponding EVM. `EPA` creates a version array (VA) page shared across enclaves but each entry of the VA is enclave specific. vSGX creates a VA page for each enclave and encrypts it with an enclave specific key by the EVM's kernel and then stores in the AVM's kernel.

(3) Fetch-and-Map: To allow the code running in the EVM to access out-of-enclave memory in the AVM, vSGX uses a fetch-and-map mechanism that hooks the page fault (#PF) handler of the EVM, from which to fetch the page from the AVM. In particular, when a page fault happens, our #PF handler sends a request to the AVM with the virtual address of the faulting page. If the page is mapped in the AVM, its data is sent back to the EVM so the #PF handler can map a new page initialized with the data received to the faulting address of the enclave execution thread. This process is very similar to the demand-paging mechanism. The page is mapped as non-executable to make sure that code outside the enclave cannot be executed in the enclave mode.

vSGX does not perform fetch-and-map on pages whose virtual addresses fall into the Enclave Linear Address Range (ELRANGE). If a page fault happens in ELRANGE, vSGX will then follow the AEX procedure and inform the AVM to handle the fault.

(4) Switchless Syncing: As vSGX maps the pages between the AVM and the EVM, it has to keep the pages synchronized. Inspired by the concept from switchless OCalls [64], we designed a similar *switchless syncing* mechanism using a background worker thread to synchronize the mapped pages without switching in and out the enclave.

More specifically, both the EVM and the AVM set up a thread during initialization, which are called *switchless syncing worker* threads. When a fetch-and-map event occurs, the page's address and contents are registered into the switchless syncing list on both sides. The worker monitors and synchronizes the changes of every page in the list periodically (e.g., every 100 ms).

Also, to avoid overwriting not-yet-synced changes, we synchronize the page using a 4096-bit bitmap. Each bit of the bitmap corresponds to one byte of the page; a 1-bit indicates the corresponding byte is changed; a 0-bit means an unchanged byte. In this way, this bitmap helps mask out all unchanged bytes so that only those changed bytes will be synced.

D. Cross-VM Communication

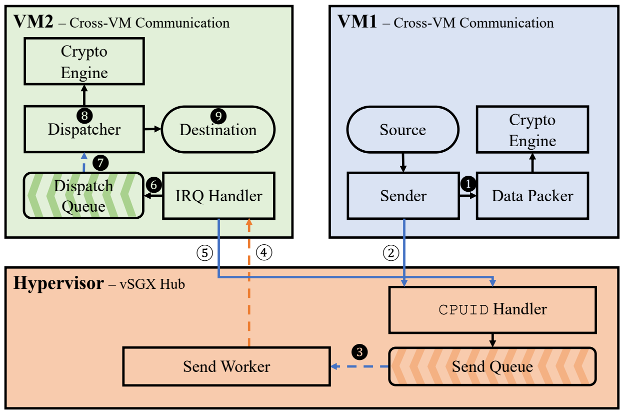
Cross-VM communications are used to serve instruction emulation and memory syncing. Multiple ways can be used to achieve such communication (e.g., using an encrypted TCP connection). However, for better performance, we proposed to transfer data using cross-VM shared pages. Such communication has to involve the hypervisor, which is untrusted in our threat model. Therefore, we must design a secure communication protocol with desired properties.

1) Properties of Cross-VM Communication: The cross-VM communication in vSGX satisfies the following properties.

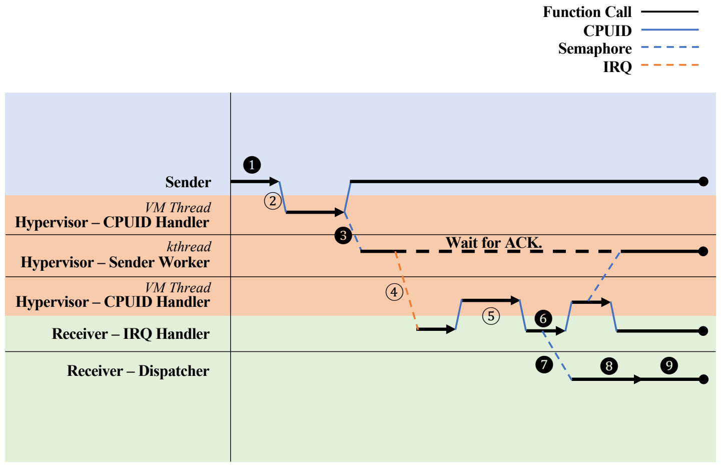
- **Arbitrary Data Size:** vSGX allows arbitrary size of data by slicing large pieces of data into smaller chunks and encapsulating them into *packets*. The size of a packet is the maximum size of data that can be sent in each round of communication. The packet has a fixed-size header which contains the total packet number, the index of the current packet and the total size of the data. The sender will send the packets one by one and the receiver will stitch them together to retrieve the whole data.
- **Concurrency:** Multiple senders may send data at the same time. vSGX needs to make sure that they can send data concurrently without collision. To this end, vSGX includes a unique sequential session number in each packet header. For a large piece of data, we consider the stream of these packets as a single session (e.g., an invocation of an `EADD`). On the receiver side, we can stitch the packets of the corresponding session to rebuild the data.
- **Confidentiality:** vSGX must make sure that the hypervisor cannot read the communication data since it is not trusted. To achieve confidentiality, we rely on end-to-end symmetric encryption such as AES-GCM 128. vSGX encrypts the entire packet including the header to make sure no data is revealed to the hypervisor.
- **Integrity:** A malicious hypervisor might change the data during the sending process. We have to ensure that the data arrives at its destination without being modified. To achieve integrity, we append a keyed Message Authentication Code (MAC) to each packet. Without the proper key, the hypervisor cannot modify the content of a packet.
- **Multiple Targets:** Because there could be multiple EVMs, vSGX should be able to send a packet to a specific VM. We achieve this by assigning an EVM with an EVM ID (e.g., a natural number indicating the order with which the EVM registers itself to the hypervisor). vSGX ensures that the encryption and authentication keys in each EVM are different, so that even if an EVM is compromised, the communications of other enclaves remain secure.
- **Replay-Prevention:** A malicious hypervisor may replay an outdated but legitimate packet to an EVM. vSGX therefore has to make sure that all data arriving at the destination is fresh. This has been addressed by using a unique session number for each data packet, which is also encrypted and integrity protected.
- **Secure Key Distribution:** vSGX assumes shared keys between an EVM and its corresponding AVM. This can be achieved securely, for example, by using a pre-embedded key in the AVM and the EVM, which can be configured by the user before deployment and protected from the hypervisor using image encryption. Other approaches to securely distributing the secret keys may also be implemented.

2) Cross-VM Communication Protocol: The communication protocol consists of 10 steps, which are illustrated in the block diagram in Figure 4(a) with its execution flow in Figure 4(b). In both figures, each step is marked with a circled number either in black or white: black circles represent intra-environment steps and white ones inter-environment steps. Both figures share the same color scheme to reflect which environment the flow is in.

We describe the protocol using an example of transferring data from the untrusted component to the trusted component: In ❶, the sender thread in the AVM prepares for the data



(a) Software Components



(b) Execution flow

Fig. 4: Cross-VM communication.

in a shared memory, and then signals the hypervisor with a CPUID instruction (②), the parameters of which indicate the ID of the target VM, as illustrated in Figure 4. The vSGX hub, a vSGX component in the hypervisor in charge of cross-VM communication, will pick up the packet in the shared memory. Next, the CPUID Handler asks the Send Worker (❸) to inform the target EVM via an IRQ Handler (④). The IRQ Handler will use a CPUID to inform the hypervisor and retrieve the packet to its shared memory (⑤). Then, it runs through a Dispatch Queue (❻ and ❼) into the Dispatcher (❽) where the data is decrypted, stitched, and finally sent to the corresponding data handler (❾). The Dispatcher is a kernel thread worker that dispatches the data to its handler (Destination). The Dispatcher and the Destination Handler are shown as two blocks, but in our implementation the Dispatcher is integrated with the Destination handler.

E. *Remote Attestation*

1) Launching EVM with SEV Attestation: The trust chain of vSGX relies on SEV's remote attestation framework [11] to deploy an EVM with a vSGX provider's encrypted VM image, where any fused secret inside and the image's encryption key are only known by the provider.

Specifically, the root trust of AMD SEV is the AMD root key, K_{ark}, which signs the AMD signing Key, K_{ask}. Both keys are known only to the AMD Key Distribution Server (KDS). During the manufacturing process, each SEV platform is equipped with a pair of chip-unique Chip Endorsement Keys, K_{cek}, the public portion of which is signed by K_{ask}. During the initialization phase of an SEV platform, the SEV firmware generates a Platform Diffie-Hellman key K_{pdh} and a Platform Endorsement Key K_{pek}. K_{pek} is signed by a K_{cek} and K_{pdh} is signed by K_{pek}.

After initialization, the hypervisor retrieves the certificates for K_{pdh} and K_{pek} along with a unique platform ID from the SEV firmware. When the guest owner requests to authenticate the platform (before launching an SEV VM on it), the hypervisor forwards these two certificates and the unique platform ID to the guest owner. The guest owner can then obtain the signed certificates of $K_{cek_{ID}}$, K_{ask} and K_{ark} from KDS using the platform ID and then authenticate the platform by verifying the following certificate chain:

$$K_{pdh} \rightarrow K_{pek} \rightarrow K_{cek_{ID}} \rightarrow K_{ask} \rightarrow K_{ark}$$

To launch an EVM, the following steps are taken: First, the guest owner sends to the hypervisor her DH public key, an encrypted enclave image, which is encrypted by a disk encryption key K_{blk}, and an Open Virtual Machine Firmware (OVMF) file. Next, the hypervisor issues the LAUNCH_START command to pass the guest owner's DH public key to the SEV firmware. A secure channel, encrypted by a DH-derived Transport Encryption Key K_{tek}, is established between the SEV firmware and guest owner. Then, the hypervisor copies the OVMF file into the memory and calls the LAUNCH_UPDATE_DATA command to perform in-place memory encryption of the OVMF file. The hypervisor calls the LAUNCH_MEASURE command to instruct the SEV firmware to calculate a measurement of the OVMF memory, which is sent to the guest owner via the secure channel. Finally, the guest owner verifies the measurement, sends K_{blk} to the hypervisor after encrypting with K_{tek}. The hypervisor uses the LAUNCH_SECRET command to provision encrypted K_{blk} into the launched guest VM, which is used by the OVMF to decrypt and load the encrypted image of the enclave VM. As such, the chain of trust is established as follows:

$$\text{EVM} \rightarrow K_{blk} \rightarrow K_{tek} \rightarrow K_{pdh} \rightarrow \cdots \rightarrow K_{ark}$$

2) vSGX Remote Attestation: vSGX stores the root secret and the hash of the vSGX platform provider's public key in the enclave kernel, which is protected from both the AVM and the hypervisor. The root secret can be used to derive the secret keys using EGETKEY by following Intel SGX's semantics. The public key hash, similar to what described in SGX's manual [32], can enable the vSGX platform provider to launch provider-signed enclaves with a similar capability like the Intel-signed Quoting Enclave. EVMs allow the provider-signed enclaves to derive attestation keys in the enclave kernel. Therefore, vSGX allows enclave code to perform remote attestation using a similar routine as legacy Intel SGX applications.

Specifically, with the capability to launch multiple enclaves, vSGX can support the legacy Intel SGX's remote attestation routine. A special vSGX-signed enclave called the Quoting Enclave (QE) is used to provide remote attestation, just like the Intel SGX. The QE has special vSGX-signed only attribute that allows it to get the signing key of the platform, K_p, which is derived using a pre-deployed fused secret by vSGX's service provider. K_p is used to generate signatures that can only be verified by the vSGX's service provider. By confirming whether the report is properly signed with K_p, we can verify the integrity of both the QE and the enclave binary.

It is worth noting that trusting the signing key K_p implies the trust on the entire EVM kernel image. The measurement of the kernel image, however, is not included in the trust chain. Doing so would require a modification of the OVMF bootloader to enable measured boot of the enclave kernel. We leave the implementation of such a measured boot to our future work.

V. Security Analysis

In this section, we analyze the security of vSGX and discuss how vSGX achieves the desired security goals, namely our **G2** and **G3**.

A. Execution Security

`ENCLS` and `ENCLU` Instructions. On Intel processors, both `ENCLS` and `ENCLU` instructions are implemented using microcode. The execution of these instructions is protected by the CPU hardware and un-interceptable by software programs. While vSGX cannot use microcode to execute `ENCLS` and `ENCLU` instructions, it uses the following approaches to achieve their security.

- `ENCLS` instructions are intended for enclave managements and thus we simply send the parameters of the instruction to the EVM to perform its functions. We use an end-to-end encryption so the hypervisor cannot modify the request. Also we perform the sanity checks specified by the SGX reference inside the EVM so that a malicious request will not succeed. The actual function of the instruction is executed in the EVM and it is thus un-interceptable by software running in the AVM or other EVMs.
- `ENCLU` instructions are mostly executed inside the EVM except for `EENTER` and `ERESUME`. The parameters of `ENCLU` instructions will be checked in accordance with the SGX reference to make sure that they are safe. Since `ENCLU` instructions are executed inside the EVM, their execution can be trusted.

Illegal Instructions inside Enclaves. Enclave code is prohibited from accessing system resources in Intel SGX by disallowing it to execute instructions like `SYSCALL`. vSGX achieves this restriction by performing a check before the entrance of `SYSCALL` and software interrupt handlers to check if the current thread is an enclave thread. If so, the handler will throw a #UD fault. By doing so, we make sure that the enclave code can never use the system services in the EVM and the behaviour is the same as Intel SGX.

Entering and Exiting Enclaves. In vSGX, we follow the exact enclave entering and exiting semantics as SGX to transfer the control flow into the enclave which protects the security of the execution.

- `EENTER` and `EEXIT`: SGX allows the control flow to transfer into the enclaves via `EENTER` and transfer out of the enclaves via `EEXIT`. Unlike SGX, control flow transferring in vSGX crosses the boundary of two VMs. `EENTER` in our implementation will put the app thread to sleep and launch an enclave thread in the EVM. When the enclave thread finishes its execution, `EEXIT` terminates the enclave thread and wakes up the sleeping app thread. The only potential attack vector is to wake up the sleeping app thread early and prevent future `EEXIT`, which can be performed by an adversary with kernel access to the AVM. However, this only affects the apps in the AVM, but not the EVM.
- AEX and `ERESUME`: Like Intel SGX, when a fault happened to an enclave, we transfer back the control flow to AVM with a synthetic state so that the enclave's data including the register state is never leaked. Unlike Intel SGX that uses `XSAVE` to save the state of the enclave, vSGX simply puts the enclave thread to sleep so we can wake it and resume the execution directly when an `ERESUME` comes. The enclave thread sleeps inside the EVM so it is protected from tampering by any adversary. However, unlike SGX, in vSGX AEX is not triggered by interrupts or VMEXITs. This is because the EVM kernel is trusted to handle the unrelated interrupts and exceptions like timer events, and therefore, unlike SGX, context switches are not necessary.

B. Memory Encryption and Isolation

Memory Encryption. Both vSGX and SGX prevent software components outside the EVM (enclave for SGX) from reading encrypted memory in plaintext, and thwart physical attacks, such as cold-boot attacks and DMA attacks, from directly reading secrets in the encrypted memory. Although SEV's memory encryption is not authenticated, and thus is slightly weaker than that of SGX, SEV-SNP does preserve memory integrity. Hence, vSGX achieves comparable security as SGX. Moreover, while SGX uses a single ephemeral memory encryption key for all enclaves [29], SEV uses different keys for different VMs. Therefore, as vSGX protects each enclave in a separate VM, which is encrypted with a different key, vSGX is even more secure than SGX in this sense.

Enforcing Enclave Memory Access Rules and Isolation. Intel SGX prevents accesses to enclave memory if (a) the CPU is not in the enclave mode, (b) the corresponding EPCM entry has *blocked* flag set, (c) the target page is not a `PT_REG` page (i.e., a regular enclave page), (d) the current enclave's

EID is not the same as the owner of the page, and (e) the virtual address does not match the EPCM entry's record [33]. vSGX achieves similar levels of security guarantees via VM isolation. vSGX implements EPCM to maintain the metadata of each EPC page, including the page types (e.g, `PT_REG`), virtual address mapping, access permission, etc.

Although our software-maintained EPCM is not consulted during page table walk, EVM ensures that its page table correctly reflects the corresponding EPCM entries: (1) None `PT_REG` pages (e.g., `PT_SECS`) do not have user-space mapping, so that they cannot be accessed by the enclave code; (2) when a page transitions to *blocked* state, vSGX sets its access permission to `PROT_NONE`, so that the page is not accessible; (3) the access permission and virtual address mapping is correct. Therefore, as the enclave's page table is protected by the enclave kernel in an SEV VM, vSGX maintains the same level of security as SGX. Moreover, because vSGX enforces one enclave per VM without re-using an EVM, with the VM isolation provided by SEV, we are able to achieve enclave isolation just like Intel SGX.

Restricted Non-enclave Memory Access in Enclave Mode. Intel SGX allows code running in the enclave mode to access non-enclave memory. However, it disallows any non-enclave memory to be mapped to virtual addresses inside ELRANGE, which is reserved for enclave memory. Moreover, the TLB entries of non-enclave memory pages loaded in the enclave mode are forced to have the Non-eXecutable (NX) flag set, in order to ensure that the enclave never executes code outside it.

vSGX achieves the same level of security via fetch-and-map and switchless syncing. First, fetch-and-map would never map non-enclave memory to the EVM if its virtual address falls in ELRANGE. Therefore, any memory access to a virtual address in the ELRANGE without a valid mapping will directly trigger a page fault. Second, to prevent executing code in the non-enclave memory, vSGX also forces that the fetched pages are non-executable. We also note that switchless syncing does not leak protected data to the outside, as protected data in the enclave must fall inside the ELRANGE, which will never be fetched or synced with untrusted memory in the AVM.

C. Cross-VM Communication

The only interface an EVM exposes to the outside world is the cross-VM communication interface. In our design, a packet must fall into one of the three categories: An instruction-emulation packet, a switchless-syncing packet and a fetch-and-map packet.

- An instruction-emulation packet is dispatched to its corresponding enclave as specified by the EPC page it operates on. The verification is enforced in the local dispatcher of that enclave according to the Intel SGX specification as discussed in §V-A.
- A switchless-syncing packet is first dispatched to its corresponding enclave. Then, the enclave will check its switchless-syncing list to see if the page to be synced is in the list. If and only if so, the packet is accepted. The correctness of the synced page is not a concern, because non-enclave memory is not expected to be correct.
- A fetch-and-map packet will be compared against a list of thread waiting in the kernel and see if any of them is waiting on the specific address. If so, the packet is accepted and the page is mapped to the non-enclave memory in the EVM. If not, the packet is dropped.

As such, as all packets sent to the EVM is scrutinized, the adversary cannot send arbitrary packets that are inconsistent to the SGX semantics. Moreover, reordering packets does not pose new security concerns. For instruction emulation, because vSGX only allows one cross-VM instruction to be executed at a time, there is no concern that the hypervisor can reorder the execution. For memory-related packets, reordering them can cause overwrite problem in untrusted memory. However because untrusted memory is not protected, this behaviour does not introduce security problem in SGX's model.

D. Discussion on TCB Size

In Intel SGX, the only software component inside the TCB is the enclave binary. However, the microcode implementation of SGX instructions is also part of TCB as they are firmware running on top of the hardware. In vSGX, the TCB contains the enclave kernel, the enclave manager and the enclave binary. In our implementation, we have added $8,840$ lines of code to a Linux Kernel 5.10.20. The enclave manager is relatively small and has only 250 lines of code. So the overall TCB size change in our system is $9,090$ lines of code plus the size of a minimized Linux Kernel. We argue that the Linux Kernel can be replaced with a formally verified kernel such as seL4 once it gets the support of AMD SEV-ES. This allows the whole extra components we have added into the enclave kernel to be fully trustworthy.

vSGX does not significantly increase the attack surface, either. Because the only interface an EVM exposes to the outside is the cross-VM communication interface, the most powerful attack an adversary outside the TCB may launch is the attack against the cross-VM communication interface, such as eavesdropping, injecting, dropping, modifying communication packets. However, as discussed in §IV-D and §V-C, these attacks are prevented via authenticated encryption, replay prevention, and sanity checks performed on the EVM side. Therefore, vSGX can reduce the attack surface of an SEV VM down to a comparable level of SGX.

VI. Evaluation

We have implemented vSGX with $16,167$ lines of C code (LoC) and 121 lines of x86-64 assembly. The AVM module contains $6,377$ LoC and 121 lines of assembly, $8,840$ LoC in the enclave kernel, 250 LoC for then enclave manager and 700 LoC in the hypervisor's KVM module. The source code of vSGX is made available at github.com/OSUSeclab/vSGX. In this section, we present the evaluation result. Since we have answered the security questions of vSGX in §V, in this section we would like to answer the questions related to the performance overhead of vSGX. To this end, we designed and chose a set of benchmarks and real world applications to understand the overhead at both the component level and the application level. A set of microbenchmaks were designed and reported in §VI-A1 to reveal the performance on an instruction and component level; A macrobenchmark software was chosen in §VI-A2 to reflect overall performance. Finally, we also report the compatibility and performance overhead for real world SGX application in §VI-B.

Note that the SEV-ES platform we conducted our experiments on was a GIGABYTE MZ31-AR0 with an AMD EPYC 7251 8-Core Processor running at 2.1GHz. This SEV-ES machine has 64 GiB of memory and is installed with a Linux kernel 5.10.0 provided by AMD to support SEV-ES host capabilities. We configured our VMs with 2 SMP cores and 4 GiB memory each with a Linux kernel 5.10.20. Additionally, we also ran controlled experiments on an Intel SGX machine, which was a DELL OptiPlex 5060 with an Intel Core i7-8700 6-Core Processor running at 3.2 GHz, to compare the performance differences. This SGX machine was equipped with 32 GiB of memory and running Linux kernel 4.15.0. By default, all of the performance overheads were measured by running the target benchmark 10 times and then calculating the average.

A. Benckmarks

1) Microbenchmarks: vSGX has many components responsible for an enclave program execution. At a high level, an enclave will (1) need to be initialized, (2) perform ECall/OCall, (3) execute specific SGX leave instructions, (4) communicate cross-VM, (5) fetch out-of-enclave memory, and (6) perform switchless synchronization if necessary. Therefore, we designed six microbenchmarks to characterize the performance overhead related to these executions.

(1) Enclave Initialization. We first measured the overhead of creating and initializing an enclave on vSGX, and this overhead often involves a set of SGX instructions including `ECREATE`, `EADD`, `EEXTEND`, and `EINIT`. Specifically, we launched enclaves of different sizes (in the number of heap pages, which will be reflected by `EADD` and `EEXTEND`) and report the measured latency in Figure 5 (a) (red line). We can observe that the enclave initialization overhead is mostly linear to its size (since `ECREATE` and `EINIT` is a one-time overhead). We also ran the same set of experiments on an Intel SGX machine. The result is shown in Figure 5 (a) (blue line). For example, to launch a 550-page enclave, it took vSGX on average 0.92s, which is 10x slower than that on Intel SGX (about 92ms). Other data points show similar slowdown. However, we emphasize that as enclave initialization is very infrequent, the amortized overhead is small for the entire life cycle of an enclave.

(2) ECall/OCall Latency. We next measured the latency of ECall/OCall. Essentially, the latency of an ECall is an `EENTER` and `EEXIT` pair, and for an OCall that is an `EEXIT` and `EENTER` pair; we just need to measure one of them. To this end, we measured the latency of an ECall

	Leaf	Average Overhead (μs)	Packets Sent
ENCLS	EADD	1421.23	3
	EAUG	990.20	2
	EBLOCK	840.85	2
	ECREATE	3719.06	3
	EDBGRD	N/A	N/A
	EDBGWR	N/A	N/A
	EEXTEND	986.76	2
	EINIT	811.03	2
	ELDB/ELDU	1958.13	4
	EMODPR	1071.26	2
	EMODT	976.15	2
	EPA	1273.26	3
	EREMOVE	1013.70	2
	ETRACK	N/A	N/A
	EWB	1818.66	4
ENCLU	EACCEPT	0.79	-
	EACCEPTCOPY	2.19	-
	EENTER	N/A	-
	EEXIT	N/A	-
	EGETKEY	5.00	-
	EMODPE	0.91	-
	EREPORT	18.91	-
	ERESUME	N/A	-

TABLE II: SGX leaf instruction performance in vSGX

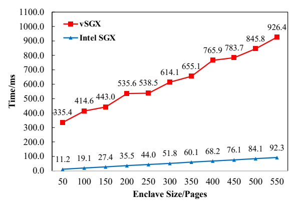
by implementing an empty ECall function, and measuring how long it takes to execute an `EENTER` and an `EEXIT` instruction. Note this empty ECall function also includes code from Intel SGX SDK. We executed this ECall 200 times on both Intel SGX and vSGX. The result is shown in Figure 5 (b). We can observe that on average the time to call an empty ECall is about 1.5ms on vSGX and 9.3μs on Intel SGX, which is 161x faster than vSGX. This 1.5ms overhead implies that the maximum throughput of vSGX's ECall is about 650 IOPS, which we believe is reasonable for most use cases like SGX-protected password authentication.

(3) SGX Leaf Instruction Latency. Next, we measured the latency of executing SGX leaf instructions by measuring the total time of running the instruction 20 times and then calculating the average. Then we repeated this measurement 10 times to estimate the average latency. The results are shown in Table II.

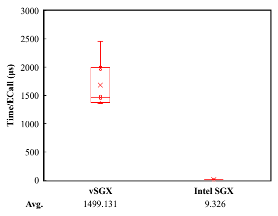
Executing an `ENCLS` leaf instruction involves either 2, 3 or 4 packets. We can observe that an `ENCLS` leaf takes about 0.9ms to finish for a 2-packet instruction, 1.3ms for a 3-packet instruction and 1.8ms for a 4-packet instruction. One exception is the `ECREATE` instruction, which takes about 3ms. This is because `ECREATE` also sets up an enclave manager for the new enclave.

`ENCLU` instructions are normally executed inside the enclave except for `EENTER` and `ERESUME`. The performance of `EENTER`, `ERESUME`, and `EEXIT` used in enclave entrance and exit is measured in our second microbenchmark and thus they are left empty. We can see that in most cases the execution of these local `ENCLU` instructions can be finished within 5μs. Note that `EGETKEY` and `EREPORT` is relatively slower because they involve cryptographic operations.

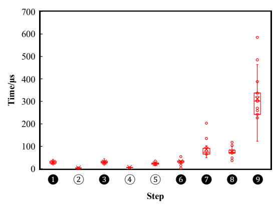
(4) Cross-VM Communication Overhead. To understand the overhead of our cross-VM communication (see Figure 4), we measured the overhead by logging a timestamp before and



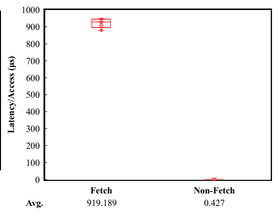
(a) **Enclave initialization overhead**



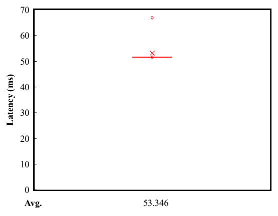
(b) **ECall overhead**



(c) **Cross-VM overhead**



(d) **Memory access latency**



(e) **Switchless syncing latency**

Fig. 5: Result of microbenchmarks of vSGX

after each step. Since the communication crosses the VM boundary, we synchronized the time of our VMs and the hypervisor using the Precision Time Protocol (PTP) [66] to achieve a sub-microsecond precision.

Figure 5 (c) reports the measured latency. Specifically, step ② and step ④ are just CPUID and IRQ event that costs less than 10μs, which is not very significant comparing with others. For other cases, we can see that most of them has a latency under 100μs except for step ❼ and step ❾. This is because step ❼ is using a semaphore to pass data to a dispatcher. Either scheduling or a busy dispatcher can result in long latency. The most time-consuming step is ❾, which handles the instruction emulation of an SGX instruction. The time variation of this step is also huge because different instructions' emulation routines can result in different overheads.

By adding up the latency of all steps, we see that if the packet is the first packet of a 3- or 4-packet instruction, which does not involve step ❿, it takes about 300μs to process. If a packet is part of an instruction, which needs emulation, it takes about 600μs to complete. Overall, with our estimation it takes about 900μs for a 2-packet instruction, 1,200μs for a 3-packet instruction and 1,500μs for a 4-packet instruction. We have a theoretical overhead that is very close to the value we get from the test.

(5) Memory-Fetching Overhead. When accessing untrusted memory from the enclave, there will be memory fetching overhead. We therefore designed a benchmark to measure the latency of accessing untrusted memory, and also compared with the cases in which the page is already fetched. The result is shown in Figure 5 (d). Accessing a local or already-fetched page took about 0.4μs while fetching a page would cause a 0.9ms latency. This result also suggests that once a page is fetched, accessing it would not cause any significant overhead, just like accessing a local page.

(6) Switchless Syncing Overhead. Finally, we would like to characterize how long it takes to have the changes in one VM (e.g., an AVM) to be synced to another VM (e.g., an EVM). We measured the latency of switchless syncing by measuring the round-trip latency then dividing it by 2. Specifically, the EVM first alters a non-enclave page to trigger a switchless syncing; when the change is noticed by the AVM, it immediately changes it to trigger another syncing. When this change is noticed by the EVM, one round-trip syncing is finished. The measurement result is shown in Figure 5 (e). We can see that the average latency is about 53ms. This result is in line with our expectation: As we chose 100ms as our switchless syncing interval, it takes on average 50ms for a page to get synced.

Test	Iterations/second		
	vSGX	AMD SEV-ES	Intel SGX
NUMERIC SORT	396.61	1428.6	2061.3
STRING SORT	45.334	944.38	107.2
BITFIELD	3.9954e+07	5.534e+08	9.0103e+08
FP EMULATION	249.51	693.26	595.62
FOURIER	27528	55657	1.432e+05
ASSIGNMENT	46.701	49.415	76.641
IDEA	9980.4	12214	7329.6
HUFFMAN	4087.7	4332.4	7264
NEURAL NET	70.121	99.638	132.22
LU DECOMPOSITION	478.95	2909.8	3982.4

TABLE III: BYTEmark raw result

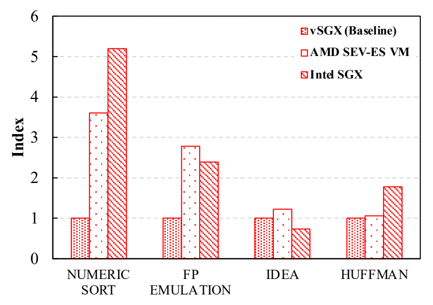
2) Macrobenchmarks: We ran NBENCH [2] (also called BYTEMARK) on vSGX, a vanilla AMD SEV-ES VM, and an Intel SGX machine to compare their performance. NBENCH is a commonly used benchmark for SGX works (e.g., [26], [75]). Table III shows the raw score of the benchmarks run on each platform. We also normalize the scores (by dividing the scores of vSGX) in Figure 6.

First, by comparing the performance scores of vSGX and the vanilla SEV-ES, we can see that the performance overhead (geometric mean) introduced by vSGX on SEV-ES machines is 205%. In Figure 6, we can see that most tests show less than 3x slowdown, except for the STRING SORT, BITFIELD and LU DECOMPOSITION which lead to higher overhead. After examining the code, we found that BITFIELD triggers massive amount of ECalls and STRING SORT randomly accesses large data objects in the non-enclave memory. LU DECOMPOSITION also has a 6x slowdown which is caused by moderate ECalls. From this we can see that for vSGX, two factors can severely impact the performance: ECall frequency and non-enclave memory accesses.

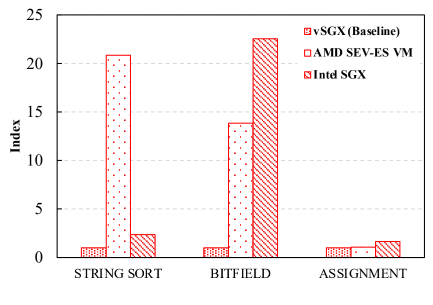
Second, the comparison between vSGX and Intel SGX shows the slowdown when migrating from SGX to vSGX. The geo-mean of the overhead is 221%. The score compared directly across different CPU architecture could be used as a reference on how the app would perform when migrating directly from an SGX machine.

B. Real World SGX Application

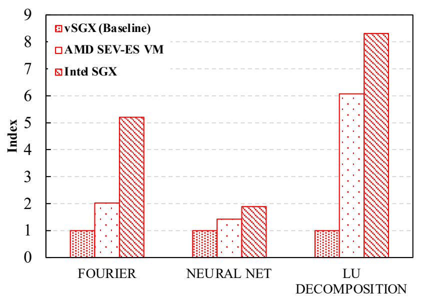
We run several real world applications (shown in Table IV) on vSGX. We particularly present the performance of running WOLFSSL [73] and Graphene, because cryptographic opera-



(a) **CPU Intensive Test**

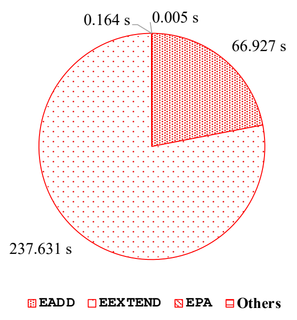


(b) **Memory Intensive Test**

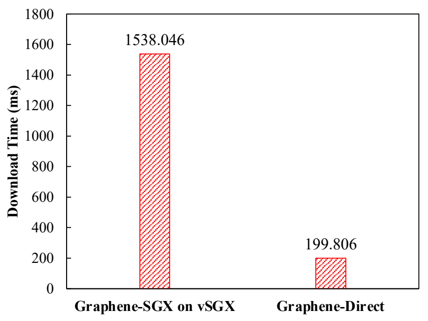


(c) **FP Intensive Test**

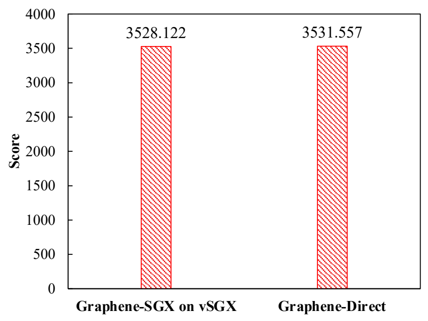
Fig. 6: Normalized BYTEMark result comparison



(a) Time Consumption Launching Graphene SGX on vSGX



(b) **cURL Execution Time**



(c) **GMPbench 0.2 Score**

Fig. 7: Performance of Graphene-SGX on vSGX

App		SDK
Graphene	Samples	None
	cURL	
	Nginx	
	GMPbench	
WOLFSSL		Intel SGX SDK
GMP Library for Intel SGX & Examples		Intel SGX SDK
Intel SGX SDK Sample Enclave		Intel SGX SDK
SGX NBench		Intel SGX SDK

TABLE IV: Apps tested to run on vSGX

tions are typical for enclave apps and Graphene is sophisticated enough to demonstrate the capabilities of vSGX.

To Run an SGX Application in vSGX: For the apps we tested, most of them just run directly without any modification. The exceptions are those checking the CPU family using the `CPUID` instruction (e.g., Graphene), which we had to bypass the check. Applications using Intel-specific instructions like AVX-512 are also not supported.

WOLFSSL Performance: The SGX implementation of WOLFSSL comes with a benchmark named WOLFCRYPT [72]. This benchmark tests encryption, decryption, digests and signature verification. We ran this benchmark on both vSGX and SGX. The result is shown in Table V. The **Ratio** column is the result of Intel SGX's raw performance divided by vSGX's. We can see that for most encryption, decryption, and digest operations, Intel SGX is about 0.5x faster than vSGX. For RSA algorithm and DH key exchange, vSGX could even beat Intel SGX. I/O intensive signature verification and key generation are the weakness of vSGX. The geometric mean of the overhead of the benchmarks shows that Intel SGX is about 0.9x faster than vSGX. Considering we are testing vSGX in a virtualization environment on an AMD's first generation Zen server processor that is architecturally less powerful than an Intel desktop processor, this result is acceptable.

Graphene Performance: We ran cURL, GMPbench and Nginx inside Graphene on vSGX to test its capability of supporting large enclave apps. The launch time of a 256 MB size Graphene is about 5 minutes on vSGX vs. 0.5 second on Intel SGX. This is because launching such a large enclave requires massive amount of `EADD` and `EEXTEND`. The time consumption of each specific instruction when launching Graphene is illustrated in Figure 7 (a). One can easily notice that `EEXTEND` is responsible for 3/4 of the overhead. This is because that each `EEXTEND` can only hash 256 bytes so it requires 16 of `EEXTEND`s to hash a whole 4,096-byte page. A solution is to piggyback contiguous `EEXTEND` requests. However this will lead to changes in SGX's semantics, so we leave this for future work.

To measure the performance of apps after Graphene is launched, we evaluated cURL and GMPbench. The former is an I/O (networking) intensive workload and the latter is CPU bound. We compared Grapehen-SGX on vSGX with Graphene-Direct mode, which runs the library OS directly outside an enclave. For the cURL test, we used it to accesss https://www.ieee-security.org and measured the latency. The performance is illustrated in Figure 7 (b). Graphene Direct is

	vSGX	**Intel SGX**	**Ratio**
	MB/s	MB/s	
RNG	82.57	117.51	1.42
AES-128-CBC-enc	187.36	363.82	1.94
AES-128-CBC-dec	172.59	399.39	2.31
AES-192-CBC-enc	156.95	309.70	1.97
AES-192-CBC-dec	184.4	341.43	1.85
AES-256-CBC-enc	139.01	269.16	1.94
AES-256-CBC-dec	123.05	291.93	2.37
AES-128-GCM-enc	54.10	94.98	1.76
AES-128-GCM-dec	56.02	94.99	1.70
AES-192-GCM-enc	54.36	90.29	1.66
AES-192-GCM-dec	54.49	90.16	1.65
AES-256-GCM-enc	51.78	86.79	1.68
AES-256-GCM-dec	49.74	86.64	1.74
ARC4	138.05	478.18	3.46
RABBIT	222.37	710.37	3.19
3DES	22.60	39.05	1.73
MD5	296.77	820.75	2.77
SHA	223.09	661.65	2.97
SHA-256	115.56	298.76	2.59
HMAC-MD5	377.70	821.12	2.17
HMAC-SHA	381.57	662.07	1.74
HMAC-SHA256	164.82	298.90	1.81
	KB/s	KB/s	
PBKDF2	9.49	34.63	3.65
	op/s	op/s	
RSA 2048 Public	10264.09	8443.25	0.82
RSA 2048 Private	188.40	146.93	0.78
DH 2048 Key Gen	378.24	374.80	0.99
DH 2048 Agree	614.50	375.19	0.61
ECC 256 Key Gen	453.50	6569.28	14.49
ECDHE 256 Agree	1461.67	2201.94	1.51
ECDSA 256 Sign	3611.59	5297.49	1.47
ECDSA 256 Verify	1336.96	1875.64	1.40
Geo Mean			1.90

TABLE V: WOLFSSL's WOLFCRYPT benchmark on vSGX and Intel SGX

about 7x faster than Graphene-SGX on vSGX. This is because the network traffics are handled outside the enclave which caused massive amount of OCalls and accesses to untrusted memory. Besides, the enclave would have to copy those buffer into its own memory causing extra untrusted memory access. The result for GMPbench is illustrated in Figure 7 (c). It can be observed that vSGX does not add burden on CPU computation. These results imply vSGX is better suited for CPU intensive workloads like cryptographic operations.

VII. Limitations and Future Work

vSGX can be improved in multiple avenues. We list some of the ideas to improve vSGX below.

- vSGX currently does not support enclave debugging. Specifically, `EDBGRD` and `EDBGWR` are not supported. We leave the support for enclave debugging to future work.
- The cross-VM memory syncing in vSGX cannot reflect real-time changes. As such, memory barriers and atomic instructions between the enclave and the application code will not behave correctly. This can interfere with locks implemented with atomic instructions sharing with the untrusted world. A solution is to use OCalls to implement locks on shared memory pages.
- vSGX does not yet fully support Intel's `CPUID` semantics. For instance, if the software uses `CPUID` to check if SGX is supported, the check would return negative. This can be supported by software emulation, but as the behavior of `CPUID` is architecture-dependent, we leave it to future work. Currently, we modified the Intel SDK and driver to bypass the application's check in our implementation.
- To improve the overall performance of vSGX, one could remove the expensive cross-VM communication and map the AVM pages directly to EVM's address space. This design choice, however, is only viable if the application memory in the AVM is not encrypted, which contradicts with our current threat model. We will explore this design in future work.
- We assume the implementation of the EVM kernel is secure. A verifiable kernel, like SEL4, can be leveraged in vSGX to build a secure kernel. As enclave binaries do not need support of system calls and I/O, integrating SEL4 into EVM would be feasible.
- vSGX illustrates an implementation of SGX using software, enabling easy extension of existing SGX functionalities without hardware or firmware upgrades. We will explore the use of vSGX as a software-defined enclave implementation in future work.

VIII. Related Works

There are numerous efforts in supporting the growth of the TEE software developer community. In particular, there are a variety of SDKs (e.g., Intel SGX SDK, Rust SGX SDK [69]). Efforts have been made to provide uniform TEE API interfaces to the developer regardless of TEE implementations. Examples include the Asylo framework proposed by Google [13], the Open Enclave framework by Microsoft [51], and the Open Portable Trusted Execution Environment (OP-TEE) [52]. There are also approaches of running legacy code directly in an enclave as demonstrated in SCONE [12], Haven [17], and Graphene-SGX [21]. Others aim to integrate SGX in cloud and containers [9], [62], [63] and to support enclave migration [28], [53]. In this work, we focus on providing binary-compatibility of SGX enclave applications and demonstrating the execution of SGX enclaves on AMD platforms.

There are also efforts to decouple TEEs from hardware. Komodo [24] is such a work representing this direction, and it is a software-defined enclave environment using ARM's TrustZone. The key idea of Komodo is to detach the enclave management such as attestation and memory encryption from hardware, based on the observation that the security properties of SGX does not necessarily have to be implemented fully in the CPU. Komodo is implemented using assembly language with formal verification, thus leading to a trustworthy design. However, unlike vSGX, Komodo does not provide any compatibility to existing software so developers have to adopt the new environment with Komodo's SDK.

Our work is also related to OpenSGX [35], which is an SGX emulation environment implemented with QEMU. It was created at the time when SGX-capable processors were not available. OpenSGX supports majority of the SGX instructions according to the Intel manuals. OpenSGX, however, is only an emulation environment without any protection to the enclave memory. In contrast, vSGX offers SGX-compatible security guarantees with the support of AMD SEV, which paves the way towards its use in production systems.

IX. Conclusion

We have presented vSGX, a novel system to virtualize the execution of Intel SGX enclave atop AMD SEV. With transparent instruction emulation, cross-VM memory synchronization, and tight integration with the SEV-based memory encryption and isolation, vSGX provides binary-compatible support for SGX enclave applications without losing security. We have implemented vSGX and demonstrated it incurs reasonable performance overhead for SGX applications.

Acknowledgments

We thank the anonymous reviewers for their insightful comments, which have significantly improved the paper. Shixuan Zhao and Zhiqiang Lin were partially supported by NSF grant 1834213 and 1834216, and Yinqian Zhang was in part supported by Ant Group.

References

[1] enigmampc/SafeTrace: Privacy preserving voluntary COVID-19 self-reporting platform. https://github.com/enigmampc/SafeTrace. (Accessed on 08/05/2020).

[2] The nbench benchmark ported to sgx. https://github.com/utds3lab/sgx-nbench. (Accessed on 11/27/2020).

[3] Azure Confidential Computing – Protect Data-In-Use — Microsoft Azure. https://azure.microsoft.com/en-us/solutions/confidential-compute/, (Accessed on 8/16/2021).

[4] Introducing Google Cloud Confidential Computing with Confidential VMs — Google Cloud Blog. https://cloud.google.com/blog/products/identity-security/introducing-google-cloud-confidential-computing-with-confidential-vms, (Accessed on 8/16/2021).

[5] Nitro enclaves. https://aws.amazon.com/ec2/nitro/nitro-enclaves/, (Accessed on 8/16/2021).

[6] Affected Processors: Transient Execution Attacks & Related Security Issues by CPU. https://software.intel.com/content/www/us/en/develop/topics/software-security-guidance/processors-affected-consolidated-product-cpu-model.html, (Accessed on 8/19/2021).

[7] Alibaba Cloud Released Industry's First Trusted and Virtualized Instance with Support for SGX 2.0 and TPM - Alibaba Cloud Community. https://www.alibabacloud.com/blog/alibaba-cloud-released-industrys-first-trusted-and-virtualized-instance-with-support-for-sgx-2-0-and-tpm_596821, (Accessed on 8/19/2021).

[8] Advancing confidential computing with asylo and the confidential computing challenge — google cloud blog. https://cloud.google.com/blog/products/identity-security/advancing-confidential-computing-with-asylo-and-the-confidential-computing-challenge. (Accessed on 09/15/2019).

[9] F. Alder, N. Asokan, A. Kurnikov, A. Paverd, and M. Steiner. S-faas: Trustworthy and accountable function-as-a-service using Intel SGX. In Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop, pages 185–199, 2019.

[10] AMD. AMD SEV-SNP: Strengthening VM isolation with integrity protection and more. White paper, 2020.

[11] AMD. Secure Encrypted Virtualization API Version 0.24, 2020.

[12] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, et al. SCONE: Secure linux containers with Intel SGX. In 12th USENIX Symp. Operating Systems Design and Implementation, 2016.

[13] Asylo: An open and flexible framework for enclave applications. https://asylo.dev/. (Accessed on 09/10/2019).

[14] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. ACM SIGOPS operating systems review, 37(5):164–177, 2003.

[15] E. Bauman, G. Ayoade, and Z. Lin. A survey on hypervisor based monitoring: Approaches, applications, and evolutions. ACM Computing Surveys, 48(1):10:1–10:33, Aug. 2015.

[16] E. Bauman and Z. Lin. A case for protecting computer games with sgx. In Proceedings of the 1st Workshop on System Software for Trusted Execution (SysTEX'16), Trento, Italy, December 2016.

[17] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation, pages 267–283. USENIX Association, 2014.

[18] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software grand exposure: SGX cache attacks are practical. In 11th USENIX Workshop on Offensive Technologies, Vancouver, BC, Aug. 2017. USENIX Association.

[19] C. Cai, L. Xu, A. Zhou, and C. Wang. Toward a secure, rich, and fair query service for light clients on public blockchains. IEEE Transactions on Dependable and Secure Computing, pages 1–1, 2021.

[20] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, et al. Fallout: Leaking data on meltdown-resistant cpus. In ACM SIGSAC Conference on Computer and Communications Security, pages 769–784, 2019.

[21] C. che Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In 2017 USENIX Annual Technical Conference, pages 645–658, Santa Clara, CA, 2017. USENIX Association.

[22] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. SGX-PECTRE: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In 4th IEEE European Symposium on Security and Privacy. IEEE Computer Society, 2019.

[23] Z.-H. Du, Z. Ying, Z. Ma, Y. Mai, P. Wang, J. Liu, and J. Fang. Secure encrypted virtualization is unsecure. arXiv preprint arXiv:1712.05090, 2017.

[24] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In Proceedings of the ACM Symposium on Operating Systems Principles, Oct. 2017.

[25] Fortanix. Fortanix Rust Enclave Development Platform, 2020. https://github.com/fortanix/rust-sgx.

[26] Y. Fu, E. Bauman, R. Quinonez, and Z. Lin. SGX-LAPD: Thwarting Controlled Side Channel Attacks via Enclave Verifiable Page Faults. In Proceedings of the 20th International Symposium on Research in Attacks, Intrusions and Defenses, Atlanta, Georgia. USA, September 2017.

[27] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache attacks on Intel SGX. In Proceedings of the 10th European Workshop on Systems Security, 2017.

[28] J. Gu, Z. Hua, Y. Xia, H. Chen, B. Zang, H. Guan, and J. Li. Secure live migration of SGX enclaves on untrusted cloud. In 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pages 225–236. IEEE, 2017.

[29] S. Gueron. A memory encryption engine suitable for general purpose processors. Cryptology ePrint Archive, Report 2016/204, 2016. https://ia.cr/2016/204.

[30] M. Hähnel, W. Cui, and M. Peinado. High-resolution side channels for untrusted operating systems. In USENIX Annual Technical Conference, 2017.

[31] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In 17th USENIX Security Symposium (USENIX Security 08), San Jose, CA, July 2008. USENIX Association.

[32] Intel. Intel Software Guard Extensions Programming Reference, 10 2014. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[33] Intel. Intel Software Guard Extensions Developer Reference for Linux OS, 6 2019. https://download.01.org/intel-sgx/linux-2.6/docs/Intel_SGX_Developer_Reference_Linux_2.6_Open_Source.pdf.

[34] Intel. SDK for Intel Software Guard Extensions, 2020. https://software.intel.com/en-us/sgx/sdk.

[35] P. Jain, S. J. Desai, B. B. Kang, and D. Han. OpenSGX: An Open Platform for SGX Research. In Proceedings of the Network and Distributed System Security Symposium, 2016.

[36] D. Kaplan. Protecting VM register state with SEV-ES. White paper, 2017.

[37] D. Kaplan, J. Powell, and T. Woller. AMD memory encryption. White paper, 2016.

[38] S. Kim, J. Han, J. Ha, T. Kim, and D. Han. Enhancing security and privacy of tor's ecosystem by using trusted execution environments. In 14th USENIX Symposium on Networked Systems Design and Implementation, pages 145–161, 2017.

[39] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In 40th IEEE Symposium on Security and Privacy (S&P'19), 2019.

[40] K. Krawiecka, A. Kurnikov, A. Paverd, M. Mannan, and N. Asokan. Safekeeper: Protecting web passwords using trusted execution environments. In Proceedings of the 2018 World Wide Web Conference, pages 349–358, 2018.

[41] K. A. Küçük, A. Paverd, A. Martin, N. Asokan, A. Simpson, and R. Ankele. Exploring the use of intel sgx for secure many-party applications. In Proceedings of the 1st Workshop on System Software for Trusted Execution, page 5. ACM, 2016.

[42] D. Lee, D. Jung, I. T. Fang, C. che Tsai, and R. A. Popa. An off-chip attack on hardware enclaves via the memory bus. In 29th USENIX Security Symposium (USENIX Security 20), pages 487–504. USENIX Association, Aug. 2020.

[43] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In 26th USENIX Security Symposium, 2017.

[44] M. Li, Y. Zhang, and Z. Lin. Crossline: Breaking"security-by-crash"based memory isolation in amd sev. arXiv preprint arXiv:2008.00146, 2020.

[45] M. Li, Y. Zhang, Z. Lin, and Y. Solihin. Exploiting unprotected i/o operations in amd's secure encrypted virtualization. In 28th USENIX Security Symposium, pages 1257–1272, 2019.

[46] M. Li, Y. Zhang, H. Wang, K. Li, and Y. Cheng. CIPHERLEAKS: Breaking constant-time cryptography on AMD SEV via the ciphertext side channel. In 30th USENIX Security Symposium, pages 717–732. USENIX Association, Aug. 2021.

[47] M. Li, Y. Zhang, H. Wang, K. Li, and Y. Cheng. TLB Poisoning Attacks on AMD Secure Encrypted Virtualization. In Annual Computer Security Applications Conference, 2021.

[48] X. Liu, Z. Guo, J. Ma, and Y. Song. A secure authentication scheme for wireless sensor networks based on dac and intel sgx. IEEE Internet of Things Journal, pages 1–1, 2021.

[49] M. Morbitzer, M. Huber, J. Horsch, and S. Wessel. SEVered: Subverting AMD's virtual machine encryption. In 11th European Workshop on Systems Security. ACM, 2018.

[50] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In 25th USENIX Security Symposium (USENIX Security 16), pages 619–636, Austin, TX, 2016. USENIX Association.

[51] Open enclave: Globals. https://openenclave.github.io/openenclave/api/globals.html. (Accessed on 09/05/2019).

[52] Open portable trusted execution environment (op-tee). https://www.op-tee.org/. Accessed Sept. 22, 2019.

[53] J. Park, S. Park, B. B. Kang, and K. Kim. eMotion: An SGX extension for migrating enclaves. Computers & Security, 80:173–185, 2019.

[54] S. Park, A. Ahmad, and B. Lee. Blackmirror: Preventing wallhacks in 3d online fps games. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pages 987–1000, 2020.

[55] M. Portnoy. Virtualization essentials, volume 19. John Wiley & Sons, 2012.

[56] F. Rodríguez-Haro, F. Freitag, L. Navarro, E. Hernánchez-sánchez, N. Farías-Mendoza, J. A. Guerrero-Ibáñez, and A. González-Potes. A summary of virtualization techniques. Procedia Technology, 3:267–272, 2012.

[57] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy Data Analytics in the Cloud using SGX. In 2015 IEEE Symposium on Security and Privacy, pages 38–54. IEEE, 2015.

[58] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. Zombieload: Cross-privilege-boundary data sampling. arXiv preprint arXiv:1905.05726, 2019.

[59] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. Springer International Publishing, 2017.

[60] F. Shaon, M. Kantarcioglu, Z. Lin, and L. Khan. A practical encrypted data analytic framework with trusted processors. In Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS'17), Dallas, TX, November 2017.

[61] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing page faults from telling your secrets. In 11th ACM on Asia Conference on Computer and Communications Security, 2016.

[62] C. Soriente, G. Karame, W. Li, and S. Fedorov. Replicatee: Enabling seamless replication of sgx enclaves in the cloud. In 2019 IEEE European Symposium on Security and Privacy, pages 158–171. IEEE, 2019.

[63] D. Tian, J. I. Choi, G. Hernandez, P. Traynor, and K. R. Butler. A practical intel sgx setting for linux containers in the cloud. In Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy, pages 255–266, 2019.

[64] 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, pages 22–27, 2018.

[65] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel virtualization technology. Computer, 38(5):48–56, 2005.

[66] A. Vallat and D. Schneuwly. Clock synchronization in telecommunications via PTP (IEEE 1588). In 2007 IEEE International Frequency Control Symposium Joint with the 21st European Frequency and Time Forum, pages 334–341. IEEE, 2007.

[67] 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, pages 991–1008, 2018.

[68] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. RIDL: Rogue in-flight data load. IEEE Symposium on Security and Privacy, 2019.

[69] 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.

[70] J. Werner, J. Mason, M. Antonakakis, M. Polychronakis, and F. Monrose. The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves. In ACM Asia Conference on Computer and Communications Security, pages 73–85. ACM, 2019.

[71] L. Wilke, J. Wichelmann, M. Morbitzer, and T. Eisenbarth. Sevurity: No security without integrity–breaking integrity-free memory encryption with minimal assumptions. 2020.

[72] wolfSSL and wolfCrypt Benchmarks — Embedded SSL/TLS library. https://www.wolfssl.com/docs/benchmarks/. Accessed Dec. 2, 2020.

[73] wolfSSL Embedded SSL/TLS Library — Now Supporting TLS 1.3. https://www.wolfssl.com. Accessed Dec. 2, 2020.

[74] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In IEEE Symposium on Security and Privacy. IEEE, 2015.

[75] W. Zhao, K. Lu, Y. Qi, and S. Qi. Mptee: bringing flexible and efficient memory protection to intel sgx. In Proceedings of the Fifteenth European Conference on Computer Systems, pages 1–15, 2020.