COMPUTERS & SECURITY 70 (2017) 290–306

Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

S-OpenSGX: A system-level platform for exploring SGX enclave-based computing

Changho Choi [a], *Nohyun Kwak* [a], *Jinsoo Jang* [a], *Daehee Jang* [a], *Kuenwhee Oh* [a], *Kyungsoo Kwag* [b], *Brent Byunghoon Kang* [a,*]

[a] *Graduate School of Information Security, School of Computing, Korea Advanced Institute of Science and Technology (KAIST), 291 Daehak-ro, Yuseong-gu, Daejeon, Republic of Korea*

[b] *Software R&D Center, Samsung Electronics, 56, Seongchon-gil, Seocho-gu, Seoul, Republic of Korea*

ARTICLE INFO

Article history:
Received 30 August 2016
Received in revised form 12 May 2017
Accepted 14 June 2017
Available online 27 June 2017

Keywords:
SGX
System emulation
Hardware modification
Thread isolation
Secure systems
Trusted Execution Environment

ABSTRACT

Intel recently introduced Software Guard Extensions (SGX) to enable applications to protect their selected code and data from privileged platform software. As it draws wider attention from the security research community, an open-source emulator for SGX (OpenSGX) has been developed to allow researchers to explore the SGX environment. To the best of our knowledge, OpenSGX is currently the only publicly available SGX emulation platform; however, its system-level support is largely limited owing to its user-mode emulation. In particular, experiments that require system functionalities or device emulation cannot be conducted with the current OpenSGX. To solve this problem, we propose System-OpenSGX (S-OpenSGX), which leverages QEMU's system emulation to provide researchers with full system-level support for exploring SGX enclave-based programming. In this paper, we show the design of S-OpenSGX, including system functionalities such as scheduling, multithreading, page table handling, and SGX paging. Non-trivial issues derived from the difference between user-mode and system emulation, and our approaches to addressing them are also described. Lastly, we utilize S-OpenSGX to experiment with a new mitigation method against the data leaking attack on enclave threads and an APIC (Advanced Programmable Interrupt Controller) device modification to reduce SGX's mode switch overhead.

1. Introduction

Intel has recently released Software Guard Extensions (SGX) to improve application security in desktop and cloud server environments. With added instructions and memory protection logics in x86 processors, Intel SGX provides an isolated region called an enclave where selected code and data are protected from manipulation by other software including privileged software, firmware, and the BIOS (Intel SGX faq). As applications in general depend on platform software such as an operating system (OS) and hypervisor, applications cannot protect their confidential data from being observed by platform software; nor can they ensure the integrity of security-critical pieces of code from potential tampering. With the introduction of Intel SGX, application developers can now assure the secrecy and the integrity of their code and data without depending on the trustworthiness of platform software. Researchers have been earnestly exploring various uses of SGX including Digital Rights Management, corporate data protections, and privacy

* *Corresponding author.*
E-mail addresses: brentkang@kaist.ac.kr (B.B. Kang), zpzigi@kaist.ac.kr (C. Choi), nhkwak@kaist.ac.kr (N. Kwak), jisjang@kaist.ac.kr (J. Jang), daehee87@kaist.ac.kr (D. Jang), okw1003@kaist.ac.kr (K. Oh), kyungsoo.kwag@samsung.com (K. Kwag).
http://dx.doi.org/10.1016/j.cose.2017.06.006
0167-4048/

protections. It is very timely and important to provide researchers with the full capability of exploring new forms of SGX enclave-based programming and related system development.

While the recent releases of Windows SGX SDK (Intel software guard extensions (Intel SGX) sdk) and Linux SGX software (Intel SGX for Linux; Intel SGX Linux driver) allow modification of all software stacks for SGX in an experiment, none of them can provide an experimental modification of hardware components for SGX. However, it would be crucial to freely explore SGX's hardware modifications, because SGX itself is based on a hardware-based solution. Suppose that the current SGX specification contains a certain limitation and that the solution to the limitation may involve modification of the hardware design of SGX. For example, Haven proposes an SGX design modification so that dynamically inserting content in an enclave and modifying page permission are allowed (Baumann et al., 2014). Their proposed methods were tested utilizing Intel's in-house SGX emulator (Baumann et al., 2014) to prove the feasibility of the new design explorations, which are now reflected in the new SGX specification. Moreover, only Intel, the manufacturer of SGX, can change the behavior of SGX hardware. Researchers not associated with Intel, however, should also be able to explore redesigning hardware components in SGX so that they can fill possible security holes in SGX such as thread isolation, secure input/output channels, and live migration of enclaves.

Recently, OpenSGX (Jain et al., 2016) introduces a platform for exploring new SGX designs. When access to Intel SGX was restricted to only a limited group of researchers, OpenSGX opened a door for exploring SGX research by providing the first open-source SGX platform. OpenSGX's platform includes not only its reinterpreted software stack (e.g., the OS layer, libraries, compile tools, etc.), but also an instruction-compatible SGX emulator. With these software stacks and this emulator, OpenSGX enables redesigning of both software and hardware components of SGX and also developing enclave programs.

Unfortunately, OpenSGX contains several essential limitations as a hardware-modifiable platform because the user-mode emulation upon which OpenSGX relies can neither run a separate OS (a guest OS) nor provide device emulation (Qemu internals). OpenSGX instead handles OS emulations in user space, bringing about non-trivial gaps between the emulator and real machines. For example, the page table cannot be modified to link an enclave linear address to an EPC (Enclave Page Cache) page's physical address. Without this modification, more than one enclave process cannot run concurrently and non-contiguous EPC pages cannot constitute an enclave process. The lack of device emulation support in the user-mode emulation is another critical problem. This mode only emulates the CPU. Thus, other virtual hardware devices can neither be inserted nor modified for various hardware experiments. For example, the absence of timer device emulation leaves OpenSGX unable to conduct experiments related to scheduling.

To overcome these non-trivial limitations of OpenSGX, we propose System-OpenSGX (S-OpenSGX), the system version of OpenSGX. S-OpenSGX consists of a System SGX emulator (SystemSGX) and SGX components running on a guest OS (SGX-Guest). With the aid of QEMU's device emulation support in system emulation, SystemSGX is equipped with numerous virtual hardware devices that can be freely modified. SGX Module, one of SGX-Guest's components, is inserted into a guest OS and provides SGX's system functionalities. The added functionalities include scheduling, multithreading, page table handling and SGX paging, which are not supported in OpenSGX owing to the absence of a guest OS and/or limitations of user-mode emulation.

To support these system functionalities, we consider the following issues.

S1. **Scheduling:** Asynchronous Enclave Exit (AEX) should be inserted into an interrupt delivery procedure to securely save the context of the running enclave when a timer interrupt is raised.
S2. **Multithreading:** Each entered thread should maintain separate SGX data structures in EPC. Thread synchronization for the shared memory area is another issue to be resolved.
S3. **Page table handling:** The linear addresses inside an enclave should be linked to EPC pages' physical addresses, which requires allocating and modifying page table entries for the unused address region.
S4. **SGX paging:** Information about the evicted EPC page should be maintained to recover it in loading. When access to an evicted page is requested, the page should be loaded seamlessly in a page fault handler.

Porting OpenSGX to system emulation mode is also not straightforward. Unlike user-mode emulation, system emulation treats a guest virtual address separately from a host virtual address. This leads to a crash when dereferencing data members in guest structures using the instruction translation routines in the system emulation. Another challenge we address here is the resolved address comparison, which requires accessing guest physical addresses in the translation routines, wherein the access to a guest machine's MMU is not available. Finally, implementing SGX's system components, which fit for a system emulator, requires a considerable amount of engineering effort, because the page table entries for enclave processes, the system call table, and the page fault handler should be modified without affecting the rest of the system.

To demonstrate the benefits of S-OpenSGX, we show how effectively a platform modification can cope with a thread attack (Chen et al. 2016; Hsu et al., 2016), which has not been addressed in the current SGX model. In the current SGX model, memory protection is applied to the granularity of a process instead of a thread. Thus, there exists no isolation between different threads belonging to the same enclave. Without a proper thread isolation in place, one compromised thread can freely access critical assets in another thread's stack or Thread Local Storage (TLS). By modifying SGX's memory protection logic to include an additional check of whether a given access is from a valid thread, S-OpenSGX can protect other threads from unauthorized data leaks caused by the compromised thread.

As another example, we also show how a hardware device modification can help address an SGX performance problem. In SGX, every enclave process repeatedly encounters AEXs to handle timer interrupts. As AEXs flush TLB entries for consistent translations (Intel software guard extensions programming reference, 2014), they cause a significant overhead for resuming the process. To address this performance issue, we suggest

modifying an APIC (Advanced Programmable Interrupt Controller) device so that it reduces the frequency of the timer interrupts when the process is running in enclave mode. This fine-grained frequency control has the advantage of reducing SGX's mode switch overhead while not severely violating the reactivity of normal programs. We explored this approach by building a prototype in S-OpenSGX.

Our contributions are summarized as follows:

1. S-OpenSGX provides the first open-source SGX "system" emulation platform that enables modification of the hardware components for SGX, including peripherals.
2. S-OpenSGX overcomes OpenSGX's limitations by providing SGX's system functionalities, including
   - Support of scheduling and multithreading;
   - Page table entry modification for handling non-contiguous EPC pages;
   - A customized page fault handler for SGX paging;
   - A more accurate system call interface.
3. We demonstrate S-OpenSGX's system experimentation capability by introducing SGX thread isolation with a platform modification.
4. We tested an APIC device modification in which timer interrupt frequency in enclave mode is controlled to reduce SGX's mode switch overhead.

2. Background

2.1. Intel SGX (*Intel software guard extensions programming reference, 2014*)

2.1.1. Overview

Intel SGX is the x86 processors' newly added instructions and memory protection. The added instructions are categorized into ENCLU instructions for user privileges and ENCLS instructions for system privileges. ENCLU(S) instructions have a set of leaf functions. ENCLS instructions provide mechanisms for creating an enclave and managing EPC, while ENCLU instructions are in charge of entrances/exits of enclaves and secure key generation.

2.1.2. Terminology

- Enclave Page Cache (EPC): The processor uses the EPC, which consists of chunks of 4 kB pages, to securely store the enclave pages in the main memory. Privileged software manages EPC pages using ENCLS instructions.
- Enclave Page Cache Map (EPCM): The EPCM contains the security metadata for each EPC page. The processor uses EPCM to enforce access control on EPC pages using the following information: read/write/execute permissions on the page and the SECS identifier of the enclave that owns the page.
- SGX Enclave Control Structure (SECS): Each enclave has the SECS, which contains enclave-specific information such as BASEADDR, SIZE, and MRENCLAVE.
- Thread Control Structure (TCS): An enclave has the TCS for each thread being executed in the enclave. It contains metadata necessary for saving or restoring the thread state during the exit from or the entry to the enclave.
- Page Information (PAGEINFO): PAGEINFO is a data structure commonly used as a parameter for SGX instructions. Among PAGEINFO's fields, SECS determines the destination EPC page's SECS and SECINFO determines the page's properties.
- Asynchronous Enclave Exit (AEX): The AEX is the hardware routine for exiting an enclave when exceptions or interrupts occur. AEX securely saves the processor state inside the enclave and replaces it with a fake state.

2.2. QEMU emulator

The Quick EMUlator (QEMU) is an open-source machine emulator (Qemu Wikipedia; Qemu open source processor emulator). In QEMU, the guest indicates the emulated machine while the host refers to the machine on which QEMU runs. QEMU has multiple operating modes. Among them, we focus on the following two modes.

2.2.1. User-mode emulation

User-mode emulation conducts only CPU emulation, which constitutes system emulation (Qemu usermode, howto; Installing and using qemu user-mode emulation). In this mode, QEMU executes Linux programs that were compiled for a different CPU, not including a guest OS (Qemu Wikipedia). User-mode emulation does not simulate a memory management unit (MMU), as the host OS handles guest memory mappings (Installing and using qemu user-mode emulation).

2.2.2. System emulation

In this mode, QEMU emulates a full computer system, including various hardware devices (Qemu Wikipedia). The emulated hardware includes a timer device, interrupt controller, bus, etc. (Qemu: User mode emulation and full system emulation, 2015). This mode maintains a software MMU (Qemu internals). System emulation is much slower than user-mode emulation because the guest OS runs while hardware components are emulated.

2.3. Attack model

S-OpenSGX can be utilized to evaluate various security problems and their defense mechanisms. In this regard, we consider two different attack models. SGX attack model is assumed as our base attack model throughout this paper except when thread isolation is covered.

2.3.1. SGX attack model

We share the same attack model that Intel SGX assumes. That is, the attacker has full control over the system software such as OS kernel. For instance, he can deploy a rootkit to steal security-sensitive information inside an application. However, Denial of Service and hardware attacks are out of scope.

2.3.2. Thread attack model

The attacker aims to obtain a valuable asset contained in a target thread by exploiting a vulnerable thread belonging to the same enclave. We assume that the attacker can request his collaborator with the root privilege for invalidating or bypassing OS-provided memory isolation.

3. System overview

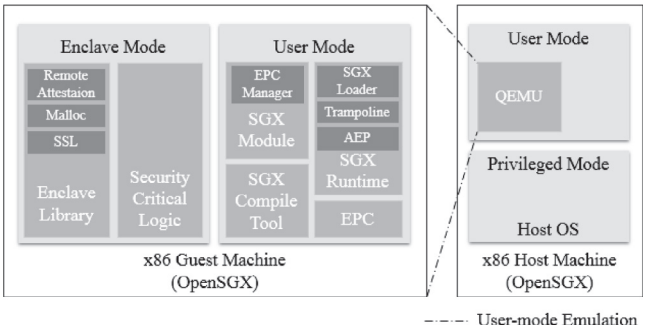
S-OpenSGX consists of SystemSGX and SGX-Guest. SystemSGX is a system SGX emulator and SGX-Guest includes SGX components running on a guest machine such as SGX Module, SGX Runtime, SGX compile tool, or in-enclave libraries (see Fig. 1).

SystemSGX is an extension of QEMU's system emulator. SystemSGX creates a guest machine where all the hardware components such as CPU, memory, and peripherals are emulated. In addition to QEMU's system emulation functionalities, SystemSGX is equipped with SGX functionalities such as interpreting SGX instructions and enforcing memory protection. On top of the guest machine, a guest OS separated from the host OS is installed and executed.

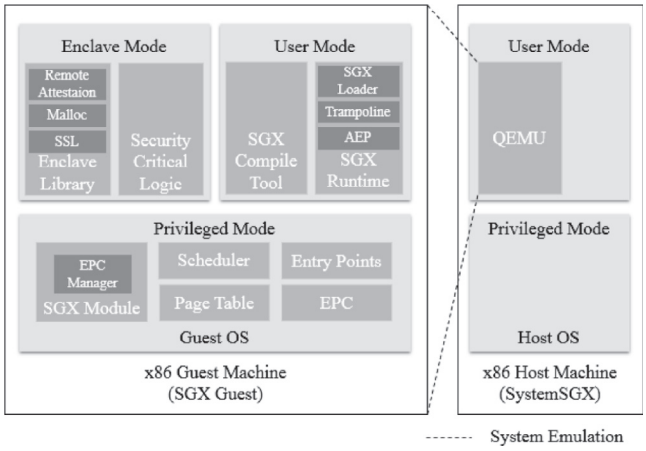
SGX Module is a dynamically inserted kernel module in the guest OS used to support SGX's system functionalities: SGX system call handling, SGX paging, etc. SGX Module dynamically allocates an EPC region in the guest OS's kernel region when the module is inserted. Upon system call requests, SGX Module creates or destroys enclave regions. The EPC manager, as a part of SGX Module, records which EPC page is used and directs EPC page eviction when the number of free EPC pages is lacking. To run an SGX binary, SGX Module should be inserted.

SGX Runtime is an SGX-supporting system library running in user mode. SGX Runtime includes an SGX loader, trampoline, and Asynchronous Exit Point (AEP). SGX Runtime receives an SGX binary as its input to run the binary. SGX Runtime prepares for the contents of the binary to be included in an enclave with SGX loader and makes a system call for an enclave creation request. Once an enclave region is created by SGX Module, SGX Runtime can enter into it with the EENTER leaf function. The trampoline is the entry point where an enclave process temporarily exits to request system services. AEP is the entry point for asynchronous exit events such as interrupts and exceptions.

In S-OpenSGX, the SGX compile tool compiles security-critical logic together with enclave libraries into an SGX binary. SGX Runtime runs in user mode while both security-critical logic and enclave libraries run in enclave mode. Because enclave mode cannot directly invoke system services, enclave libraries are modified to utilize SGX Runtime's trampoline.



(a) OpenSGX with user-mode emulation.



(b) S-OpenSGX with system emulation.

Fig. 1 – S-OpenSGX overall architecture.

4. Design

In this section, we describe S-OpenSGX's added system functionalities compared to OpenSGX; these differentiate S-OpenSGX from OpenSGX in terms of support for system-level SGX emulation. We illustrate the design of S-OpenSGX based on the following aspects: CPU, memory, and kernel entry point management.

4.1. *CPU management*

CPU management is one of the core OS features. OSs manage CPUs by distributing finite CPU resources to multiple tasks (processes or threads) through scheduling. S-OpenSGX's scheduling and multithreading follow Intel SGX's design objective, which mandates that enclaves should not disrupt the ability of legitimate system software to schedule (Intel SGX for dummies (Intel SGX design objectives)). In addition to scheduling and multithreading, S-OpenSGX's unique thread isolation is described in this section.

4.1.1. *Scheduling*

Process scheduling, the activity of selecting a process to run on a particular policy, allows multiple processes to share the CPU using time multiplexing (Operating system – process scheduling). Intel SGX is designed to provide scheduling for enclave processes without modification of system software. This is achieved by an Asynchronous Exit (AEX) mechanism. Whenever a timer interrupt is generated in the execution of an enclave process, the process saves its context and leaves enclave mode with an AEX (Intel software guard extensions programming reference, 2014). The program control then reaches the timer interrupt handler located in the kernel by referring to the Interrupt Descriptor Table (IDT). In the timer interrupt handler, scheduler code is invoked and the OS chooses the next process to run (Bovet and Cesati, 2005).

OpenSGX does not provide scheduling because the user-mode emulator on which OpenSGX is based assumes that only a single process is running on a machine and no timer device is attached. The user-mode emulator can only handle application-generated exceptions, not hardware-generated exceptions/interrupts such as timer interrupts. To simulate the situation where multiple enclave processes are running, OpenSGX should execute multiple QEMU instances. In effect,

this means running each enclave process on a different machine as one enclave process cannot interact with other enclave processes. To run multiple enclave processes on the same machine, scheduling is an indispensable feature.

To provide S-OpenSGX with a scheduling feature, we utilize the same AEX mechanism described by the Intel SGX specification (Intel software guard extensions programming reference, 2014). QEMU's system emulation is equipped by default with a timer device. With each tick elapsed in the timer device, the device generates an interrupt and sends the interrupt to the Local Advanced Programmable Interrupt Controller (LAPIC). The LAPIC then conveys the assigned interrupt vector number (e.g., 239) to the corresponding CPU core. Before program control jumps to the locations to which IDT's entry points, we execute the inserted AEX routine upon receiving a hardware interrupt (if enclave mode is on) so that an enclave process can save its context safely. After AEX, the timer interrupt handler addressed by IDT is executed and the next process can be scheduled.

Because AEX replaces the program control register with an Asynchronous Exit Point (AEP), which is defined by EENTER/ERESUME, the AEP is pushed onto the stack as the address to return to (McKeen et al., 2013). After IRET is executed at the end of the timer handler, the AEP is popped from the stack, and the program control is transferred to the AEP. The AEP simply invokes ERESUME, and ERESUME restores the interrupted enclave's context (e.g., RIP, RAX, FS, etc.) saved in the AEX. Thus, the enclave process can resume executing from the interrupted point seamlessly in scheduling.

An enclave's context is securely protected during scheduling through AEX mechanism and EPC protection. AEX pushes the context in the running thread's private region called State Save Area (SSA), and replaces it with a synthetic state (Intel software guard extensions programming reference, 2014). This synthetic state, while accessible, prevents an attacker from obtaining a meaningful information of the interrupted enclave. The attacker also cannot pass an incorrect context to the resuming enclave by replacing the saved context, because SSA is located in the EPC, to which all the accesses from illegal entities are denied.

4.1.2. *Multithreading*

Multithreading enables multiple threads composing one process to run concurrently while sharing the process' resources such as executable code and global variables (Thread (computing) Wikipedia). Intel SGX is designed to support multithreading in enclave programs by providing the TCS, which manages each thread of an enclave process. The TCS contains the entry point of a thread and starting/ending points of thread-specific EPC pages such as the State Save Area (SSA) and Thread Local Storage (TLS). Hardware utilizes the SSA for storing the context of a thread in an AEX event, whereas software, a thread code, utilizes TLS for storing private data. Intel specifies that multithreading in SGX can be enabled by allocating multiple TCSs (McKeen et al., 2013).

OpenSGX does not support multithreading because user-mode emulation causes crashes in emulating multithreading.[1] However, S-OpenSGX emulates the multithreading of Intel SGX by utilizing the multithreading supported by system emulation. The system emulator's multithreading support is insufficient for emulating multithreading in SGX because each thread in enclave mode should maintain a different context. If more than one thread tries to have the same context by utilizing the same TCS at an enclave entrance, the CPU raises an exception for the latter threads as they are attempting to access a TCS whose state is already occupied by the first thread.

To maintain a separate context for each thread, our SGX Module allocates multiple TCSs as well as multiple thread-specific EPC pages (SSA, TLS, and stacks) upon the system call requests for creating an enclave. The SSA and TLS are naturally connected to their corresponding TCS as the TCS keeps their information in its field. However, the TCS has no field about stacks, while a per-thread stack area is required. Thus, we connect each stack area to a TLS that the corresponding thread can utilize separately. For this purpose, SGX Module stores the per-thread stack's virtual address in the TLS after allocating the TLS so that each thread can securely obtain its stack's address by referring to its TLS (e.g., movq fs:(0), rsp) (Costan and Devadas, 2016). At the end of enclave creation, the module stores a list of the TCSs' virtual addresses in an enclave descriptor, and the host process that requested creating the enclave can later access it through another system call, `sys_stat_enclave()` in OpenSGX (Jain et al., 2016). After creating an enclave, multiple threads forked from the host process can concurrently enter into the enclave with their own TCS with the aid of multithreading support in system emulation.

Multithreading brings about the consideration of thread synchronization. S-OpenSGX, as with OpenSGX, maintains a dedicated shared memory area called a stub for a strict form of communication between an enclave code and its host code (Jain et al., 2016). Because a stub is also shared between threads in the same enclave and frequently accessed for I/O, access to this resource should be properly controlled with thread synchronization. For example, without thread synchronization, the output of a multithreaded SGX program becomes unordered and is omitted. To avoid this problem, S-OpenSGX supports a simple lock mechanism and applies it to the uses of stubs. We include the lock in stubs so that both the enclave code and the host code can easily access it. The enclave code acquires the lock before accessing a stub for sending data to its host code, and later the trampoline (the entry of the host code) frees the lock after accessing the stub to receive data from the enclave code.

4.1.3. *Thread isolation*

Intel SGX provides memory protection in the granularity of an enclave. This means that threads belonging to the same enclave are trusted mutually. Under this assumption, compromising one thread of an enclave can influence other threads of the same enclave. For example, an attacker can control his/her compromised thread to access private information in another thread. To prevent this, we propose a thread isolation method in our SGX model by modifying data structures and memory protection logic in Intel SGX. A thread isolation provides memory protection in the granularity of a thread instead of an enclave. Thus, compromising one thread in an enclave cannot affect the other threads of the same enclave (see Fig. 2).

[1] There are some locking issues in user-mode multithreading where the translated cache flush is not protected against reentrancy (Qemu internals).

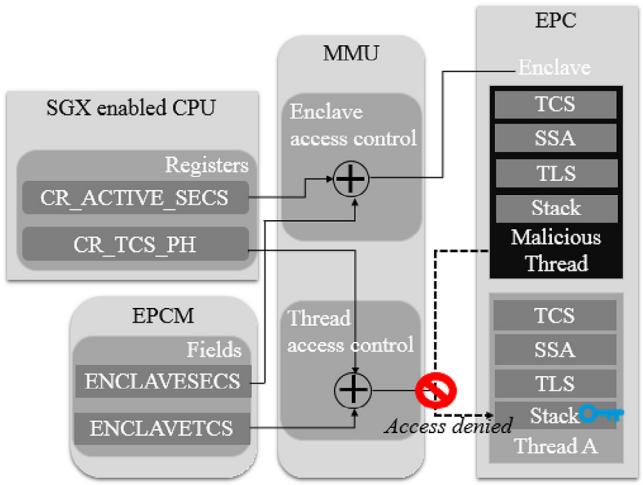


Fig. 2 – S-OpenSGX with a thread isolation.

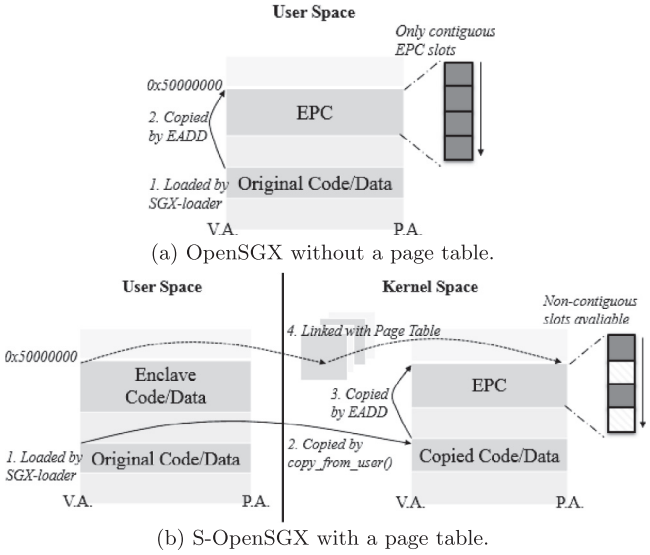
We illustrate the original SGX access control and how a simple modification to it can help defend against thread attacks. When the CPU accesses a linear address, the MMU translates the address into a physical address and checks the permission of the corresponding page table entry against the access. After this traditional page table check, the SGX-enabled CPU examines whether the access occurred in enclave mode. If so, an access resolved to the EPC region goes through EPCM checks. An EPCM entry is similar to a page table entry in that read, write, and execute permissions can be set to determine the access control of the corresponding EPC page. However, in contrast to a page table entry, an EPCM entry can only be configured through ENCLS(U) instructions with a PAGEINFO data structure. This ensures that malicious system software cannot modify the EPCM. In addition to permission checks, the EPCM is used to check whether the access of the EPC is conducted from the owner of the page, the enclave. This is achieved by the combination of ENCLAVESECS fields in EPCM entries and CR_ACTIVE_SECS register. ENCLAVESECS is the identifier of the enclave to which the page belongs, and CR_ACTIVE_SECS indicates the currently running enclave. If CR_ACTIVE_SECS does not match the ENCLAVESECS in the EPCM entry of a target EPC page, then access is denied. Thus, only the enclave that owns the page can access its EPC page, preventing other enclaves from accessing it.

We suggest inserting TCS fields into PAGEINFO structures and ENCLAVETCS fields into EPCM entries. The TCS field in PAGEINFO is used to fill the ENCLAVETCS in an EPCM entry that determines which thread owns the target EPC page, and the ENCLAVETCS in the EPCM entry aids in enforcing the thread isolation. In Intel SGX, system software passes PAGEINFO as a parameter of the ENCLS instructions (e.g., EADD or EAUG) to determine the EPCM entry fields of the destination EPC page as the values specified in PAGEINFO (Intel software guard extensions programming reference, 2014). In our design, the TCS in PAGEINFO should be filled with the effective address of the TCS page of the target thread. The TCS in PAGEINFO can be referenced in order to overwrite the ECNLAVETCS in the destination EPC page's EPCM entry upon successful execution of an ENCLS instruction. The ENCLAVETCS in the EPCM entry indicates that the target EPC page is owned by the thread that the TCS page of the ENCLAVETCS controls. In the case where an EPC page needs to be shared by multiple threads in the same enclave (e.g., code or global data), the TCS in PAGEINFO can be simply filled as null to indicate that the thread isolation should not be applied to that page.

The other component necessary for enforcing the thread isolation is the CPU's context information about a thread. It naturally exists in SGX-enabled CPUs in the form of CR_TCS_PH registers. Similar to CR_ACTIVE_SECS, CR_TCS_PH contains the physical address of the TCS that controls the currently running thread and the value of the CR_TCS_PH is determined only upon the entrance into enclave mode (e.g., EENTER and ERESUME). By referring to this register, the CPU can determine in which thread's context it is running inside the enclave mode. We insert into the MMU a simple logic for examining whether the value of the CR_TCS_PH is the same as the value of the ENCLAVETCS in the EPCM entry to check that a valid thread is accessing the EPC page. If a thread accesses the EPC page that belongs to the same enclave but not the same thread, the access will be denied. Through this mechanism, a thread can be isolated.

Our approach does not trust in OS for enforcing the thread isolation. This is desirable in SGX attack model where OS is excluded out of Trusted Computing Base (TCB). In contrast, previous thread isolations such as Shreds (Chen et al. 2016) or Secure Memory View (SMV) (Hsu et al., 2016) all include OS in their TCB. An attacker with the root privilege can bypass their isolations by modifying isolation-related metadata that OS maintains. In our case, metadata for enforcing the thread isolation are all contained either in SGX's protected memory area or CPU registers that are only configured through SGX instructions. Thus, a manipulation attempt can arise on the isolation configuration. However, if an attacker fills the TCS field in PAGEINFO with an incorrect value, it would either be rejected (non-TCS value) or harm the availability of the application (another thread's TCS value). The attacker might also intentionally make a thread-owned page as a shared page to circumvent our thread isolation. This can be prevented by reflecting the hash value of each thread-owned page's TCS contents on enclave identity (MRENCLAVE) in the enclave building time (EADD). Then, an attacker-converted shared page will generate a different enclave identity which can be identified in SGX's local/remote attestation procedures (Anati et al., 2013).

The thread isolation should be harmonized with all SGX instructions. For example, thread isolation should not be detoured in paging instructions (EWB, ELD(B)), where an incorrect TCS value can be passed as the thread owner of the page to be loaded. Because TCS's physical address can change in paging, we additionally introduce a thread id as another thread identifier to supplement TCS. Our thread id is assigned to TCS in EADD when the inserted page's page type is TCS and can only be referenced in the paging instructions. Similar to an enclave id (Costan and Devadas, 2016), we pass a thread id as one of the inputs to authenticated encryption in SGX paging. This prevents attackers from passing an incorrect TCS address during EPC page loading, because the MAC extracted from an incorrect thread id of an incorrect TCS will be different from the original MAC. We argue that our thread isolation does not have the side effect of degrading the original security guarantee



(a) OpenSGX without a page table.

(b) S-OpenSGX with a page table.

Fig. 3 – Memory layout differences between OpenSGX and S-OpenSGX. "V.A." indicates a virtual address while "P.A." indicates a physical address.

provided by Intel SGX because the change is minimal and analogous to the original enclave isolation.

4.2. *Memory management*

Memory management is another core OS feature. OSs have been advanced over many generations to manage memory well, and Intel SGX is designed not to disrupt this crucial role (Intel SGX for dummies (intel SGX design objectives)). In this section, we describe S-OpenSGX's memory management in terms of the EPC, the protected memory region utilized in SGX. We begin the section by introducing S-OpenSGX's EPC management. Topics related to the page table and how to oversubscribe a finite EPC resource are described next.

4.2.1. *EPC management*

The EPC is SGX's special memory region. To manage the EPC, S-OpenSGX conducts different actions according to the EPC's life cycle: creation, reservation, freeing, and destruction. We describe EPC management in a pair of these actions.

EPC Creation/Destruction: One major difference between Intel SGX and OpenSGX is that the former initializes EPC pages at booting time with the aid of the BIOS, whereas the latter creates and initializes them at runtime. S-OpenSGX chooses the second scheme for easy adoption and flexibility. OpenSGX creates EPC pages in the user space; however, S-OpenSGX does so in the kernel space (see Fig. 3). In this aspect, S-OpenSGX's EPC pages are closer to those of Intel SGX, where each page is managed in kernel space, than those of OpenSGX. In addition, S-OpenSGX creates and removes EPC pages upon SGX Module loading and unloading, respectively, while OpenSGX does so upon system call requests. The reason for relocating EPC creation/removal to module loading/unloading is natural: EPC pages are needed for every enclave process, and the time of demand for EPC pages corresponds to the time of module loading. Disposing of them upon a system call request, in contrast, allows either one initial process or every enclave process invoke that system call, neither of which is desirable.

EPC Reservation/Freeing: S-OpenSGX reserves EPC pages by utilizing the same procedure in OpenSGX. Upon an enclave creation request, the EPC manager chooses free EPC pages and marks them as reserved for an enclave process. SGX Module obtains those reserved pages from the EPC manager and uses them as destination EPC pages for a series of ENCLS instructions: ECREATE, EADD, EEXTEND, and EINIT. S-OpenSGX newly supports enclave destruction similar to that provided by Intel SGX SDK (Intel software guard extensions (Intel SGX) sdk). Upon an enclave destruction request, SGX Module finds all EPC pages related to the requested enclave id and gives those pages as input to EREMOVE. After executing EREMOVE, the EPC manager marks the pages as free.

EPC Protection: Upon EPC creation, additional memory protection is applied to the EPC area. Traditional memory protection is configured by system software (e.g., access flag configuration in page table), and this cannot be used for EPC protection, where an attacker is assumed to have kernel privilege. EPC protection by default prohibits either system software or an application from accessing the EPC area (Isca 2015 tutorial slides for Intel SGX, 2015). Because this rule cannot be modified, the EPC area is free from an attacker's manipulation. While EPC is accessible to an enclave, the enclave's access to an EPC page is examined to determine its approval by referring to the corresponding EPCM entry.

4.2.2. *Page table handling*

In modern computing systems, memory is virtualized to give the illusion that every process has its own memory. The memory view of a process is called a process address space. The enclave region is also a part of the process address space, and accessing the region is conducted through enclave linear addresses (McKeen et al., 2013).

OpenSGX does not support page tables, as page tables require system software with which user-mode emulation is not equipped. Instead, OpenSGX lets an enclave process possess a set of contiguous and fixed EPC pages and assumes that accessing the EPC pages' virtual addresses is accordant with accessing enclave linear addresses. This design is tolerable for running only one enclave process; however, page tables are indispensable for running multiple enclave processes concurrently. This is because, when multiple enclave processes exist, the EPC pages possessed by an enclave process can be dispersed with dynamic EPC page allocation (Xing et al., 2016), and their location can be moved on runtimes with SGX paging (Section 4.2.3). Treating EPC pages' virtual addresses as enclave linear addresses is then no longer valid. To handle these non-contiguous and relocated EPC pages seamlessly, S-OpenSGX makes use of page tables.

Fig. 3 illustrates the differences in memory layout between OpenSGX and S-OpenSGX. In the case of OpenSGX, after the SGX loader loads the target enclave contents (e.g., code/data) to somewhere in the process address space, EADD copies the contents to the predefined address of the EPC area (e.g., 0×50000000). The address is assumed as an enclave linear address as well as an EPC page's virtual address. In the case of S-OpenSGX, loading enclave contents to process address space by the SGX loader is the same as in OpenSGX. However,

the enclave contents are copied to kernel address space before being passed directly to EADD instructions. This is because EADD receives the kernel's virtual address as the location of a source page. After EADD successfully copies the contents to the EPC area, S-OpenSGX links an enclave linear address (here, 0×50000000) with the physical address of the destination EPC page using a page table. As a result, when the corresponding enclave process accesses the enclave linear address, it has the effect of accessing the EPC page that the process perceives as its contents.

4.2.3. *SGX paging*

Paging generally refers to a memory management scheme for oversubscribing main memory by storing and retrieving data from secondary storage (Paging Wikipedia). In Intel SGX, EPC paging indicates eviction and loading of EPC pages for oversubscribing a finite EPC (McKeen et al., 2013; Intel software guard extensions programming reference, 2014). Specifically, the evicted EPC page is stored in main memory after authenticated encryption, which outputs a cryptographic MAC and an encrypted page given the page to evict, a processor key, the page's inherent metadata, and a counter as input. The counter used for encryption is stored in one of the slots in the Version Array (VA) in the EPC. The evicted page can be reloaded back to the EPC only if the newly calculated MAC – from the encrypted page, the processor key, the metadata, and the counter – matches with the previous MAC (Intel software guard extensions programming reference, 2014).

OpenSGX does not support EPC paging functionality even though instructions related to EPC paging (EWB/ELD(B)) can be emulated in OpenSGX. This is because the user-mode emulator assumes that only one process is running on a given machine; however, EPC paging is meaningful only when more than two enclave processes are running. In such multiple-enclave process environments, one enclave process can yield its EPC pages to other enclave processes. Another reason OpenSGX cannot support EPC paging is that EPC paging requires handling page fault exceptions with a customized page fault handler because accessing the page that was previously evicted generates a page fault. However, the limitation in user-mode emulation prevents OpenSGX from implementing a customized page fault handler.

S-OpenSGX provides EPC paging by hooking an existing page fault handler. EPC paging in S-OpenSGX works as follows. If a new EPC page is requested for reservation (e.g., enclave creation and dynamic EPC allocation), the EPC manager checks the number of free EPC pages available. If it is below a threshold, MIN_EPC,[2] the EPC manager launches EPC page eviction. Upon a request for EPC page eviction, the EPC manager searches for an empty Version Array (VA) slot. After obtaining a slot, it finds the target EPC page to evict. To find a suitable EPC page, we use the First In First Out (FIFO) mechanism. The mechanism can be replaced with the Least Recently Used (LRU) or more sophisticated algorithms to reduce the number of potential page faults generated, enhancing overall performance in running SGX applications concurrently. Finding the most suitable mechanism remains a topic for future research.

The EPC page found for eviction is passed as a parameter to EBLOCK to mark the state of the page as blocked in the EPCM entry. The blocked EPC page and VA slot address are then passed as parameters to EWB with the PAGEINFO address. The output of an executing EWB, such as an encrypted page and MAC, will be stored in the buffers designated by the PAGEINFO structure. The addresses of these buffers and the page's inherent metadata, such as its enclave linear address and page protection flag, should be maintained as a data structure to reload the evicted page into the EPC area and ensure that the reloaded page has the same page properties as the evicted page (e.g., the same enclave linear address and protection flag).

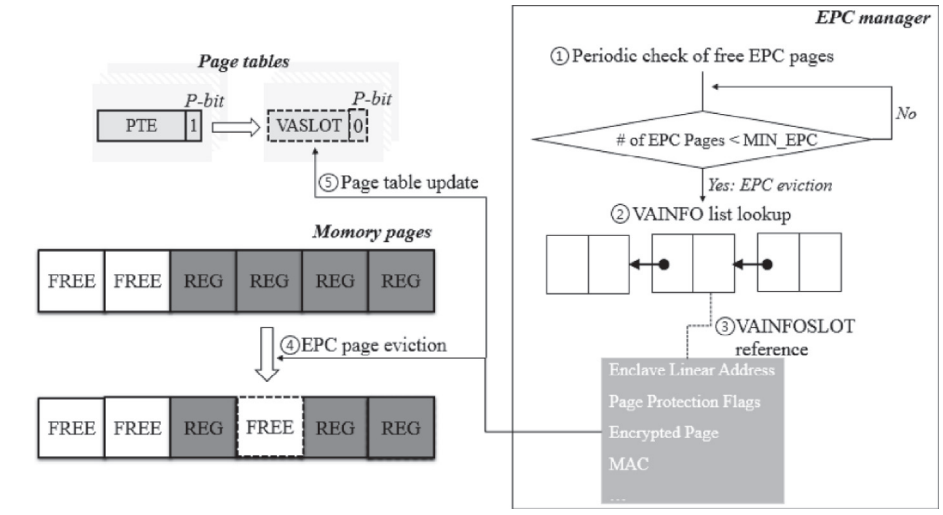
For this purpose, we introduce new system data structures called Version Array Information (VAINFO) and a Version Array Information Slot (VAINFOSLOT) (see Fig. 4). We design each Version Array (VA) slot to have a corresponding VAINFOSLOT, and VAINFOSLOT is used to store all the information needed for reloading the evicted page into the EPC. VAINFO is a linked-list data structure used to find the corresponding VAINFOSLOT when a VA slot address is given. This raises a question as to how this VA slot address can be obtained in page loading. We find the page table entry of the evicted EPC page as a suitable place. After executing EWB, S-OpenSGX uses one VAINFOSLOT and modifies the page table entry of the evicted EPC page so that the entry contains the corresponding VA slot address.[3] This page table entry modification is also not present in the SGX specification, and we infer that the Intel SGX driver (Intel SGX Linux driver) would take a similar approach. Incidentally, the above EPC page eviction procedure is repeated as many times as the specified number NR_EVICT_PAGES to supply enough free EPC pages at once.

The evicted page loading takes opposite steps. When an enclave process accesses an enclave linear address for the evicted page, a page fault is raised because the page table entry for the evicted page is modified to be non-present. The customized page fault handler then checks whether the faulted address obtained through the CR2 register is within the enclave linear address region to ensure that the page loading procedure is only performed for an enclave linear address. If so, through page table walking, the handler obtains a page table entry for the faulted address and retrieves a VA slot address from the entry. The handler then uses the VA slot address to find the appropriate VAINFOSLOT by referring to VAINFO, the head of which is defined as a global variable. With this VAINFOSLOT, the evicted page's inherent metadata can be recovered when the page is reloaded.

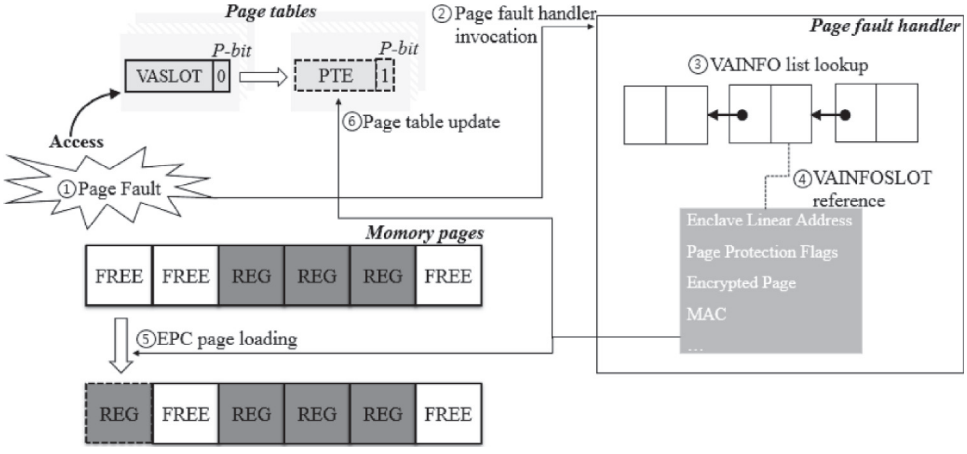
The handler requests a free EPC page from the EPC manager and executes ELD with the free EPC page, VA slot address, and the PAGEINFO that contains all remaining information needed to load the EPC page such as the evicted page, MAC, etc. Once ELD is executed, the free EPC page is filled with the loaded page only if the saved MACs are consistent with the newly calculated MACs based on the properties of the loaded page. As the last step, the page table entry is modified to contain the enclave linear address with the present bit set. Upon success of the customized page fault handler routine, the original page fault

[2] This is the defined minimum number of free EPC pages.

[3] We used two extra bits to mark it as evicted and the existing present bit as "non-present".



(a) EPC page eviction.



(b) EPC page loading.

Fig. 4 – SGX paging in S-OpenSGX.

handler routine is skipped so that accessing the faulted address accesses the loaded EPC page seamlessly. If the faulted address is not within an enclave linear address region, the original page fault handler routine handles the fault instead.

4.3. *Kernel entry point management*

Applications can enter into the OS kernel when they invoke a system call for requesting a system service or events such as interrupts or exceptions are encountered. In this section, we describe how these kernel entry points are managed in S-OpenSGX to support SGX's system functionalities.

4.3.1. *System call handler*

SGX instructions require both privileges: system privileges and user privileges. To execute the SGX instructions of system privileges, called ENCLS, system software should be involved. When a user process needs to request system software to perform some task requiring system privileges, a system call is used as the sole communication medium between a user process and system software. An enclave creation request is one of the representative system calls used in SGX. Because of the absence of a guest OS, OpenSGX cannot insert new system call handlers. Instead, OpenSGX regards invoking several user-level functions as invoking system calls.

S-OpenSGX takes the approach of inserting new system call handlers. Because OSs have system call tables of fixed size, the end of system call entries is not the right place for insertion. To insert SGX system call entries without kernel recompilation, we find and overwrite empty entries in the middle of the system call table to point to our inserted SGX handlers. Utilizing empty entries prevents new system call insertions from influencing other regular system calls. Because the system call table is write-protected, our SGX Module temporarily turns off page table protection to hook the table. After insertion, a user process can directly invoke an SGX system call with the corresponding system call number and parameters.

4.3.2. *Interrupt/exception handler*

When a process running in enclave mode encounters an interrupt/exception event, the process exits from enclave mode through AEX, and the IDT is referenced to find the

corresponding interrupt/exception handler. Each entry in the IDT specifies the location of a stub, and the stub invokes its handler after saving the previous context.

OpenSGX cannot modify interrupt/exception handlers because the IDT, stubs, and handlers are all located in the kernel region. Thus, when a certain event requires a customized interrupt/exception handler, OpenSGX gives up handling the event. Page fault exceptions generated in accessing an evicted EPC page are representative examples requiring a customized handler in SGX. S-OpenSGX can insert a customized handler by modifying interrupt/exception handlers in its guest OS.

In order not to influence or interfere with other normal interrupt/exception events, S-OpenSGX utilizes a hooking technique to preserve original interrupt/exception handlers. For this purpose, S-OpenSGX maintains two handlers: one for enclave events and the other for normal events. After page table protection is temporarily disabled for modifying the write-protected IDT, a target entry in the IDT is modified to point to our stub instead of the original stub. Our stub is the same as the original stub except that our stub invokes a hooking mechanism instead of the original handler. The hook determines which handler to invoke depending on the kind of event: enclave events are handled in our customized handler and normal events are handled in the original handler.

5. Implementation

In this section, we describe the implementation details of SystemSGX, SGX Module, and SGX Runtime. SystemSGX is built based on the QEMU emulator (version 2.1.1) and SGX Module is implemented as a loadable kernel module (LKM). SGX Runtime supports additional user-level functionalities such as multithreading and signals, which are not provided by OpenSGX. Linux version 3.13.11-ckt29 and 3.13.0-91-generic are used as guest and host OSs, respectively.

5.1. *SystemSGX*

SystemSGX, S-OpenSGX's emulator, is implemented based on QEMU. OpenSGX's emulator utilizes QEMU's user-mode emulation while SystemSGX utilizes QEMU's system emulation. In the OpenSGX emulator, all SGX leaf functions are implemented in helper routines, which are in charge of emulating complex guest instructions or events. SystemSGX is designed to make use of OpenSGX's helper routines in emulating SGX leaf functions. However, SystemSGX cannot directly utilize the OpenSGX emulator's implementation because different environments exist in system emulation and user-mode emulation. We face the following challenges in porting existing parts, to which we present solutions.

5.1.1. *Dereferencing problems*

SystemSGX should be able to dereference guest structure data members without crashing. SystemSGX's helper routines utilize host virtual addresses, but the pointer for guest structures (e.g., SECS, TCS, SECINFO, etc.) contain guest virtual addresses. This does not cause any problems in user-mode emulation because guest and host virtual addresses are identical there. However, system emulation treats two addresses differently, and dereferencing data members in the guest structure in helper routines causes a problem. This is because the dereferenced pointer has a guest virtual address while helper routines regard it as a host virtual address. Thus, a guest virtual address is translated with the host's MMU when dereferencing data members in the guest structure in helper routines.

To dereference guest structure data members without encountering problems, we introduce shadow object handling. Shadow object handling first copies the target guest structure to the host's allocated memory. Next, every point where dereferencing a data member in a guest structure occurs is replaced with dereferencing the same data member in a copied host structure. In this case, dereferencing data members in the copied structure does not cause any problems because the copied structure contains a host virtual address and the host's MMU can translate it correctly. Note that address comparison routines are only valid for guest addresses and not for host addresses. Thus, when address comparison is needed, the address of the original guest structure is used.

5.1.2. *Resolved address comparison*

SystemSGX should be able to handle resolved address comparisons. There are several points where SGX leaf functions should reference the EPC's physical address. For example, to check whether a given enclave linear address is resolved within an EPC, the EPC's physical address range should be compared against the resolved address. However, system emulation cannot access guest physical addresses because the guest OS's MMU translates guest virtual addresses into guest physical addresses and there is no routine for accessing a guest OS's MMU in system emulation. Thus, enclave linear addresses cannot be translated into guest physical addresses in helper routines for comparison.

SystemSGX addresses the challenge for the resolved address comparison by utilizing QEMU's soft MMU. QEMU's soft MMU is in charge of translating guest virtual addresses into host virtual addresses. When an enclave linear address needs to be resolved into a guest physical address for comparison, SystemSGX instead resolves it into a host virtual address using QEMU's soft MMU. SystemSGX then compares it against the EPC's host virtual address range. To maintain the host virtual address of an EPC page, an additional field called epcHostAddress is inserted into EPCM entries and is assigned in QEMU initialization. We argue that comparing host virtual addresses has the same effect as comparing guest physical addresses because a guest physical address is mapped one-to-one to a host virtual address and the host virtual address is beyond an attacker's manipulation. This approach can avoid the overhead of maintaining an additional MMU that can translate a guest virtual address into a guest host address.

5.1.3. *Emulating AEX*

In QEMU, `x86_cpu_do_interrupt()` is in charge of handling exceptions while `do_interrupt_x86_hardirq()` is in charge of handling interrupts. Before both of the functions invoke `do_interrupt64()` to emulate hardware handling of interrupts and exceptions (Bovet and Cesati, 2005), we insert enclave-mode checking and AEX calling. If interrupts or exceptions occur

in enclave mode, SystemSGX invokes `helper_sgx_ehandle()`, which emulates the AEX operation as described in the SGX specification (Intel software guard extensions programming reference, 2014).

5.2. SGX module

We implement S-OpenSGX's SGX Module as a Linux Loadable Kernel Module (LKM). The module is built based on OpenSGX's kernel components. As OpenSGX's kernel components run in user mode while those of S-OpenSGX run in privileged mode, the discrepancy between user space and kernel space should be resolved in porting.

5.2.1. Discrepancy adjustment

First, functions available in kernel space are different from those in user space because kernel space cannot utilize the standard C library. We seamlessly replace the standard C library functions with kernel-supported functions. For example, to allocate a dynamic memory region, `malloc()` and `memalign()` are commonly used in OpenSGX. We replace them with either `kmem_cache_alloc()` or `kmalloc()`. To allocate frequently allocated objects efficiently, SGX Module uses a slab allocator's `kmem_cache_alloc()`. For allocating a large amount of memory (e.g., creating an EPC space), SGX Module uses kmalloc. While vmalloc can allocate more memory than kmalloc, we prefer to use kmalloc because it allocates physically contiguous memory blocks. In every memory access, S-OpenSGX's memory protection logic checks whether the accessed address is resolved into the EPC region. With a physically contiguous EPC region, the logic simply compares the resolved address against both ends of the EPC; in the other case, every EPC page's physical address should be compared.

Second, kernel space includes different header files from those in user space. We either replace them or remove the header files unsupported in kernel space.

5.2.2. Page table linking

SGX Module links a range of enclave linear addresses with EPC pages' physical addresses in page table entries when EPC pages are inserted into an enclave. As enclave linear addresses are in the process address space's unused region, they have no page table entries (pud, pmd, and pte). Thus, SGX Module allocates the entries before modification. After allocation, SGX Module conducts page table walking by passing a linear address as an input to find the page table entry of the linear address. For page table walking, SGX Module utilizes a slightly modified version of `follow_pte()` where present bit checking is removed because newly allocated page table entries have no present bit set. Using `set_pte()`, SGX Module modifies the page table entry found to contain the target EPC page's physical address and the corresponding page protection flags. Because `follow_pte()` obtains a lock, SGX Module releases it after modification.

5.2.3. Enclave linear address usage

There are several points where an SGX data structure stores an enclave linear address as its field. For example, SECS's BASEADDR field contains the enclave's base linear address and PAGEINFO's LINADDR field contains the enclave linear address for a target EPC page. OpenSGX does not distinguish enclave linear addresses from EPC pages' virtual addresses. Because this practice is against Intel SGX's specification, we correct all these points to store enclave linear addresses instead of EPC pages' virtual addresses.

5.2.4. Error code returns

System call handlers return an error code so that user processes can grasp the reason(s) for a failure. SGX Module's added system call handlers return different error codes in different failure cases to facilitate debugging of SGX Module's code. For example, if an EPC page cannot be reserved owing to the lack of available EPC pages, our system call handler returns an ENOMEM error code. By assigning this code to a global variable errno, the user process can perceive why the requested system call fails: in this case, "Out of Memory."

5.3. SGX runtime

We implement S-OpenSGX's SGX Runtime based on OpenSGX's user components. Because the user-mode emulation and the system emulation both provide a similar environment for the user context, the code is mostly similar except for the following parts.

5.3.1. Multithreading

To support multithreading in an enclave process, SGX Runtime utilizes POSIX threads, known as pthreads (Posix threads programming). After preparing for thread-specific EPC pages in the kernel, SGX Runtime receives an array of TCS addresses. When multiple threads are created with pthread, SGX Runtime passes this array of TCS addresses to the created threads as an argument. Each thread finds its assigned TCS address in the array by indexing with its thread id and executes EENTER with its TCS address. Finally, the main thread waits for all other created threads to terminate.

5.3.2. System call invocation

Unlike OpenSGX, which regards function calls as system calls, S-OpenSGX directly invokes system calls with syscall instructions inserted with an extended asm. SGX Runtime passes syscall's first three parameters (RDI, RSI, and RDX registers) using input operands in the extended asm. However, because of the restriction in the extended asm, R10, R8, and R9 registers cannot be used directly in input operands while they are required as the next parameters in syscall. SGX Runtime instead receives input in other registers (e.g., RAX, RBX, and RCX) and copies the value of these registers to the target registers using movq instructions. After preparing for syscall parameters, SGX Runtime overwrites RAX as a target system call handler's number. Using this technique, SGX Runtime is able to invoke syscall instructions with up to six parameters.

5.3.3. Signal handling

We insert signal handlers in SGX Runtime to handle exception events properly in enclave mode. A signal is a positive integer that the OS sends to a process to report exceptional behavior or an asynchronous event. When an exception event occurs in a process running in enclave mode, the AEX hardware

routine saves an Asynchronous Exit Point (AEP) into the RIP register. The value of the RIP register is pushed onto the kernel stack, and the OS kernel handles the exception in the corresponding handler. When an IRET instruction is executed at the end of the handler, program control is transferred to the pushed RIP by popping it from the stack. If the signal handler for the event is not registered in the process, the kernel chooses not to execute IRET and instead kills the process. Thus, an AEP that reenters an enclave with ERESUME is not reached.

6. Evaluation

In this section, we evaluate S-OpenSGX with the following experiements: SGX paging, thread isolation, timer interrupt frequency control, and attack model test. The experiments demonstrate S-OpenSGX's system functionalities and hardware modification capability. A brief summary comparing S-OpenSGX with OpenSGX and Intel SGX also follows.

6.1. *SGX paging*

We conducted a paging experiment that demonstrates S-OpenSGX's two system functionalities: SGX paging and page table handling. For this experiment, the number of EPC pages was set as 256 while MIN_EPC and NR_EVICT_PAGES were set as 128 and 16, respectively. To observe SGX paging functionality's effect, we evaluated the number of enclave processes that can run concurrently with and without paging functionality as the enclave size increases. We counted the number of enclave programs successfully executed until "Out of Memory" was displayed, which indicated that the requested EPC slots for a new execution were not available.

Fig. 5 shows that the number of concurrently running enclaves increases with paging. The increase rate grows as enclave size decreases, because the number of SGX leaf functions executed for paging (EWB and ELD) increases as enclave size decreases and further paging is conducted before the enclave processes fill out the EPC area. We also observe that SGX executes more enclave processes with more paging overhead.

6.2. *Thread isolation*

We implemented a thread isolation example to verify our proposed defense mechanism in Section 4.1.3 and to demonstrate S-OpenSGX's CPU and MMU modification capability together with multithreading. The example SGX program consists of two threads: the main thread and input thread. The input thread receives and passes an input message to the main thread while the main thread decrypts the passed input message with its secret key. To simplify the example, we assume that an attacker knows the main thread's secret key address and that the input thread has a buffer overflow vulnerability with Data Execution Prevention (DEP) disabled.

From the perspective of an attacker, we crafted and passed an attack payload to the input thread, which exploited the vulnerability to steal the main thread's secret key. When CPU and MMU were not changed, the designed payload successfully stole the secret key. After our thread isolation was applied, the attack

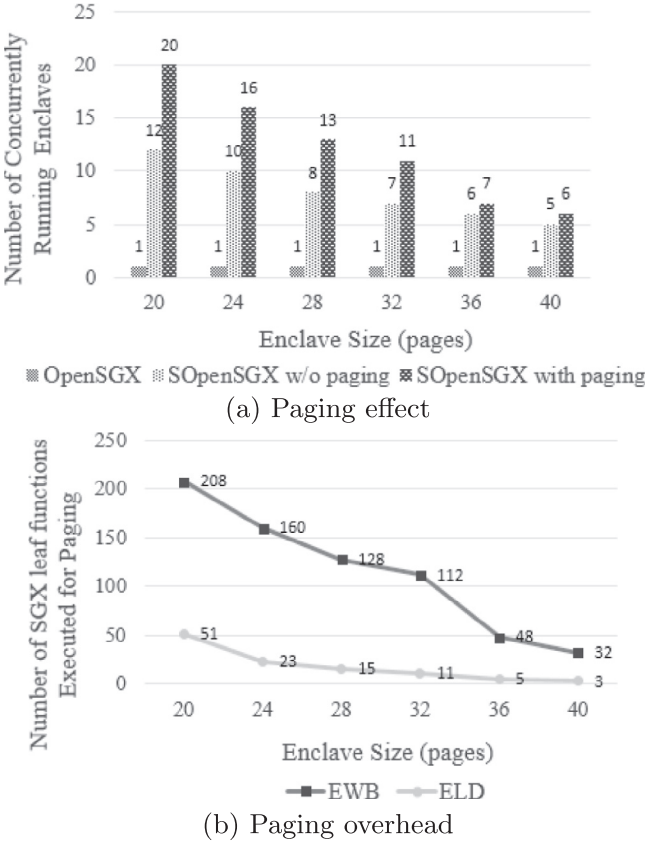


Fig. 5 – Paging effect and overhead.

failed because the input thread's CR_TCS_PH was different from the accessed stack page's ENCLAVETCS, which was owned by the main thread.

6.3. *Timer interrupt frequency control*

To demonstrate S-OpenSGX's device modification capability, we modified an APIC device so that it could retrieve the current value of a CR_ENCLAVE_MODE register from the CPU and control the frequency of timer interrupts when enclave mode was on. This modification can enhance the overall performance of enclave-mode programs because the TLB flush caused by the timer interrupt and the AEX constitutes the main overhead in SGX (Baumann et al., 2014). Note that SGX programs should exit enclave mode and reenter it with the TLB flushed during every timer interrupt event, whereas normal programs can be resumed with the reserved TLB entries for the programs. This additional overhead justifies fewer timer interrupts in enclave mode.

For this experiment, we created a simple SGX program and measured the total number of timer interrupts delivered in enclave mode, which was the same as the number of AEXs. To control the frequency, we imposed a certain length of delay on the arrival time of the next timer interrupt in the APIC device if the interrupt was delivered while the program was running in enclave mode. The results are shown in Table 1. It is clear that the frequency of timer interrupts (AEXs) is decreased by nearly half as the timer delay is doubled. Note that our approach provides a fine-grained frequency control specific for

Table 1 – Timer interrupt frequency in enclave mode with different timer delays.

Timer delay	Default	10M	20M	40M	80M	160M
# of timer interrupts	37	18	10	6	5	2

the enclave whereas the methods of naively controlling the overall frequency can severely compromise the reactivity of normal programs.

6.4. *Attack model test*

To show how S-OpenSGX helps evaluate security problems and enables system-level security experiments, we conducted an attack model test with our synthetic rootkit. The rootkit was implemented as a LKM and designed to leak the secret key of a target program. We tested the rootkit in two scenarios: one with a normal program and the other with an enclave program. For simplicity, target programs print out the location of their secret keys.

The rootkit first disables Supervisor Mode Access Protection (SMAP) to access the application memory with higher privilege. It then finds out by name the process descriptor of the target program in the process descriptor list and temporarily replaces the page table base address (CR3) with the one contained in the found process descriptor. With this replacement, the rootkit can walk the target program's page table to access the secret key.

As a result, the rootkit successfully leaked the secret key of the normal program. However, the rootkit was not able to leak the secret key of the enclave program due to the existence of EPC protection.

6.5. *Comparison*

We compared OpenSGX, S-OpenSGX, and Intel SGX and summarized it in Table 2. In CPU functionality support, OpenSGX, S-OpenSGX, and Intel SGX all support SGX specification revision 1 (Intel software guard extensions programming reference, 2013). Although the SGX specification, revision 2 (Intel software guard extensions programming reference, 2014), is publicly available, the hardware that supports this revision has not been released yet. As emulators, OpenSGX and S-OpenSGX support revision 2 as well.

Lack of system functionalities is one of the fundamental drawbacks in OpenSGX. S-OpenSGX and Intel SGX support all basic system functionalities such as scheduling, multiprocessing, multithreading, page table handling and SGX paging. As thread isolation is our proposed scheme, only S-OpenSGX can currently support it.

In terms of experiments, S-OpenSGX and Intel SGX can freely modify system software, but OpenSGX is restricted because it runs in user space. The existence of kernel space makes S-OpenSGX and Intel SGX favorable for attack model testings such as rootkit experiments. As emulators, OpenSGX and S-OpenSGX have a similar advantage over Intel SGX of being able to modify the CPU and MMU. In addition, S-OpenSGX can insert or modify virtual devices for better utilization of SGX. Because of the limitations of system emulations, S-OpenSGX provides the lowest performance compared to OpenSGX and Intel SGX.

7. Limitation and future work

7.1. *SMP support*

Symmetric multiprocessing (SMP) is the processing of programs by multiple processors sharing main memory and a single OS (Smp (symmetric multiprocessing), 2007). S-OpenSGX assumes that the guest machine has a single core CPU. In system emulation, the SMP can be supported with up to 255 CPUs (Qemu emulator user documentation). We expect that problems similar to thread synchronization would arise from introducing SMP. It remains our future work to extend S-OpenSGX to support SMP.

7.2. *KVM support*

In QEMU's KVM hosting mode, KVM handles the execution of the guest while QEMU emulates devices rather than the CPU (Qemu Wikipedia). KVM-hosting mode is much faster than

Table 2 – Comparison of OpenSGX, S-OpenSGX, and Intel SGX.

Category	Functionality	OpenSGX	S-OpenSGX	Intel SGX
CPU	SGX SPEC v1 support	o	o	o
CPU	SGX SPEC v2 support	o	o	x
System	Scheduling	x	o	o
System	Multiprocessing	x	o	o
System	Multithreading	x	o	o
System	Page table handling	x	o	o
System	SGX paging	x	o	o
System	Thread isolation	x	o	x
Experiment	System S/W modification	Restricted	Flexible	Flexible
Experiment	Attack model test	Restricted	Flexible	Flexible
Experiment	H/W modification (CPU+MMU)	Flexible	Flexible	Restricted
Experiment	H/W modification (peripherals)	Restricted	Flexible	Restricted
Experiment	Performance	Medium	Slow	Fast

system emulation mode because it uses hardware virtualization, where instructions are executed directly by hardware. (Note that dynamic translation is accompanied by emulation.) We design S-OpenSGX to make use of SGX-enabled CPU emulation code in OpenSGX; thus, we opt out of KVM, which is not involved with CPU emulation. However, to enhance the overall performance of S-OpenSGX, we are planning to equip S-OpenSGX with KVM as our future work.

7.3. *Compatibility issues*

S-OpenSGX is built based on OpenSGX; thus, it follows OpenSGX's programming model. Note that Intel's Linux SGX software stack, including SDK and its driver, were not publicly available until the last stage in writing this paper. There are several gaps between S-OpenSGX and Intel's SGX software stack (e.g., APIs and system calls). To allow S-OpenSGX to become more widely adopted, it is one of our primary goals to reduce these gaps and to make S-OpenSGX compatible with Intel's SGX software stack. We believe that not only can we learn some lessons from the software stack's design and code, but that Intel can also learn from those of S-OpenSGX.

Compatibility with real hardware is another issue we must resolve. While S-OpenSGX's emulated machine is closer to a real machine than OpenSGX's, some differences still exist. For example, a real machine sets up an EPC region at boot time in the BIOS while S-OpenSGX does this at runtime in module insertion.

7.4. *Unimplemented functionalities*

In Intel SGX, the Memory Encryption Engine (MEE) protects the EPC against hardware attacks. With the aid of MEE, the contents of the EPC can be stored in encrypted form. S-OpenSGX does not have MEE because a memory controller, to which MEE should be attached, is not emulated in QEMU. By inserting additional helper functions in read/write instructions, we expect that MEE's functionalities can be emulated.

In evicting an EPC page, S-OpenSGX can handle a page of regular type but not a page of other types. This is sufficient for reserving free EPC pages because pages of regular type occupy most of the region in an EPC. However, we plan to support pages of other types to cover corner cases.

8. Discussion

8.1. *Malicious thread*

In this paper, a malicious thread indicates a thread that is compromised by an attacker at runtime via software vulnerabilities such as logic bugs and memory corruption. We note that software vulnerabilities are likely to be exploited when there is any form of interaction between an attacker and the software.

In general, threads are multiple execution flow units inside a single process which share the same virtual address space. However, sharing the virtual address space does not necessarily mean each threads share the security permission levels as well. In fact, threads can have their own software/hardware permission levels. For example, two threads in a same process can have different effective-user-ids in Linux system. Moreover, recent studies such as Shreds (Chen et al. 2016) and SMV (Hsu et al., 2016) proposed system architectures that can separate threads in terms of memory access control. Considering the recent state-of-the-art thread security model, it is important to isolate per-thread components (e.g., TLS, stack, and so forth) from other threads.

Consider a scenario that [thread A] is a piece of code that interacts with possibly a malicious attacker, and [thread B] is a component that does not interact with outside world. In such a case, an attacker cannot attack [thread B] directly since there is no interaction. However, compromising [thread A] also leads to the compromise of [thread B] as well, since threads share the virtual memory address space. For example, compromised [thread A] can not only access private information in [thread B] but also hijack [thread B]'s program control by modifying a return address in [thread B]'s stack. In order to prevent such security breaches, we implemented an experimental thread-isolation feature in the SGX emulation environment, which prevents unnecessary inter-thread memory accesses in an enclave.

8.2. *Deployment issues*

To support thread isolation in Intel SGX, the existing SGX instructions should be slightly modified. Because our modification on SGX instructions only requires changes in the microcode in which SGX instructions are implemented (Costan and Devadas, 2016), they do not require expensive changes in the CPU's circuitry. Note that modern Intel processors provide a microcode update facility (Costan and Devadas, 2016), and thus our proposition can be provided as a microcode update. Adding a new field in every EPCM entry can be easily done, because EPCM is just a data structure contained in a trusted memory (Costan and Devadas, 2016). Lastly, enforcing thread isolation can be implemented in the Page Miss Handler (PMH), where Intel SGX's memory access checks are performed (Costan and Devadas, 2016).

To support timer interrupt frequency control, Local APIC should be able to retrieve the current value of CR_ENCLAVE_MODE register. This may require inserting an additional line between the Local APIC and CPU. Another timer register for maintaining a delay value is also necessary. In an APIC timer, the timer is started by programming its initial-count register (Intel 64 and ia-32 architectures software developers' manual, 2016). When the timer senses enclave mode with the inserted line, the delay value could be added to this initial-count value so that the added value instead of the initial-count value be copied into the current-count register. After the current-count value reaches zero as the count-down begins, a timer interrupt would also be generated.

9. Related work

9.1. *SGX research*

Intel has published a series of white papers explaining each component of Intel SGX. SGX's instructions (McKeen et al., 2013), sealing and attestation (Anati et al., 2013), the memory encryp-

tion engine (Gueron, 2013), epid provisioning (Johnson et al. 2013), dynamic memory allocation (Xing et al., 2016), and dynamic memory management (McKeen et al. 2016) are covered in these publications. Costan and Devadas (2016) assembled publicly available resources regarding Intel SGX and analyzed Intel SGX's security properties in depth from a third-party perspective.

Meanwhile, several works have aimed to enhance the security of SGX. Haven (Baumann et al., 2014) enables unmodified applications to run under SGX with sandbox protection by inserting Drawbridge (Porter et al., 2011) together with the application in an enclave and supporting it with new SGX instructions. Moat (Sinha et al., 2015) provides a tool for formally verifying the confidentiality of an enclave program. The tool called "/confidential" (Sinha et al., 2016) is a successor of Moat and can scale to large programs. Controlled-channel attacks (Xu et al., 2015) reveal that the confidentiality of SGX can be detoured by a side channel based on the page faults that can be controlled by an attacker. Shinde et al. (2016) proposed a mitigation of the controlled-channel attacks by masking page fault patterns.

The expectations of Intel SGX have made it widely applied in various areas. Intel has demonstrated applications of Intel SGX to protecting one-time passwords (OTP), Enterprise Rights Management, and video conferencing (Hoekstra et al., 2013). VC3 has applied SGX to MapReduce jobs in Hadoop for providing SGX's protection for data analytics in the cloud (Schuster et al., 2015). The Town Crier has used SGX's remote attestation in authenticated data feeds for smart contracts (Zhang et al., 2016). Kim et al. (2015) explored adapting diverse network applications to realize the benefits of SGX. Shih et al. (2016) protected the states of Network Function Virtualization (NFV) applications with SGX.

9.2. *Emulator*

QEMU is commonly used in research communities and products. For example, Google's Android emulator is based on QEMU (Running android l developer preview on 64-bit arm qemu, 2014). Groups of teams from Samsung and Linaro have worked to emulate ARM's TrustZone in QEMU (Winter et al., 2012; Arm trustzone in qemu, 2014). QEMU's device emulation is also widely used in virtualization such as KVM and Xen hypervisor.

Several SGX emulators have been proposed and realized. Haven (Baumann et al., 2014) and VC3 (Schuster et al., 2015) have utilized the Intel-provided SGX emulator in their developments and tests. While Intel's emulator is hidden, VC3's own emulator works by hooking an exception handler so that SGX instructions can be emulated while the host OS handles invalid opcode exceptions. OpenSGX (Jain et al., 2016) uses QEMU's dynamic translation to emulate SGX instructions. In addition to the emulator, OpenSGX provides an SGX platform including a software stack for three different modes (user, privileged, and enclave) and versatile tools for compiling, debugging, and performance profiling. While not an emulator, Open-TEE provides a virtual TEE for developing and debugging a trusted application that conforms to the GlobalPlatform (GP) specification (McGillion et al., 2015). Open-TEE maps SGX instructions and events to TEE core APIs so that SGX programs also follow GP standard interfaces (Nyman et al., 2015). However, these previous emulators and virtual platforms are limited to providing instruction emulation or API emulation because a guest OS does not run and devices cannot be emulated.

10. Conclusion

We introduced S-OpenSGX, an open source system SGX emulator, for researching and developing a new SGX design. In contrast to the previous emulator, S-OpenSGX provides indispensable system functionalities such as scheduling, multithreading, and paging, which the previous emulator could not provide because of the limitations of user-mode emulation. With QEMU's system emulation, S-OpenSGX opens a door for modifying not only the CPU and MMU but also peripherals. To show the efficacy of S-OpenSGX, we modified MMU and SGX data structures to implement thread isolation in enclaves. The suggested isolation successfully prevented a compromised thread from leaking sensitive data that belonged to another thread, while not violating nor diminishing SGX's basic security guarantees. In S-OpenSGX, we also modified an APIC device so that the APIC can control the frequency of timer interrupts according to the on/off of enclave mode to reduce SGX's mode switch overhead, which is regarded as the main performance degradation factor in SGX. We hope S-OpenSGX becomes a useful platform for exploring new designs and defense methods based on SGX.

Availability

We have released S-OpenSGX's source code into the public repository. SystemSGX is currently available at https://github.com/CySecLab-KAIST/systemsgx while SGX-Guest is available at https://github.com/CySecLab-KAIST/sgx-guest.

Acknowledgment

The authors would like to thank Taesoo Kim at Georgia Tech and Zhiqiang Lin at UT Dallas for their insightful comments and suggestions. This research is in part based on the work supported by the Software R&D Center, Samsung Electronics. This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2017R1A2B3006360).

REFERENCES

Anati I, Gueron S, Johnson S, Scarlata V. Innovative technology for CPU based attestation and sealing. In: Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy; 2013. p. 10.

Arm trustzone in qemu; 2014. Available from: http://www.linaro.org/blog/core-dump/arm-trustzone-qemu/.

Baumann A, Peinado M, Hunt G. Shielding applications from an untrusted cloud with haven. In: USENIX symposium on operating systems design and implementation (OSDI); 2014. p. 267–83.

Bovet DP, Cesati M. Understanding the Linux kernel. O'Reilly Media, Inc.; 2005.

Chen Y, Reymondjohnson S, Sun Z, Lu L. Shreds: fine-grained execution units with private memory. In: Proceedings of the 37th IEEE symposium on security and privacy. 2016.
Costan V, Devadas S. Intel SGX explained, Tech. rep., Cryptology ePrint Archive, Report 2016/086; 2016. Available from: https://eprint.iacr.org/2016/086.
Gueron S. A memory encryption engine suitable for general purpose processors. 2013.
Hoekstra M, Lal R, Pappachan P, Phegade V, Del Cuvillo J. Using innovative instructions to create trustworthy software solutions. In: Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy. ACM; 2013. p. 11.
Hsu TC-H, Hoffman K, Eugster P, Payer M. Enforcing least privilege memory views for multithreaded applications. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. ACM; 2016. p. 393–405.
Installing and using qemu user-mode emulation. Available from: https://www.ibm.com/support/knowledgecenter/linuxonibm/liaal/liaalqemuemulate.htm.
Intel 64 and ia-32 architectures software developers' manual; 2016. Available from: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.
Intel SGX faq. Available from: https://software.intel.com/en-us/sgx-sdk/faq.
Intel SGX for dummies (Intel SGX design objectives). Available from: https://software.intel.com/en-us/blogs/2013/09/26/protecting-application-secrets-with-intel-sgx.
Intel SGX for Linux. Available from: https://github.com/01org/linux-sgx.
Intel SGX Linux driver. Available from: https://github.com/01org/linux-sgx-driver.
Intel software guard extensions (Intel SGX) sdk. Available from: https://software.intel.com/en-us/sgx-sdk.
Intel software guard extensions programming reference; 2013. https://software.intel.com/sites/default/files/329298-001.pdf.
Intel software guard extensions programming reference; 2014. Available from: https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.
Isca 2015 tutorial slides for Intel SGX; 2015. Available from: https://software.intel.com/sites/default/files/332680-002.pdf.
Jain P, Desai S, Kim S, Shih M-W, Lee J, Choi C, et al. OpenSGX: an open platform for SGX research. In: Proceedings of the network and distributed system security symposium. San Diego (CA). 2016.
Johnson S, Scarlata V, Rozas C, Brickell E, McKeen F. Intel software guard extensions: Epid provisioning and attestation services. 2013.
Kim S, Shin Y, Ha J, Kim T, Han D. A first step towards leveraging commodity trusted execution environments for network applications. In: Proceedings of the 14th ACM workshop on hot topics in networks (HotNets). Philadelphia (PA): 2015.
McGillion B, Dettenborn T, Nyman T, Asokan N. Open-tee-an open virtual trusted execution environment. arXiv preprint arXiv:1506.07367. 2015.
McKeen F, Alexandrovich I, Anati I, Caspi D, Johnson S, Leslie-Hurd R, et al., Intel software guard extensions (Intel SGX) support for dynamic memory management inside an enclave. 2016.
McKeen F, Alexandrovich I, Berenzon A, Rozas CV, Shafi H, Shanbhogue V, et al. Innovative instructions and software model for isolated execution. In: Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy. ACM; 2013. p. 1.
Nyman T, McGillion B, Asokan N. On making emerging trusted execution environments accessible to developers. In: Trust and trustworthy computing. Springer; 2015. p. 58–67.
Operating system – process scheduling. Available from: http://www.tutorialspoint.com/operating_system/os_process_scheduling.htm.
Paging Wikipedia. Available from: https://en.wikipedia.org/wiki/Paging#cite_note-1.
Porter DE, Boyd-Wickizer S, Howell J, Olinsky R, Hunt GC. Rethinking the library OS from the top down. In: ACM SIGPLAN notices, vol. 46. ACM; 2011. p. 291–304.
Posix threads programming. Available from: https://computing.llnl.gov/tutorials/pthreads/.
Qemu emulator user documentation; Available from: http://wiki.qemu.org/download/qemu-doc.html.
Qemu internals. Available from: https://qemu.weilnetz.de/qemu-tech.html.
Qemu open source processor emulator. Available from: http://wiki.qemu.org/Main_Page.
Qemu usermode, howto. Available from: http://nairobi-embedded.org/qemu_usermode.html.
Qemu: user mode emulation and full system emulation; 2015. Available from: http://www.cnblogs.com/pengdonglin137/p/5020143.html.
Qemu Wikipedia. Available from: https://en.wikipedia.org/wiki/QEMU#cite_note-3.
Running android l developer preview on 64-bit arm qemu; 2014. Available from: http://www.linaro.org/blog/core-dump/running-64bit-android-l-qemu/.
Schuster F, Costa M, Fournet C, Gkantsidis C, Peinado M, Mainar-Ruiz G, et al. Vc3: Trustworthy data analytics in the cloud using SGX. In: Proceedings of the 36th IEEE symposium on Security and Privacy, S&P. 2015.
Shih M-W, Kumar M, Kim T, Gavrilovska A. S-NFV: Securing NFV states by using SGX. In: Proceedings of the 2016 ACM international workshop on Security in Software Defined Networks & Network Function Virtualization, SDN-NFV security '16. New York (NY): ACM; 2016. p. 45–8.
Shinde S, Chua ZL, Narayanan V, Saxena P. Preventing page faults from telling your secrets. In: Proceedings of the 11th ACM on Asia conference on computer and communications security. ACM; 2016. p. 317–28.
Sinha R, Rajamani S, Seshia S, Vaswani K. Moat: verifying confidentiality of enclave programs. In: Proceedings of the 22nd ACM SIGSAC conference on computer and communications security. ACM; 2015. p. 1169–84.
Sinha R, Costa M, Lal A, Lopes NP, Rajamani S, Seshia SA, et al. A design and verification methodology for secure isolated regions. In: Proceedings of the 37th ACM SIGPLAN conference on programming language design and implementation. ACM; 2016. p. 665–81.
Smp (symmetric multiprocessing); 2007. Available from: http://searchdatacenter.techtarget.com/definition/SMP.
Thread (computing) Wikipedia. Available from: https://en.wikipedia.org/wiki/Thread_(computing)#Multithreading.
Winter J, Wiegele P, Pirker M, Tögl R. A flexible software development and emulation framework for arm trustzone. In: Trusted Systems. Springer; 2012. p. 1–15.
Xing BC, Shanahan M, Leslie-Hurd R. Intel software guard extensions (Intel sgx) software support for dynamic memory allocation inside an enclave. 2016.
Xu Y, Cui W, Peinado M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In: 2015 IEEE symposium on security and privacy, SP 2015, San Jose, CA, USA, May 17–21; 2015.
Zhang F, Cecchetti E, Croman K, Juels A, Shi E. Town crier: an authenticated data feed for smart contracts, Cryptology ePrint Archive, Report 2016/168; 2016. Available from: http://eprint.iacr.org/2016/168.

Changho Choi received his B.S. degree in Computer Science and Electrical Engineering from Handong Global University in 2012. He also received his M.S. in the Graduate School of Information Security at Korea Advanced Institute of Science and Technology (KAIST) in 2014. He is currently working toward his Ph.D. degree at the Division of Computer Science, KAIST. His research interest includes system security, and trusted execution environments especially in Intel SGX.

Nohyun Kwak received his B.S. degree in Computer Science and Engineering from Pohang University of Science and Technology (POSTECH), South Korea, in 2002. He also received his M.S. degree in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 2005. He is currently working toward his Ph.D. degree at the Division of Computer Science, KAIST. His research interest includes system security, especially in Intel SGX.

Jinsoo Jang received his B.S. degree in Information and Computer Engineering from Ajou University, Korea, in 2007. He also received his M.S. degree in Information Security from Korea Advanced of Science and Technology (KAIST) in 2014. He is currently working toward his Ph.D. degree at the Division of Computer Science, KAIST. His research interest includes system security, especially in the trusted execution environment (TEE).

Daehee Jang received his B.S. degree in Computer Engineering from Hanyang University, South Korea, in 2012. He also received his M.S. degree in Information Security from Korea Advanced Institute of Science and Technology (KAIST) in 2014. He is currently working toward his Ph.D. degree at the Division of Computer Science, KAIST. His research interest includes software vulnerability, operating system, Intel SGX.

Kuenwhee Oh received his B.E. degree in the Division of Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 2014. He is currently working toward his M.S. degree at the Division of Computer Science, KAIST. His research interest includes system security and application of Intel SGX.

Kyungsoo Kwag received his B.S degree in Computer Science from Sogang University, Seoul, Korea, in 2004. Currently, he works for Samsung Electronics as Senior Engineer. His research interests are security issues such as trusted execution environment and malicious application detection.

Brent Byunghoon Kang is currently an associate professor at the Graduate School of Information Security at Korea Advanced Institute of Science and Technology (KAIST). Before KAIST, he has been with George Mason University as an associate professor. Dr. Kang received his Ph.D. in Computer Science from the University of California at Berkeley, and M.S. from the University of Maryland at College Park, and B.S. from Seoul National University. He has been working on systems security area including botnet defense, OS kernel integrity monitors, trusted execution environment, and hardware-assisted security. He is currently a member of the IEEE, the USENIX and the ACM.