OpenSGX: An Open Platform for SGX Research

Prerit Jain† Soham Desai† Seongmin Kim⋆ Ming-Wei Shih† JaeHyuk Lee⋆
Changho Choi⋆ Youjung Shin⋆ Taesoo Kim† Brent Byunghoon Kang⋆ Dongsu Han⋆

†*Georgia Institute of Technology* ⋆*KAIST*
†{pjain43, sdesai1, mingwei.shih, taesoo}@gatech.edu
⋆{dallas1004, jhl9105, zpzigi, yj_shin, dongsuh, brentkang}@kaist.ac.kr

***Abstract*—Hardware technologies for trusted computing, or trusted execution environments (TEEs), have rapidly matured over the last decade. In fact, TEEs are at the brink of widespread commoditization with the recent introduction of Intel Software Guard Extensions (Intel SGX). Despite such rapid development of TEE, software technologies for TEE significantly lag behind their hardware counterpart, and currently only a select group of researchers have the privilege of accessing this technology. To address this problem, we develop an open source platform, called OpenSGX, that emulates Intel SGX hardware components at the instruction level and provides new system software components necessarily required for full TEE exploration. We expect that the OpenSGX framework can serve as an open platform for SGX research, with the following contributions. First, we develop a fully functional, instruction-compatible emulator of Intel SGX for enabling the exploration of software/hardware design space, and development of enclave programs. OpenSGX provides a platform for SGX development, meaning that it provides not just emulation but also operating system components, an enclave program loader/packager, an OpenSGX user library, debugging, and performance monitoring. Second, to show OpenSGX's use cases, we applied OpenSGX to protect sensitive information (e.g., directory) of Tor nodes and evaluated their potential performance impacts. Therefore, we believe OpenSGX has great potential for broader communities to spark new research on soon-to-be-commodity Intel SGX.**

I. Introduction

Hardware technologies for trusted computing, so called trusted execution environments (TEEs), have rapidly matured over the last decade [3, 18]. Trusted execution environments are at the brink of widespread commoditization with the recent introduction of Intel Software Guard Extensions (Intel SGX) [2, 19, 36]. Intel SGX allows an application, or its sub-component, to run inside an isolated execution environment, called an enclave. Intel SGX hardware protects the enclave against any malicious software, including operating system, hypervisor, and low-level firmware (e.g., SMM), which attempts to compromise its integrity or steal its secrecy. With the widespread adoption of cloud computing, the speculation is that Intel SGX can be a vehicle for enabling secure cloud computing and allowing many unforeseen security applications.

NDSS '16, 21-24 February 2016, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2016.23011

The adoption of Intel SGX can have a dramatic impact on software design and implementation. For example, the introduction of SGX may require new programming models or even a new paradigm to be considered. However, despite the rapid development of TEEs, software technologies for TEE are still at a nascent stage. In fact, for Intel SGX, it is not feasible to fully explore all the potential that the SGX can provide because the research community lacks any usable platform for experimentation. In particular, access to the Intel SGX platform is currently limited to only a select group of people [5, 42, 43]. This is one of the fundamental barriers to innovation and software research on SGX, especially at its early phase.

In this paper, we attempt to address this fundamental issue of designing and implementing a basic infrastructure that allows full instrumentation and exploration of SGX research. To this end, we proposed and developed an open platform, called OpenSGX, that emulates Intel SGX at the instruction-level by extending an open-source emulator, QEMU. In particular, we leverage QEMU's userspace binary translation to implement SGX instructions. However, OpenSGX is not just an SGX instruction emulator, but it serves as a complete platform that includes an emulated operating system layer and services, an enclave program loader/package, a user library, debugging support, and performance monitoring. Because OpenSGX is implemented purely in software, it allows easy instrumentation in virtually all aspects of SGX, such as hardware components and system software, as well as memory encryption schemes. In addition to its use for research, OpenSGX can be used for developing SGX applications, as it is a self-contained platform, which provides isolated execution and remote attestation.

However, it is not straightforward to design and implement such an open platform for both research and development communities. Not only does it require an intensive amount of engineering effort, but it also needs to be designed to inspire new research opportunities in operating systems and applications. In particular, we find that while the Intel SGX specification describes in detail the instruction set and internal data structures, it leaves other important components largely unspecified, such as support for system software and application programming interface. This raises a number of non-trivial issues. For example, many of the Intel SGX instructions are ring 0 instructions that require kernel privilege. This implies that an operating system, an untrusted entity with respect to SGX, must be involved to provide service (e.g., through system calls). Thus, a secure SGX design is required for SGX enclave applications to defend against potential attack vectors, such as Iago attacks [11].

Furthermore, SGX requires that application code and data be placed on Enclave Page Cache (EPC), a reserved, encrypted area of memory, and that its execution must stay within EPC. For executing a binary on EPC, an SGX instruction can allow one to copy a normal page onto an EPC page; however, a dynamic loader is additionally required to supply the provisioning of the code, data, and stack sections on EPC (e.g., relocation). To provide an ecosystem, OpenSGX must address these issues, while the SGX specification largely concerns the instruction set and low-level interfaces.

OpenSGX design fills this gap to provide necessary support for SGX application programmers to readily implement their TEE applications and explore the feasibility. In particular, OpenSGX provides six components to support all aspects of SGX development: hardware emulation module, operating system emulation, enclave loader, user library, debugging support, and performance monitoring. To evaluate all components of the system and demonstrate the potential of OpenSGX, we conducted a case study using Tor, the anonymity network, as a concrete example. We discuss how SGX might be used to enhance the security and privacy guarantees of Tor, redesign Tor to leverage SGX, and use OpenSGX to implement its SGX-based Tor design. Finally, we present the performance profiling result of the SGX-enabled Tor. The profiling result was produced using one of the services provided by OpenSGX.

In summary, we make the following contributions:

- The first open platform for SGX research and development, which includes a wide range of emulation components and toolchain.
- An initial exploration of system support, its interface design, and the security issues involving system calls and user library for SGX programming.
- We applied OpenSGX to Tor nodes to isolate sensitive information (e.g., a signing key of the directory server) and evaluate its potential performance implications.

We find that it is timely to introduce OpenSGX to the community, considering the early-stage of Intel SGX—the first SGX-equipped CPUs (S-Spec: SR2L0, SR2L1, ..., SR2L9, SR2LC, ..., SR2LH, SR2LJ, ..., SR2LN) have been on the market since October 26, 2015 [23], but there are no known motherboards that support SGX other than a few DELL Inspiron laptops (i3153, i3158, i7353, i7359, i7568) as of December 2015. We believe the open research opportunities that OpenSGX brings will help not only the software community in exploring new interfaces and semantics for SGX programming, but also the hardware community in testing and implementing new requirements and services for TEEs that the software community may be able to identify and suggest as new hardware features.

This paper is organized as follows. §II provides background of Intel SGX. §III and §IV describe the system design. §V explains the implementation details. §VI and §VII evaluate OpenSGX through concrete case studies and describe our initial experience of redesigning Tor to adopt Intel SGX. §VIII presents related work, §IX discusses remaining issues, and finally, §X concludes our work.

P	Type	Instruction	Description	V	S
P	MEM	EADD	Add a page	r1	✓
P	MEM	EBLOCK	Block an EPC page	r1	✓
P	EXE	ECREATE	Create an enclave	r1	✓
P	DBG	EDBGRD	Read data by debugger	r1	-
P	DBG	EDBGWR	Write data by debugger	r1	-
P	MEM	EEXTEND	Extend EPC page measurement	r1	✓
P	EXE	EINIT	Initialize an enclave	r1	✓
P	MEM	ELDB	Load an EPC page as blocked	r1	✓
P	MEM	ELDU	Load an EPC page as unblocked	r1	✓
P	SEC	EPA	Add a version array	r1	✓
P	MEM	EREMOVE	Remove a page from EPC	r1	✓
P	MEM	ETRACK	Activate EBLOCK checks	r1	-
P	MEM	EWB	Write back/invalidate an EPC page	r1	✓
P	MEM	EAUG	Allocate a page to an existing enclave	r2	✓
P	SEC	EMODPR	Restrict page permissions	r2	✓
P	EXE	EMODT	Change the type of an EPC page	r2	✓
U	EXE	EENTER	Enter an enclave	r1	✓
U	EXE	EEXIT	Exit an enclave	r1	✓
U	SEC	EGETKEY	Create a cryptographic key	r1	✓
U	SEC	EREPORT	Create a cryptographic report	r1	✓
U	EXE	ERESUME	Re-enter an enclave	r1	✓
U	MEM	EACCEPT	Accept changes to a page	r2	✓
U	SEC	EMODPE	Enhance access rights	r2	✓
U	MEM	EACCEPTCOPY	Copy a page to a new location	r2	✓

TABLE I: Intel SGX Instruction support in OpenSGX. P: Privileged (ring 0) instructions; U: User-level (ring-3) instructions; V: Version; S: Supported by OpenSGX; r1: Revision 1 [21]; r2: Revision 2 [22]; MEM: Memory management related; EXE: Enclave execution related; SEC: Security or permissions related.

Instruction		Description	S
EPCM	Enclave Page Cache Map	Meta-data of an EPC page	✓
SECS	Enclave Control Structure	Meta-data of an enclave	✓
TCS	Thread Control Structure	Meta-data of a single thread	✓
SSA	State Save Area	Used to save processor state	✓
PageInfo	Page Information	Used for EPC-management	✓
SECINFO	Security Information	Meta-data of an enclave page	✓
PCMD	Paging Crypto MetaData	Used to track a page-out page	✓
SIGSTRUCT	Enclave Signature Structure	Enclave certificate	✓
EINITTOKEN	EINIT Token Structure	Used to validate the enclave	✓
REPORT	Report Structure	Return structure of EREPORT	✓
TARGETINFO	Report Target Info	Parameter for EREPORT	✓
KEYREQUEST	Key Request	Parameter for EGETKEY	✓
VA	Version Array	Version for evicted EPC pages	✓

TABLE II: Intel SGX Data Structure implemented in OpenSGX, marked ✓in S if supported.

II. Background

A. Intel SGX

Intel SGX is an extension to the x86 instruction set architecture that enables an application to instantiate a protected container, called an enclave, containing code and data. The memory region residing in the enclave is protected against all external software access including privileged ones such as operating system and the hypervisor. To support enclave, SGX consists of a set of new instructions and memory access changes. Also, SGX supports remote attestation and sealing that allow remotely verifying an enclave and securely saving enclave data in non-volatile memory for future use, respectively.

SGX memory protection. When the processor accesses enclave data, it automatically transfers to a new CPU mode, called *enclave mode*. The enclave mode enforces additional hardware checks on every single memory access, such that only code inside the enclave can access its own enclave region. That is, memory accesses from both non-enclaves and different enclaves are prohibited. Note that memory access policy on the non-enclave regions remains the same, i.e., traditional page walk is performed for both accesses from non-enclaves and enclaves to non-enclave memory.

The enclave data is stored in a reserved memory region

called Enclave Page Cache (EPC). To defend against known memory attacks such as memory snooping, memory content in EPC is encrypted by the Memory Encryption Engine (MEE). The memory content in EPC is only decrypted when entering the CPU package, where the code and data are protected by the enclave mode, and then re-encrypted when leaving to the EPC memory region.

Instruction Set Architecture (ISA). SGX introduces a set of instructions and data structures to support enclave and EPC-related operations (see Table I and Table II). Instructions are classified into user-level instructions (ring 3) and privileged instructions (ring 0). Note that the family of user-lever/privileged instructions is called `ENCLU`/`ENCLS`. For example, the user-level instruction `EENTER` allows the host program to transfer the program's control to an existing enclave program, while `ECREATE` is a privileged instruction that allocates available EPC pages for a new enclave.

B. OpenSGX Specification

One might imagine that faithfully implementing the Intel SGX specification is sufficient for producing a usable emulation and development environment. However, the specification leaves system software including operating system support, debugging, and toolchains for software development, largely under-specified. For example, many SGX instructions (see Table I) require kernel privilege (ring 0), but system call interface and operating system service/support for SGX have not been explored. The system call interface is critical for SGX applications because they must rely on necessary support from an operating system that they do not trust. To fill this gap, we explore system support for OpenSGX application developers and define an interface with the operating system within the SGX OS emulation layer, which provides service to OpenSGX applications.

Disclaimers and threat model. OpenSGX does not support binary-compatibility with Intel SGX because no specification or standardization exists for the binary-level interoperability[1]. Although OpenSGX supports most instructions specified, we do not implement all instructions. Specifically, OpenSGX does not implement debugging instructions, as our software layer can provide a rich environment for debugging (e.g., familiar GDB stub). OpenSGX is a software emulator and provides no security guarantees. Its security guarantees are *not* at all the same level as Intel SGX.

However, we consider the same threat model of Intel SGX in designing the emulation platform. In particular, as in Haven [5], we assume an adversary who has control over all software components, including the operating system and hypervisor, and hardware except the CPU package. In the design and implementation of the system call interface, OpenSGX considers mechanisms to thwart attacks that can be mounted by system software, such as the Iago attacks [11]. For example, we integrate `EAUG`/`EACCEPT` instructions into the dynamic memory allocation API to perform validation on newly allocated memory, which a malicious OS cannot simply bypass. However, protection against denial-of-service is out of scope; an adversary can still launch a denial-of-service attack on SGX [36]. Finally, OpenSGX cannot provide accurate (i.e., wall-clock) performance measures because it is a software emulator. Instead, OpenSGX helps developers and researchers to speculate on potential performance issues by providing its emulated performance statistics similar to that of the `perf` counter.

[1]Recent Windows 10 has been reported to have a preliminary support, SGX R1, as described in a technical report [25]

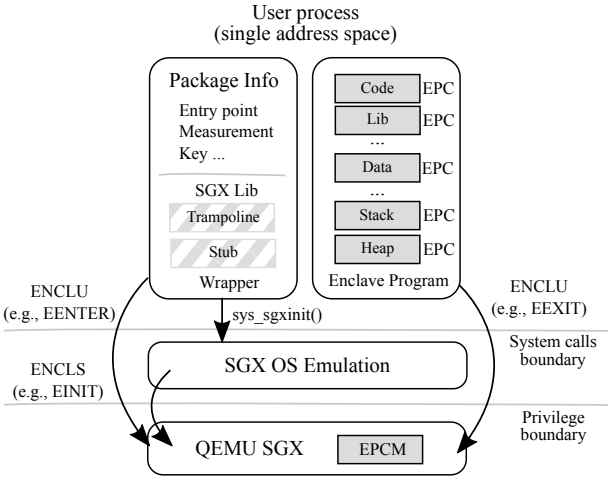


Fig. 1: Overview of OpenSGX's design and memory state of an active enclave program: A packaged program, marked as *Wrapper* and *Enclave Program* together, runs as a single process in the same virtual address space. Since Intel SGX uses privilege instructions to initialize and setup enclaves, OpenSGX introduces a set of system calls to service the requests from the Wrapper program. The grayed boxes represent isolated enclave pages, and stripped boxes depict the shared memory region used to service the enclave program (note this interface is not specified in Intel SGX [21, 22]).

III. System Overview

OpenSGX emulates the hardware components of Intel SGX and its ecosystem, which includes operating system interfaces and user library, in order to run enclave programs. In this section, we describe a high-level overview of OpenSGX's design and lifecycle of an enclave program starting from its development.

A. OpenSGX Components

To implement Intel SGX instructions and emulate its hardware components, we leverage QEMU. In particular, we implement OpenSGX's OS emulation layer and hardware emulation on top of QEMU's user-mode emulation by extending the binary translation. Figure 1 illustrates the overall design components. OpenSGX consists of six components that work together to provide a fully functional SGX development environment. Each component is summarized below and the detail is explained in §IV.

- **Emulated Intel SGX hardware**: We implement hardware components, including SGX instructions, SGX data structures, EPC and its access protection, and the SGX *processor key* as software within QEMU (§IV-A). Note that these components are actually part of the processor or reside in the EPC as part of protected data structures. OpenSGX provides a tool for specifying hardware configurations, such as the size of EPC and the SGX *processor key*.
- **OS emulation**: Some SGX instructions (`ENCLS`) are privileged instructions that should be executed by the kernel. Thus, we define new system calls that the emulated enclave programs use to perform SGX operations, such as enclave provisioning and dynamic memory allocation (§IV-B). Note

that this interface is crucial for development, but is not defined in Intel SGX specifications. Our interface allows user programs to easily use Intel SGX features. The current version supports only user-mode emulation, but we plan to extend OpenSGX for system-wide emulation as well.

- **Enclave program loader**: To launch an enclave program, code and data sections to be executed inside the enclave must be properly loaded to EPC, beyond the measurement and attestation. `ENCLS` instructions allow provisioning of enclave on an EPC page-by-page basis. Our OpenSGX loader automatically takes care of the process by loading the enclave code and data sections into EPC and provisioning an enclave heap and stack regions (§IV-C).
- **OpenSGX user library (`sgxlib`)**: OpenSGX provides a set of useful functions, (`sgxlib`), that can be used inside and outside the enclave (§IV-D). It provides wrappers for all SGX user-level instructions, as well as high-level APIs—for example, `sgx_host_read/write()` for copying data from and to an enclave program. The in-enclave APIs that `sgxlib` provides have been designed to thwart known attack vectors such as Iago attacks and more generally, deputy attacks.
- **Debugger support**: OpenSGX naturally allows easy debugging of the emulation platform due to the nature of software-based emulation. However, for SGX applications, QEMU's binary translation can make debugging more difficult because a debugger can only observe translated instructions. Thus, we extend `gdb` to map to the instruction being emulated. We also expose key SGX data structures (e.g., EPCM) through gdb commands (§IV-E).
- **Performance monitoring**: Finally, OpenSGX supports performance counters/profiler and allows users to collect performance statistics (e.g., the number of enclave-to-host context switches) about enclave programs (§IV-F).

B. Development Lifecycle with OpenSGX

OpenSGX provides a rich development environment, allowing the research community to easily emulate a program running inside an enclave. In this section, we show the development lifecycle of an enclave program and highlight potential research opportunities that reside at each point of the cycle.

Lifecycle 1: Development. Developing an enclave program in OpenSGX is as simple as developing an ordinary program written in `C` language, using our convenience library, `sgxlib`.

Figure 2 shows one of the simplest enclave programs. Developers can solely develop a `C` program to be executed inside an enclave, similar to `hello.c`. We use `wrapper.c` to demonstrate how a developer can launch and execute an enclave program by using the APIs provided by `sgxlib`. Note that `enclave_main()` is merely a default entry point that is loaded to the enclave in this example. Using the section attributes and `sgxlib` APIs, programmers can also specify additional code and data sections to be loaded in the enclave or create multiple enclaves. Also, `sgxlib` defines specific APIs to support host-enclave communication.

The clear, easy-to-use programming model and a convenience library allow developers to easily create an enclave program. The separation of enclave code and data sections in the code also helps them to separate security-sensitive pieces of application code (PAL) [34] for isolated execution.

Lifecycle 2: Launch. OpenSGX provides a toolchain, `opensgx`, to compile the code into an OpenSGX-aware binary (a `.sgx` file) linked to `sgxlib` (see Figure 2). It also generates a configuration file (a .conf file) that contains the program measurement (a hash value), signature signed by a specified RSA key, and other enclave properties that are required to validate the program during the enclave initialization.

To execute the enclave program, OpenSGX performs the following tasks: (1) performs the bootstrapping process via `sgx_init()`; (2) leverages OpenSGX loader API with program information obtained from ELF files (e.g., offset and size of code and data sections) to pre-load the program into allocated memory; (3) initiates enclave initialization by using `init_enclave()`; (4) Once the enclave initialization is done, the host program transfers the program's control to the enclave program via `sgx_enter()`, which is the wrapper of `EENTER`.

Lifecycle 3: Execution. OpenSGX enforces the SGX memory access policy on the enclave program by interposing every single memory access through QEMU's binary translation. Any access violation, such as memory write from non-enclave memory to enclave memory, results in a general protection fault.

Since an enclave program can legitimately access its host memory, any such access can open new attack surfaces to the enclave. To minimize the attack surfaces, `sgxlib` provides substitute APIs that allow an enclave program to avoid the use of a shared library that resides in the host memory. OpenSGX also provides a stricter form of communication protocol by using shared code and data memory, called *trampoline* and *stub*, respectively. After completing tasks in *enclave mode*, the enclave program exits this mode with the help of `EEXIT` instruction, thereby returning the program's control back to the location right after the `EENTER` instruction. Note that it is an operating system's responsibility to reclaim the EPC pages (i.e., `EREMOVE`), perhaps when its wrapper process terminates. However, OpenSGX leaves this clean-up routine unimplemented, as only one enclave runs on an OpenSGX's instance.

Lifecycle 4: Debugging and performance monitoring. OpenSGX supports debugging and performance monitoring. Although OpenSGX is not a cycle-accurate emulator, it exposes a system call to query the OpenSGX emulator about statistics, such as the number of context switches that occurred and SGX instructions executed. SGX researchers can utilize this to estimate the performance implications of an enclave program.

IV. OpenSGX Design

In this section, we provide OpenSGX's design in various levels of abstractions, ranging from the hardware emulation to a user-level library and utilities.

A. Hardware Emulation

OpenSGX emulates the hardware specification of Intel SGX by leveraging the dynamic binary translation of QEMU. OpenSGX enables instruction-to-instruction compatibility; achieving binary-to-binary compatibility is presently not possible because ABI is not officially specified by any entity, neither by OS nor hardware vendors, yet.

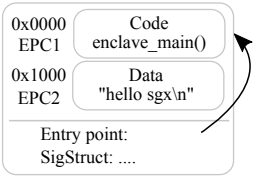
Instruction Set Architecture (ISA). OpenSGX supports instruction-level compatibility (revision 1 and 2 of Intel SGX) to SGX-aware binaries by implementing the core Intel SGX instructions. OpenSGX also provides simple `C` APIs that directly

```c
/* wrapper.c */
#include <sgx-lib.h>

int main(int argc, char **argv)
{
  sgx_init();
  /*
   * Take section information from ELF file as input
   * and compute entry_offset, program_size (code + data).
   */
  ...
  char *base = OpenSGX_loader("hello.sgx",
                              binary_size,
                              section_offset,
                              program_size);
  tcs_t *tcs = init_enclave(base,
                            entry,
                            program_size,
                            "hello.conf");
  sgx_enter(tcs, exception_handler);

  char *buf = malloc(11);
  sgx_host_read(buf, 11);
  printf("%s", buf);

  return 0;
}
```



Packaged Enclave Program

```c
/* hello.c */
#include <sgx-lib.h>

void enclave_main()
{
    char *hello = "hello sgx!\n";
    sgx_enclave_write(hello, sgx_strlen(hello));
    sgx_exit(NULL);
}
```

```
$ opensgx -k
> generate sign.key
$ opensgx -c hello.c
> generate hello.sgx executable
$ opensgx -s hello.sgx --key sign.key
> generate hello.conf
$ opensgx hello.sgx hello.conf
hello sgx!
```

Fig. 2: Example code snippet: "hello sgx!"

wrap the assembly code for each SGX instruction. It is worthwhile to mention that there are two types of instructions in Intel SGX depending on the required privilege: user privilege for `ENCLU` leaf instructions and super privilege for `ENCLS` leaf instructions (see Figure 1). Accordingly, user-level instructions are accessible to a user-level library, called `sgxlib` (see §IV-D), and super-level instructions are only accessible by the OS emulation layer of OpenSGX (see §IV-B).

To be clear, we currently do not support instructions for debugging (e.g., `EDBGRD`). However, as OpenSGX is a software emulator, it provides a better debugging interface (e.g., reading or writing enclave memory regions) and greater flexibility. Also, OpenSGX does not implement paging (e.g., features related to maintaining page tables), as it utilizes user-level dynamic translation. This design decision is intentional; otherwise, OpenSGX users must install a custom operating system to run an enclave program, making it cumbersome and inconvenient.

Enclave Page Cache (EPC). OpenSGX takes advantage of QEMU's user-mode emulation for EPC management. To emulate EPC, we set aside a contiguous memory region that represents the physical EPC available in the system, the same address space of the emulated process. In Intel SGX, the EPC region is configurable via BIOS at boot time by using the Processor Reserved Memory (PRM) [21, 22]. Similarly, to reserve an EPC area in the QEMU user-mode, OpenSGX introduces a custom bootstrapping mechanism. Upon the initialization of an enclave program, the host program invokes a special system call, namely `sgx_init()`, that allocates and initializes the system-wide EPC pages. The `sgx_init()` system call first reserves a memory region (a contiguous virtual address space) for EPC and then notifies the reserved region to the QEMU via a special instruction. At runtime, the QEMU instruction translator enforces the access permission of the memory region allocated for EPC pages.

To be precise, a significant difference between the OpenSGX emulation and the Intel SGX hardware is that, while Intel SGX keeps track of the permission of EPC pages via MMU (physical address), OpenSGX mimics the behavior of such enforcement at the user space (virtual address).

EPC access protection. To protect the memory of enclaves (isolation), OpenSGX needs to ensure that an enclave accesses only its own EPC pages and normal processes never access the enclave memory. To enforce such memory accesses, OpenSGX interposes every single memory access and checks the execution context (e.g., enclave or non-enclave) and the corresponding access permission. Specifically, OpenSGX instruments all memory accesses by checking the memory operands of all x86 instructions. Also, for access control to individual EPC pages, OpenSGX maintains their access permissions and the owner enclave in a CPU-specific data structure, called Enclave Page Cache Map (`EPCM`), following the Intel SGX [21, 22][Ch. 1.5.1].

OpenSGX defines two kinds of memory accesses: *enclave accesses* that are initiated by an enclave program and *non-enclave accesses* that are equivalent to traditional memory accesses [21, 22][Ch. 2.3]. However, regardless of CPU modes (i.e., enclave mode or not), all memory accesses should be coordinated by an underlying memory management unit (e.g., permissions in the page table entries), which means an enclave program and its host application will share a process' linear address.

Depending on the entity (from enclave or host code) and the type of memory region (either enclave or host data), Intel SGX decides whether to approve the requested memory access or not [36]. Memory accesses to own data or code are always permitted (e.g., enclave code → its data or host code → its data); however, memory accesses to another enclave's code are strictly prohibited. Note that memory accesses from an enclave to its *host's data or code* are permitted [36][Figure 2]. OpenSGX leverages this to set aside a shared memory for communication between enclaves and their host programs (e.g., *trampoline*), as explained in §IV-D.

EPC encryption. OpenSGX is not intended to run enclave programs under strong adversarial models; the TCB of OpenSGX includes its emulation layer, host operating system, and hardware, unlike Intel SGX, which can provide a strong protection of running enclaves with a single trust, the CPU. Accordingly, OpenSGX does not perform encryption for every single memory operation, but rather simply restricts the memory accesses to the EPC region because this relaxation does not affect the functional correctness of OpenSGX.

However, OpenSGX as a baseline platform can be easily extended to implement various types of Memory Encryption Engines (MEE) or exploratory encryption schemes for research. Furthermore, combined with the precise cache tracking provided

by QEMU, one can quantitatively measure and compare the performance of potential encryption schemes. For example, AES CTR Encryption or MAC algorithms described in the Intel SGX Workshop [24] can be easily implemented and their performance impact can be studied in a well-controlled environment.

Data structures and SGX *processor key*. To accurately emulate Intel SGX, OpenSGX implements critical data structures described in the Intel SGX specifications from Intel, including SGX Enclave Control Structure (`SECS`), Enclave Signature Structure (`SIGSTRUCT`), Thread Control Structure (`TCS`), and Enclave Page Cache Map (`EPCM`) inside the emulator as additional states for the CPU (see Table II). Finally, OpenSGX provides a configurable SGX *processor key*, which is a unique key that Intel SGX provisions to each processor. SGX *processor key* serves as the root of the key hierarchy that is used to generate keys, including launch key, report key, provision key, and seal key. Note that Intel SGX uses a group signature scheme (EPID [9]) for attestation and verification to prevent an SGX-equipped platform from being uniquely identified. In our current implementation, we use a public signature scheme (RSA) as a proof-of-concept and leave the adoption of an EPID-like infrastructure as future work.

B. OS Emulation Layer

Intel SGX does not work out-of-box for end-users. It requires an intimate assistance from operating systems to properly launch an enclave program. In OpenSGX, instead of requiring users to use a specific type of operating system or relying on a custom implementation, we expose an OS-neutral interface by implementing an underlying emulation layer that executes the privileged SGX instructions. The OpenSGX OS emulation layer implements a set of new system calls for enclaves, described in Table III. The OS emulation layer provides three major services for enclave programming: 1) enclave initialization, 2) EPC page translation, and 3) dynamic EPC page allocation. In addition, OpenSGX implements two additional system call interfaces to bootstrap the emulated hardware and to fetch the internal report generated with the performance profiler.

Bootstrapping. `sys_init()` performs the bootstrapping process as described in §IV-A. After bootstrapping, the OS emulation layer obtains a contiguous chunk of EPC and its physical address. Then, we can use the EPC region to initialize an enclave. Note that the EPC access is controlled by OpenSGX QEMU, and thus is only visible to QEMU. The OS must map EPC pages to an enclave using appropriate instructions before an enclave can access EPC pages.

Enclave initialization. OpenSGX initializes an enclave with four SGX instructions, namely `ECREATE`, `EADD`, `EEXTEND`, and `EINIT`. A new system call, `sys_create_enclave()`, is implemented to initiate, extend, and measure an enclave program, and requires privileged SGX instructions. `ECREATE` first creates an empty enclave by allocating the number of requested EPC pages (specified in `n_of_pages`) and initiates the measurement of the new enclave. Starting from `base_address`, which indicates the starting address of code and data memory, `EADD` loads all pages in this source memory to the allocated EPC pages in sequence. Whenever a new EPC page is added, `EEXTEND` needs to be executed to correctly measure the page content, and it extends the measurement registers of the initializing enclave. Finally, `EINIT` completes the enclave measurement and the initialization process. `sig` and `token` are used to verify the enclave measurement, representing `SIGSTRUCT` and `EINITTOKEN` respectively. `sig` contains a pre-computed enclave measurement, a signature over the data structure using a private key (usually signed by developer), and also the corresponding public key that represents the certified issuer. During `EINIT`, the pre-computed enclave measurement is first verified by using the signature and the public key. Then, OpenSGX compares the pre-computed measurement with the final measurement. If two measurements are equal, the enclave becomes ready to securely execute the enclave code in isolation.

To support multiple enclaves concurrently, OpenSGX maintains a per-enclave structure that describes the execution context of each enclave; for example, an enclave id, contents of `TCS`, and stack size are stored, similar to `task_struct` in Linux. The structure also contains debugging information and performance counters (e.g., the number of leaf commands executed), leaving rooms for future extensions for research.

EPC page translation. For convenience, the OS emulation layer of OpenSGX pretends to identically map the virtual address of EPC pages to the physical memory space, similar to the direct-mapped regions in the Linux kernel. This design decision simplifies the implementation of the emulated OS and helps researchers using OpenSGX to conceptually distinguish physical and virtual addresses in their projects.

Dynamic EPC page allocation. The Intel SGX revision 2 [5, 22] provides a mechanism to dynamically expand the enclave memory by using `EAUG` and `EACCEPT`. Based on these two instructions, OpenSGX provides `sys_add_epc()` to dynamically allocate additional EPC pages for the enclave that requires more memory. When an enclave needs a new EPC page, the emulated OS allocates a free EPC page via `EAUG`. Then the enclave should invoke `EACCEPT` to accept the new page to its own enclave region. In fact, `EACCEPT` embodies a few interesting checks that play a key role in thwarting the Iago attack, which is descried in §IV-D.

Performance monitor. When an enclave is created, OpenSGX keeps track of the new enclave by assigning a custom identifier (`keid`) in the emulated OS and a descriptor. For the given `keid`, the enclave descriptor collects stat/profiling information including statistics and enclave-specific metadata (e.g., `SECS` and `TCS`). A host application later can query the collected profiling information through `sys_stat_enclave()`.

System call emulation. The OS emulation layer is implemented as a user space library that OpenSGX programs can link to. Thus, we emulate the system call interface using the function calls that follow the convention implemented in the compiler, instead of using the conventional system call interface. When a system call is invoked inside an enclave, a context switch occurs by first storing the context of the enclave inside a specially reversed region inside the EPC, called State Save Area (`SSA`). Then, we exit the enclave and context switch to the kernel. After the kernel's execution of the system call, it returns back to the enclave to restore its context and verifies the kernel's return value inside the enclave. `sgxlib`, described in §IV-D, automatically performs these tasks during the invocation of `sys_add_epc()`.

Instruction	Description
`bool sys_sgx_init()`	Allocates EPC, sets cpusvn, and initializes sgx and custom data structures in QEMU side.

`int sys_init_enclave(`
`  void *base_address,`
`  unsigned int n_of_pages,`
`  tcs_t tcs,`
`  sigstruct_t *sig,`
`  einittoken_t *token)`
	Allocates, adds, measures EPC pages, and initialize OS-specific structures.
	Starting address of code/data pages, a linear address
	The number of total pages to be loaded
	Thread control structure address used for entering enclave, a linear address
	Information about the enclave from the enclave signer
	Token for verifying that the enclave is permitted to launch
	Leaf commands: `ECREATE`, `EADD`, `EEXTEND`, `EINIT`

`unsigned long sys_add_epc(` `  int keid)`	Allocates a new EPC page to the running enclave. Enclave id Leaf commands: `EAUG`

`int sys_stat_enclave(`
`  int keid,`
`  keid_t *stat)`
	Obtains the enclave stats: such as eid, #encls, #enclu calls, allocated stack/heap, perf etc.
	Enclave id
	Container of stat information of enclave

TABLE III: List of system calls that OpenSGX newly introduced to the kernel in order to coordinate enclave programs. In Intel SGX, the operating system should be in charge of authorization, fairness, and execution of the requested enclave program in order to fully take advantages of the OpenSGX-compatible hardware. We introduced four different system calls (not specified in Intel SGX) and explored the possibility of deploying a subsystem to support Intel SGX in the commodity operation system such as Linux.

C. OpenSGX Toolchain, Compilation and Loader

OpenSGX provides a toolchain, called `opensgx`, that automates the building process of an enclave program. Figure 2 shows an example of how developers can use `opensgx` to generate an OpenSGX-aware binary (a `.sgx` file) that contains code and data for enclave programs, as well as a configuration file (a `.conf` file) that contains required keys and measurement for `SIGSTRUCT` and `EINITTOKEN` data structures.

Compilation. One key feature of `opensgx` is that it generates a binary that can be easily relocated to EPC. According to the SGX specification, `EADD` instruction loads code and data into EPC by direct memory copying, which implicitly assumes that developers take care of program relocation by themselves. To ease the developers' efforts in handling program relocation, OpenSGX provides a build script to automatically tweak the compilation options to make the enclave code and data easily relocatable at runtime. More specifically, OpenSGX provides a custom linker script that specifies the locations of all code and data (including initialized, uninitialized, and global data sections) properly onto the enclave address space.

After the compilation with `opensgx`, the final enclave binary will include a set of pre-defined symbols (e.g., `enclave_main()` that describes the entry point of the enclave code) and embed `sgxlib` as a separate section. For more involved source code, developers can provide a custom linker script that specifies the code (`.enc_text`) and data (`.enc_data`) sections to be included in the enclave. As an optimization, OpenSGX can avoid additional relocation of both sections by statically linking all symbols at compile time.

Loader. OpenSGX loader determines the memory layout of code, data, stack, and heap sections, and necessary data structures on the EPC region during the initialization of an enclave. Similar to a typical loader, the OpenSGX loader obtains the information of code and data sections (i.e., offset and size of `.enc_text` and `.enc_data` sections) and the program base address from corresponding ELF files. The required enclave size and the memory layout are determined based on code and data size, memory configuration (we set default heap and stack size and allow developers to easily adjust), and other necessary data structures (see Figure 3). Then, the OpenSGX loader forwards the memory layout information to the OS emulation layer to initiate the enclave initialization process. Note that the starting address of EPC for loading is statically determined by the base address of code and the data section so that the base address remains the same after loading into EPC.

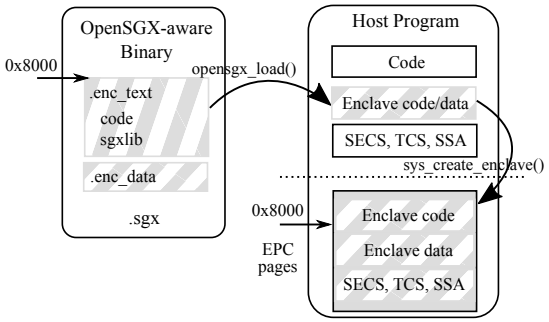


Fig. 3: Loading process performed by OpenSGX loader. First, `.enc_text` and `.enc_data` sections are loaded in to host memory. OpenSGX loader then forwards two sections along with stack, heap, and other necessary data structures to EPC via `sys_create_enclave()`.

D. OpenSGX User Library

`sgxlib` is a user-level library for enclave that is designed to (1) facilitate the enclave programming and (2) minimize the attack surface between the enclave and its *potentially malicious* host process. Table IV lists APIs implemented by `sgxlib`, classified into `HOST` for host applications and `ENCLAVE` for enclave programs. This section describes the design decisions made in `sgxlib` and its security considerations.

Custom in-enclave library. Standard `C` libraries, such as `glibc`, are frequently used by normal `C` programs. However, using standard `C` libraries inside an enclave raises two concerns: (1) any function call that relies on OS features or resources will break the execution of enclave programs and (2) enabling such functions opens up new attack surfaces (e.g., malicious host can return a crafted input to the enclave). Thus, we implement a number of custom library functions that have a similar counterpart in the standard library, but we add a `sgx_` prefix to distinguish the two (e.g., `sgx_memmove()` for `memmove()`).

Trampoline and stub. Although an enclave can legitimately

Type	API	Description
HOST	`void sgx_init(void)`	Perform system initialization
HOST	`void sgx_enter(tcs_t tcs, void (*aep)())`	`EENTER` wrapper
HOST	`void sgx_resume(tcs_t tcs, void (*aep)())`	`ERESUME` wrapper
HOST	`int sgx_host_read(void *buf, int len)`	Read from enclave
HOST	`int sgx_host_write(void *buf, int len)`	Write to enclave
HOST	`void launch_quoting_enclave(void)`	Launch quoting enclave
ENCL	`void sgx_exit(void *addr)`	`EEXIT` wrapper
ENCL	`void sgx_remote(const struct sockaddr *target_addr, socklen_t addrlen)`	Request remote attestation
ENCL	`void sgx_getkey(keyrequest_t keyreq, void *key)`	`EGETKEY` wrapper
ENCL	`void sgx_getreport(targetinfo_t info, reportdata_t data, report_t *report)`	`EREPORT` wrapper
ENCL	`int sgx_enclave_read(void *buf, int len)`	Read from host
ENCL	`int sgx_enclave_write(void *buf, int len)`	Write to host
ENCL	`void *sgx_memcpy(void *dest, const void *src, size_t size)`	Memory copy
ENCL	`void *sgx_memmove(void *dest, const void *src, size_t size)`	Memory copy
ENCL	`void sgx_memset(void *ptr, int value, size_t num)`	Memory set to the specified value
ENCL	`int sgx_memcmp(const void *ptr1, const void *ptr2, size_t num)`	Memory comparison
ENCL	`size_t sgx_strlen(const char *string)`	Get string length
ENCL	`int sgx_strcmp(const char *p1, const char *p2)`	String comparison
ENCL	`int sgx_printf(const char *format, ...)`	Write formatted data to standard out

TABLE IV: List of APIs in `sgxlib`. `HOST` APIs are for host process and `ENLC` is for in-enclave use.

access the host memory shared outside the enclave, it is not a recommended practice since a malicious host or operating system can potentially modify non-enclave memory. Thus, instead of allowing such a practice, OpenSGX provides a stricter form of communication protocol by using shared code and data memory—we call them *trampoline* and *stub*, respectively. The use of trampoline and stub defines a narrow interface to the enclave, which is readily tractable for enforcing the associated security properties.

The communication is one-way and entirely driven by the requesting enclave. For example, to request a socket for networking (see Figure 4), the enclave first sets up the input parameters in *stub* (e.g., sets `fcode` to `FSOCKET` in Figure 4), and then invokes a predefined handler, *trampoline*, by exiting its *enclave mode* (i.e., by invoking `EEXIT`). Once the host program (or OS) processes the enclave request, it stores the result or return values to *stub* and enters the *enclave mode* again by invoking `ERESUME`. After transferring the program's control back to the known location inside the enclave, the enclave program can finally obtain the returned value (e.g., socket via `in_arg0` in *stub*). The current design of *trampoline* and *stub* is extensible enough for other purposes, allowing customization by enclave developers without reinventing the whole communication protocols.

Enclave-Host communication. `sgxlib` provides a dedicated communication channel between an enclave and its host, similar to a pipe. The host program sends data via `sgx_host_write()` (respectively `sgx_host_read()` for receiving) and the enclave receives data via `sgx_enclave_read()` (respectively `sgx_enclave_write()` for sending). The communication APIs are implemented by using *stub* (without *trampoline*). More specifically, we pre-allocate two memory blocks alongside *stub* to support two different data flows. For security reasons, the memory block is cleared before/after each write/read operation.

Dynamic memory allocation. Although it is permitted for an enclave program to use dynamically allocated host memory, it can severely break the enclave isolation feature. To avoid this, `sgxlib` supports a customized dynamic memory allocation API, `sgx_malloc()`, which behaves similarly to glibc `malloc()` [50], but only allocates memory from the enclave heap (pre-allocated EPC pages, see Figure 3). `sgx_malloc()`

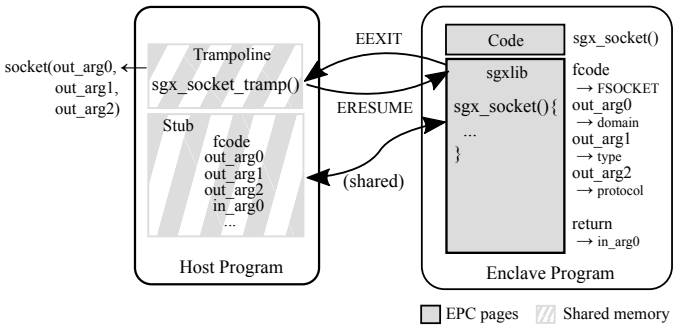


Fig. 4: Interface defined for communicating with the enclave's host program that performs the delegated calls to the operating system. In this figure, a `sgxlib` library, `sgx_socket()`, running inside the enclave, requests a socket system call via *trampoline* and *stub*, which are pre-negotiated between the enclave and its wrapper when packaged together.

manages the enclave heap by maintaining heap pointers, which are initially set to the heap with the aid of OS during the first initial `sgx_malloc()` call. When a pre-allocated heap area becomes full, `sgx_malloc()` leverages the dynamic EPC page allocation (via `sys_add_epc()`) to extend the enclave heap. With `EAUG/EACCEPT`, the dynamic EPC page allocation ensures that only a zero-filled EPC page, with an associated pending bit of `EPCM`, is added to the enclave that invoked `EACCEPT`. Since the pending bit can be switched only by executing `EAUG/EACCEPT`, a malicious OS cannot deceive (Iago attack) the enclave to add another EPC page. If an attacker passes an address that overlaps an EPC stack page to `EACCEPT`, it returns an error code.

Defense against malicious host application and OS. To provide enclave with the ability to communicate with host application or OS, it is unavoidable to introduce additional attack surfaces to the enclave, which is often known as Iago attacks [11]. `sgxlib` is designed after careful consideration of the Iago attacks, broadly in three aspects: dynamic memory allocation, network and I/O, and non-determinism/resources. Note that Intel SGX neither prevents denial-of-service attacks nor guarantees strong privacy (e.g., IP address), but provides strong isolation and confidentiality. With this in mind, we inspect potential attack surfaces in Table V and discuss defenses

Type	Interface	Attack surfaces	In-enclave usage/check
MEM	sgx_malloc() → <out>addr	1) incorrect pointers 2) incorrect EPC page addition	EACCEPT verifies the status of a new EPC page
MEM	sgx_free() → N/A	1) not freed (used later for use-after-free)	sgx_free() fills a freed chunk with zero
DBG	sgx_puts() → N/A	1) ignored output	No general way to prevent without trusted I/O
TIME	sgx_time() → <out>time	1) arbitrary time	Validate time from the NTP server (not implemented)
RAND	sgx_rand() → <out>rand	1) arbitrary value	Relying on rdrand inst (emulated if not supported)
IO	sgx_write() → <out>len	1) arbitrary reported len	No general way to prevent without trusted I/O
IO	sgx_read(<out>*buf) → <out>len	1) crafted buf, 2) incorrect len	Encrypted message with integrity checking
IO	sgx_close() → N/A	1) not closed	Never reuse fd (monotonically increasing int)
NET	sgx_socket() → <out>fd	1) non-closed fd, 2) incorrect fd	Relying on packet encryption
NET	sgx_send() → N/A	1) ignored	Relying on packet encryption
NET	sgx_recv(<out>*buf) → <out>len	1) crafted buf, 2) incorrect len	Relying on packet encryption
NET	sgx_accept() → <out>fd	1) pre-allocated fd, 2) arbitrary number	Relying on packet encryption
NET	sgx_bind() → N/A	1) failed binding	Stop-on-failure
NET	sgx_listen() → N/A	1) failed listen	Stop-on-failure
NET	sgx_connect() → <out>err	1) failed connection	Stop-on-failure

TABLE V: Consideration of Iago attacks in primitive sgxlib functions that are implemented by using the shared trampoline between host and enclave programs. Note that Intel SGX does not consider denial-of-service attacks (e.g., stopping enclave execution) nor strong privacy (e.g., where to talk to).

on each category of attack.

- **Memory-related operations (marked MEM):** Since the Intel SGX revision 2, enclave programs can dynamically request EPC pages at runtime, which opens a large attack surface, traditionally known as Iago attacks. However, Intel SGX takes this into account and provides an EACCEPT instruction that performs basic validation on newly assigned EPC pages (e.g., non-enclave pages or pre-allocated EPC pages), which thwarts a major source of memory-related attacks.
- **Network and I/O services (marked IO, NET):** Two principles are considered to prevent network- and I/O-related attacks: encryption and the fail-stop model. To guarantee the confidentiality of packets, enclave programs should encrypt all out-going packets and also validate the integrity and authenticity of packets on all incoming packets. Upon any failure, the enclave stops its execution, which dramatically reduces the potential attack surface in handling all errors and corner cases.
- **Non-determinism and resources (marked DBG, TIME, RAND):** Enclave programs often need time and randomness to provide rich experiences to users. To prevent Iago attacks, OpenSGX attempts to fetch such values from trusted parties: time from trusted entities (e.g., an encrypted packet from known NTP servers) or randomness from trusted CPU (e.g., rdrand instruction).

Remote Attestation. OpenSGX provides sgx_remote() with which the programmers can readily generate a remote attestation request in an enclave program through a single API. It uses sgx_getkey() and sgx_getreport() to get a report key and create a report. By specifying the socket information of a target enclave, a challenger can issue a remote attestation to check (1) the correctness of the target program (based on the hash of EPC contents) and (2) whether it is actually running inside an enclave on the SGX-enabled platform (MAC with report key). To launch and service a special enclave called quoting enclave that verifies a target enclave through intra-attestation, OpenSGX also provides launch_quoting_enclave() host API. The overall procedure of remote attestation in OpenSGX is implemented based on the SGX specification [2] by using an RSA key scheme as an alternative to EPID (see §IV-A).

E. Debugging

Debugging is an essential feature for OpenSGX programmers to develop SGX-aware applications. As a software emulator, OpenSGX can be easily integrated with a mature debugging infrastructure such as gdb, not only to mimic Intel SGX debugging-related instructions but also to examine the status of internal hardware.

Debugging hardware. The hardware components of Intel SGX are neither observable nor modifiable in real hardware. However, because OpenSGX emulates them using software, developers can observe the inner working of Intel SGX instructions and internal data structures stored inside the hardware (e.g., EPCM). OpenSGX provides a gdb interface to debug its emulated hardware components.

Debugging enclaves. Since OpenSGX is implemented by using QEMU's dynamic code translation (known as TCG), the gdb on OpenSGX can only see the instructions translated by the TCG, not the application's original instructions. To provide a single stepping feature in debugging an enclave code, we implement gdb-stub, which is a platform to support a bridge to the remote gdb instance. The gdb-stub inside QEMU helps gdb to understand the context of the enclave's original instructions, thus enabling a convenient single-step debugging. Once a remote gdb instance is connected, developers can debug enclave programs with a familiar gdb interface.

New gdb commands. On top of the built-in gdb commands, OpenSGX provides four new useful gdb commands that researchers can use in debugging enclave programs. They are info epc, info epcm, and info secs to examine EPC-related data structure, and list enclaves to list all the active enclaves (and their contexts) with corresponding eid. All new commands are implemented by using the gdb-python script.

F. Performance Monitoring

Unlike real hardware, the emulation environment can provide a precise, yet flexible way to measure performance aspects of running enclaves. Since the wall-clock time is not a meaningful metric in OpenSGX, we provide various other metrics (e.g., context switches, invocations of SGX instructions) that are useful to understand the performance characteristics of an enclave.

OpenSGX keeps track of such information in a per-enclave

Component	Lines of code
QEMU SGX	5,954 lines of C
OS emulation	1,531 lines of C
SGX libraries	2,978 lines of C
Examples	1,532 lines of C
Tool	2,199 lines of C
Tor	5,087 lines of C
Total	19,281 lines of code

TABLE VI: The modified lines of code for each component in OpenSGX.

data structure, called an enclave descriptor, that stores the fields such as TCS and usage statistics. In particular, it counts the number of context switches, the entries/exits of the OS emulation layer; the number of `ENCLU`/`ENCLS` leaf instructions that can indicate the behavior of an enclave; the number of TLB flushes, as it is reported as the main overhead in enclave transition [5]; and the number of dynamically allocated EPC pages, which is a good indicator to the enclave memory usage. Then, the host program later can query its statistics through the `sys_stat_enclave()` system call.

As OpenSGX is built on top of the software stack, it can be easily extended to measure other behaviors, such as Last Level Cache Misses [42]. However, the current prototype neither provides nor guarantees any precision of direct and time-oriented performance characteristics of enclaves running on Intel SGX, because OpenSGX never reflects the actual cost of memory isolation or encryption provided by the SGX hardware during the emulation.

V. OpenSGX Implementation

Table VI shows the number of lines of code (LoC) of OpenSGX's components. OpenSGX and its applications consist of approximately 19K lines of `C` code with QEMU SGX being the largest component.

To implement QEMU SGX, we extensively modify several components of QEMU. In particular, QEMU holds all x86 CPU register values, such as `EIP`, in a data structure called `CPUX86State` that represents the CPU state of the guest machine. We extend the CPU state by adding the `CREGS` data structure [21, 22], to support additional CPU states introduced by OpenSGX. The `CREGS` structure maintains registers related to the enclave context, such as the on/off state of *enclave mode* and the current instruction pointer.

To control a program's next executing point upon enclave entries/exits, we execute our mode checking routines before generating a new translation block (TB). For example, if a TB executes an `ENCLU` instruction (except `EEXIT`) and *enclave mode* is on, the `CR_CURR_EIP` value in `CREGS` overwrites the `EIP` in `CPUX86State`. Since `CR_CURR_EIP` is supplied while emulating the `ENCLU` instruction, it directs QEMU to generate/execute the next TB from the supplied `EIP` instead of the one determined from executing the current TB without `ENCLU`.

To emulate `ENCLU`/`ENCLS` instruction families and memory protection, we utilize the QEMU helper routine. The QEMU helper routine, which consists of `gen_helper_*()` and `helper_*()`, was originally designed to interpose an event such as an interrupt while emulating guest instructions. We add `gen_helper_enclu/encls()` functions at the point where the opcode for `ENCLU`/`ENCLS` instruction is found, while translating a guest instruction. This ensures that QEMU calls `helper_enclu/encls()` when the `ENCLU`/`ENCLS` instruction is invoked. Then, we implement the `ENCLU`/`ENCLS` leaf instructions inside `helper_enclu/encls()`. For memory protection, we insert `gen_helper_mem_access()` at all points where an instruction causes memory load/store, and `gen_helper_mem_execute()` before jmp/call/ret instructions. The former enforces memory access control according to *enclave mode* state and `EPCM`'s read/write permissions, whereas the latter performs Data Execution Prevention (DEP) within the EPC according to `EPCM`'s execute permission. If the *enclave mode* is on, `helper_mem_access()` generates an exception when an enclave code is trying to access EPC outside its enclave or `EPCM`'s read/write/execute permissions are violated. If the *enclave mode* is off, all accesses to the EPC region are simply blocked.

Attacks	Target node	Description
Compromising keys	Directory	Tampering with voting/consensus creation
Constructing bad consensus	Directory	Tie-breaking, Include compromised ORs
Spoiled Onion [52]	Exit node	HTTPS man-in-the-middle attack, sslstrip
Bad Apple [6]	Exit node	De-anoymize, plaintext tampering/snooping

TABLE VII: Attacks on Tor

Node type	Data structure	Description
Directory node	Identity key	Certify signing key
	Signing key	Sign vote and consensus
Exit node	Identity key	Sign router descriptor/TLS certificate
	Onion key	Encrypt/decrypt cells

TABLE VIII: Key data structures of Tor that are protected inside the enclave

VI. Application Case Studies

To evaluate OpenSGX, we develop a number of applications utilizing SGX. Our experience demonstrates that OpenSGX supports non-trivial applications and is a suitable platform for testing new ideas and developing SGX applications. We focus on two applications, namely Tor and secure I/O paths.

A. Shielding Tor Nodes

Motivation. The Tor network relies on a network of world-wide volunteers to achieve anonymous communication. While the software is open source and publicly available for verification, its hardware is donated by volunteers who run pieces of Tor software (e.g., onion routers [15]). Thus, the current model is that users (semi-) trust the execution environment that volunteers provide using the hardware and operating system of their choice. When a security breach actually occurs to a server that hosts Tor's directory service, identity keys of the server need to be replaced. This causes users to update their software, which is very cumbersome [14].

Applying trusted execution on the core components of Tor will strengthen the security of the Tor network and enhance its trust model. Instead of trusting the hardware and the system software that hosts Tor relays, users can simply trust the underlying TEE. Using TEEs also allows us to secure the interactions between Tor nodes, allowing Tor to deliver its anonymity service on top of a distributed TEE.

To demonstrate the benefits, we first look at the known attacks on Tor and examine how trusted execution may strengthen the security of the Tor network. Finally, we present a design and implementation of Tor that utilizes the TEE.

Node type	Enclave Operations	Non-enclave Operations
Both	Create key pairs, offer key information	Transfer control/relay cells, initialize data structures, ...
Directory node	Create certificates, sign and verify vote/consensus	Transfer votes, broadcast consensus, TLS connection, ...
Exit node	Create TLS certificates, encrypt/decrypt packets	Get connection with destination, send/receive packets, ...

TABLE IX: Separation of Tor operations in a directory server and an exit node. Operations which accesses to sensitive data are located inside the enclave. Rest of Tor operations are executed in untrusted region for saving EPC pages.

Threat model. Previously known attacks on Tor that deanonymize Tor users can be divided into two categories in terms of the target of manipulation [47]: *manipulating Tor components* [30, 53] and *manipulating routing* [26, 47]. We assume the attack model that manipulates the Tor components, including Tor nodes and directory servers, by gaining control over the nodes. The consideration of network-level adversaries, such as a man-in-the-middle and AS-level adversaries, is out of scope in this paper.

Tor consists of a small number of directory servers. Attacks mainly target the directory server, entry, and exit node because subverting the middle relay nodes is not sufficient to break Tor's anonymity. There are several known attacks and reported security incidents to Tor [14]. Table VII summarizes several known attacks against Tor. The first two target the directory server to either compromise the directory server's key or force them to sign an arbitrary consensus that an attacker specifies. Attackers then can leverage this power to admit malicious Tor nodes that themselves control and influence Tor nodes that users choose from. Once attackers have control over Tor nodes, they can launch attacks to break the anonymity or eavesdrop an ongoing communication [1, 17, 30]. For example, a malicious exit node can inject or modify HTMLs, conduct man-in-the-middle attacks, and modify DNS responses [53]. These techniques, in turn, can be used to identify the source address of Tor users or obtain other private information.

Benefits of applying TEE. Utilizing TEEs can enhance the security of Tor in two ways.

- Attestation of software components: Users can perform remote attestation to ensure that Tor nodes are running the unmodified code by verifying the integrity of software. We believe that this will help users to select Tor nodes, and the Tor network can benefit from the new trust model.
- Protection against tampered OS: One can also protect Tor from the malicious operating systems or subverted system software. In particular, the memory region that Tor is using can be protected against a malicious OS by running the critical components of Tor inside an enclave. Even if the OS is tampered with, the private keys are not exposed to the attackers. This is especially helpful in protecting directory servers, as revoking their keys often requires reinstallation of all Tor nodes [14].

We demonstrate the benefit of our OpenSGX implementation by porting Tor to OpenSGX. We adopt the second approach to protect Tor against tampered system software. In particular, we separate critical parts of Tor that use cryptographic operations and store all private and session keys inside EPC. For directory servers, we also store the consensus (i.e., the list of Tor nodes).

OpenSGX-based Tor design. We take a minimalistic approach in which we define a narrow interface between generic Tor code that runs outside the enclave, interacting with the OS, and Tor-enclave that runs inside OpenSGX. Tor-enclave contains sensitive data structures to be protected and related functions that utilize the data structure. Generic Tor code and Tor-enclave run as separate processes. The generic Tor process (Tor-non-enclave) invokes RPC to request services that Tor-enclave provides. In our implementation, we port Tor's directory server and exit nodes. Table VIII shows the core data structures of the directory server and Tor exit node that we protect inside the enclave. For the case of a directory server, it has two private keys: an authority identity key and an authority signing key. Directory authority uses the authority identity key to certify the authority signing key. The authority signing key is used for signing and verifying votes and consensus documents, which are important information for a Tor circuit creation. For an exit node, a secret identity key is used for signing a router descriptor and TLS certificate while constructing a 3-hop circuit. Finally, an exit node uses a secret onion key to decrypt a relay cell received from the previous onion router.

Table IX summarizes the operation that Tor-enclave and Tor-non-enclave support. We contain all critical operations that use private data structures in Tor-enclave and expose an RPC interface to Tor-non-enclave. Tor-enclave only receives requests from Tor-non-enclave. We summarize some of the operations supported by Tor-enclave. First, it supports secure key generation and stores the keys inside EPC. Also, since operations such as creating certificates and signatures require a private key, these modules should be run inside the enclave. Information related to the private key (e.g fingerprint, digest and public key string) is also offered by Tor-enclave.

B. Secure I/O path

OpenSGX allows us to extend the platform and develop new ideas in conjunction with TEE. To demonstrate its flexibility, we implement a simple secure I/O path that allows secure communication between the CPU/memory and devices. The idea has been explored by Intel with its Identity Protection Technology (IPT) [10], which supports protected transaction display and audio I/O. However, the development platform is not widely available to the research community.

While many applications require secure I/O paths [10], to focus our discussion, we explore this in conjunction with our main application of the paper, Tor. In particular, Tor's exit node can benefit from having a secure network I/O path between Tor-enclave and the NIC. Tor exit nodes decrypt the packets and can observe plain-text unless end-to-end encryption is used between the client using Tor and the server it is communicating with. This has led to a number of security vulnerabilities in which Tor exit nodes modify or eavesdrop on the message [53]. A secure I/O path can protect users from these attacks. In particular, if the communication channel between the Tor-enclave and the NIC is secured, (subverted) system software cannot eavesdrop on or modify the message. Only an attacker that has access to the network between the Tor exit node and the server can mount the attacks, but the Tor network has protection against attacks from inside the Tor network.

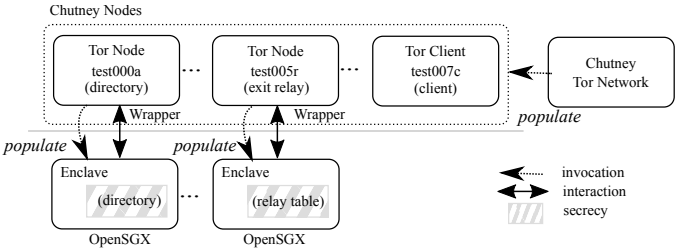


Fig. 5: An overview of the execution environment of an OpenSGX-enabled Tor. To reduce the size of TCB (an enclave program), we use SGX to protect the secrecy of Tor nodes; directory in Directory Node and relay table in Exit Node.

To this end, we emulate the encrypted communication channel with message authentication. We assume that a shared secret is established between the Tor-enclave and the secure network device. We also assume that the device can perform TCP/IP processing. All the messages between Tor-enclave and the secure NIC are then sent via the secure channel. Thus, the operating system of the Tor exit node cannot observe any plain-text communication between the client and the server.

C. Implementation

We implement Tor-enclave, Tor-non-enclave, and the RPC interface between the two. For directory servers, Tor-enclave contains the private key and the list of onion routers to be used, making the information private. For onion routers, we store their private keys in Tor-enclave. Non-private information and external interface to remote parties are handled by Tor-non-enclave. For evaluation, we use Chutney [48] to construct a private Tor network. Our private Tor network consists of seven nodes: three directory servers, three relays (onion routers), and a client proxy. At minimum, at least three directory servers are required to prevent tie-breaking, and three relay nodes are needed to build a 3-hop circuit. The process of running a private Tor network can be divided into three phases: key generation, consensus creation, and service phases. During the key generation phase, private keys and certificates are created for both onion routers and directory servers. In the consensus creation phase, directory servers sign votes and create consensus in order to agree upon the set of relays to be used. Then, a client proxy requests a list of available onion routers to directory servers. Based on this, the client proxy establishes a circuit. Finally, the client proxy sends/receives relay cells using the circuit for users' requested service.

VII. Performance Profiling

We evaluate OpenSGX by showcasing the SGX-enabled Tor application. Using Tor as a case study, we demonstrate that OpenSGX can run non-trivial applications and enable SGX application developers to profile their applications using our performance counter.

Environment setup. Figure 5 illustrates the overall execution environment of Tor with OpenSGX. Chutney [48] launches each node as a process within a single machine. Because we separated enclave components for the Tor directory and Tor exit node, each directory and exit node runs two processes: a Tor-non-enclave process and a Tor-enclave process. We use a Quad core Intel Core i5-4690 3.5GHz CPU machine running Linux 3.11.0 and tor-0.2.5.10 and torsock-1.3 for evaluation. To support cryptographic operations (e.g., RSA key creation) for Tor-enclave, we statically link and load the OpenSSL 1.0.2 library into the enclave.

Type	Number of pages		Note
	Directory node	Exit node	
Enclave pages	4	4	SECS and TCS
Code/Data pages	366	369	Including library
SSA pages	2	2	Configurable
Stack pages	50	50	Configurable
Heap pages	50	50	Configurable

TABLE X: The number of EPC pages for Tor-enclave process.

Type	Directory node			Exit node		
	Code	Data	Total	Code	Data	Total
OpenSSL	270	88	358	271	89	360
SGX libs	3	1	4	3	1	4
Tor-enclave	3	1	4	4	1	5
Total	276	90	366	278	91	369

TABLE XI: The number of EPC pages for code/data section of directory and exit Tor nodes.

We first quantify the number of EPC pages used to run Tor-enclave. Then, we measure the performance of SGX-enabled Tor using three metrics: additional instructions and CPU cycles, the number of context switches between enclave and non-enclave, and the number of RPC calls between Tor-enclave and non-enclave.

The number of EPC pages used. Table X shows the total number of EPC pages required to run Tor-enclave and their breakdown. We categorize EPC pages into five types: enclave pages, code/data pages, SSApages, stack, and heap pages. Enclave pages contain SECS and TCS data structures required for an enclave. Code and data pages are used for enclave code/data sections, which are proportional to the size of the enclave program. SSA pages are State Save Area pages used to support asynchronous exit. Stack pages are used as the stack section for the enclave program, and heap pages for dynamically allocated memory. The number of SSA, stack, and heap pages is configurable; we initialize them as 2, 50, and 50 pages. However, the number of EPC heap pages can increase during execution. For example, in a large Tor network, directory servers may require more heap space because they store the list of relays in EPC.

For code/data pages, Table XI shows the breakdown of EPC code and data regions. The page types are categorized by the OpenSSL library, SGX library, and separated Tor-enclave program. We see that the OpenSSL library dominates the EPC usage. This is because we load the entire library into EPC to support the cryptographic operations needed by Tor. The SGX library and Tor-enclave consist of five code and data pages only. Overall, the trusted component is relatively small (54%) compared to placing the entire Tor code base into the enclave without the separation.

Additional CPU cycles. We now evaluate the performance overhead of SGX-enabled Tor by measuring the additional number of instructions executed and CPU cycles consumed. The use of SGX instructions executed, SGX library calls, and system call support, such as enclave creation, contribute to the overhead. To quantify this, we obtain the number of all instructions and SGX instructions executed using OpenSGX by leveraging QEMU and OpenSGX performance counter. We then translate

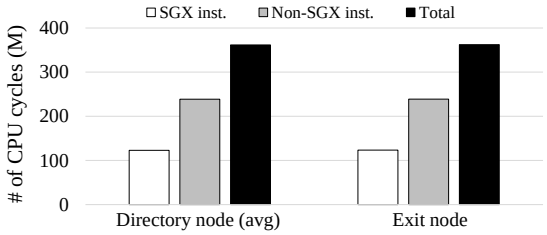


Fig. 6: The number of instructions and CPU cycles while loading Tor-enclave process. Here, (M) stands for a million.

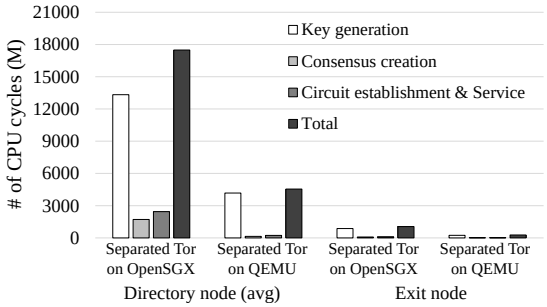


Fig. 7: The number of CPU cycles of Tor-enclave process for the three phases. Here, separated Tor on QEMU means executing Tor-enclave process without using SGX instructions and `sgxlib` calls on the native QEMU.

the instruction count to CPU cycles using the performance estimate from recent SGX literature. In particular, we assume that each SGX instruction takes 10K CPU cycles, and non-SGX instructions run at native speed within the enclave [5]. To estimate the CPU cycles for non-SGX instructions, we measure the average instructions per cycle by executing Tor natively without OpenSGX.[2] We report the average of 20 runs because the actual instruction count slightly varies depending on the random number generated during the cryptographic operations, such as the prime number.

Figure 6 shows the number of CPU cycles consumed to create and load the program into an enclave. It takes about 361M cycles to start enclave-Tor for a directory node and 362M cycles for an exit node. The CPU cycles required for loading both directory node and exit node are similar because the number of EPC pages required for running them are almost the same (see Table X). During the process, only the privileged `ENCLS` instructions are invoked (e.g., `ECREATE`, `EADD`, `EEXTEND`, and `EINIT`). Most of them are `EEXTEND` instruction because it is called 16 times for each EPC page (4KB) to obtain the hash value of its content. Additionally, non-SGX instructions are also invoked for handling the system calls (e.g., `sys_sgx_init()`). Note that enclave creation and program loading are one-time costs that only occur at the beginning.

We now quantify the overhead of key generation, consensus creation, and service phases of the Tor-enclave process. To estimate the overhead of using SGX, we compare the number of instructions and CPU cycles of Tor-enclave running on OpenSGX and on native QEMU without OpenSGX. Note that the latter just runs Tor nodes as two separate processes communicating through a pipe. Thus, the comparison shows the amount of extra overhead of using SGX. Figure 7 presents the number of CPU cycles for each phase.

The key generation phase only occurs once at the beginning

[2]The resulting average IPC is 1.81 instructions/cycle.

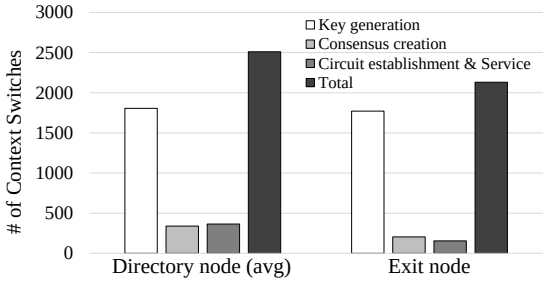


Fig. 8: The number of context switches (enclave exit and entry) of Tor-enclave process for directory and exit Tor nodes.

to create identity keys and signing keys for directory nodes and onion keys for exit nodes. Most of CPU cycles in this phase are used for generating RSA keys. SGX Tor consumes 3.2 times the CPU cycles of the separated Tor running on the native QEMU without OpenSGX for the directory node, and 3.6 times for the exit node. This is because the key creation phase uses multiple `sgxlib` calls, such as `sgx_malloc()`, that invoke `ENCLS` and `ENCLU` instructions and involve enclave exit and re-entry.

During the consensus creation phase, directory nodes perform voting and agree upon the relay and exit nodes to use. Because consensus creation is performed periodically, we measure the cost of the first consensus creation. The directory node consumes 12 times more cycles, while exit node spends 4.9 times more cycles in this phase. For both nodes, `sgx_send()` and `sgx_recv()` calls that are used for sending and receiving data (e.g., fingerprint, public key string, etc.) consume the extra CPU cycles because they involve enclave exit and re-entry. Furthermore, because we have separated the process into Tor-enclave and non-enclave, the RPC communication between the two involves `sgx_read()` and `sgx_write()` calls that also contribute to the overhead. Note that this is common across all three phases because our design puts part of the application code in the enclave in an attempt to reduce the TCB. Later in the section, we quantify the number of RPC calls in each phase.

In the service phase, our client proxy gets the list of Tor nodes by querying the directory nodes and generates a circuit from the proxy to an exit node using onion routing. After circuit establishment, the client's traffic is directed to the circuit via the proxy. We measure the overhead of a circuit establishment and serving a request at the exit node. For the latter, we generate a single `wget` request for `http://www.google.com`. Figure 7 shows that the directory node consumes 10 times more cycles and 5.6 times for the exit node. Similar to the previous case, additional overhead of the directory node is caused by `sgx_send()` and `sgx_recv()` calls to get a consensus verification result during the circuit creation. For an exit node, receiving encrypted relay cells from relay nodes requires `sgx_recv()` calls that contribute to the overhead.

Context switch overhead. We now quantify the number of context switches (i.e., enclave entries and exits) that occur during the Tor-enclave execution. Switching the CPU mode between enclave and normal mode incurs overhead, including saving and restoring the CPU state and registers and a TLB flush. Each invocation of `EENTER`, `EEXIT`, and `EREUSME` instructions causes the CPU mode to change. Figure 8 shows the number of context switches of the Tor-enclave process for

Type	Key generation	Consensus creation	Service
Directory node	14	40	24
Exit Node	74	288	10
Total	88	328	34

TABLE XII: The number of RPC calls between Tor-enclave and Tor-non-enclave during the three phases of Tor execution

	License	**Concurrency**	**OS support**	**Performance** [37]
TPM	Variable	No	Yes	Slow
Flicker	Dependent on TPM	No	Yes	Fast [a]
ARM TrustZone	ARM	Yes	Yes	Fast
Intel IPT	Intel	Yes	Yes	Moderately Fast
Intel SGX	Intel	Yes	Yes	Fast

[a] Flicker TEE code runs on main CPU fast, however it entails significant performance overhead when utilizing TPM operations [35].

TABLE XIII: Comparison of TEE hardware. **'Concurrency'** denotes the case when concurrency is supported by the device and **'OS support'** denotes the case when the TEE requires a special OS support.

directory and exit nodes during the three execution phases. A major source of context switching is system calls and I/O, such as `sys_create_enclave()`, `sgx_read()`, and `sgx_write()`. In particular, OpenSGX I/O APIs, such as `sgx_read()` and `sgx_write()`, use *trampoline* and *stub*, which cause the program to exit the enclave mode to request I/O operation to the kernel. Because enclave-Tor performs I/O frequently, the result shows that context switching occurs very often (i.e., every 7M instructions for a directory node and 0.5M instructions for an exit node). However, we expect that the cost of context switching can be amortized through batching system calls and I/O operations [44].

The number of RPC calls. Finally, we count the number of RPC calls between Tor-enclave and Tor-non-enclave. Although this is not a direct measure, it reflects another aspect of the overhead due to the new design of Tor (separation of Tor process). In our implementation, each RPC call involves `sgx_write()` or `sgx_read()` calls. Table XII shows the number of RPC calls measured for the three phases of Tor execution. In the key generation phase, an exit node additionally executes a larger number of RPC calls because it creates three X509 certificates for the TLS connection, whereas the directory node only creates a certificate for the signing key. Consensus creation involves a large number of RPC calls. In this phase, a directory node signs its votes, creates a consensus, and checks the state of reachable Tor nodes. Then, it sends a message periodically to Tor nodes and each node replies with liveness information by authenticating itself using its identity key. This causes many I/O calls during the launching phase in the exit node. In the service phase, the directory node requires RPC calls to verify the signature of consensus. Also, the exit node invokes several RPC calls for decrypting and encrypting DNS and HTTP requests. We believe that the cost can be amortized by batching the systems calls.

VIII. Related work

TEE has been considered an effective way of constructing a secure area residing in the main processor in mobile and smartcard platforms [49]. TEE is designed to ensure protected storage of sensitive data and to guarantee safe executions of trusted applications. Although various types of TEE including TPM, ARM Trusted Zone, Intel TXT, and AMD SVM, have already been deployed to those platforms [34, 41], their usage has not reached further into the cloud yet, due to their limited form factors and, critically, performance. The recent introduction of Intel SGX [2, 19, 36] changes this landscape by restricting the TCB (Trusted Computing Base) to the processor itself while providing the performance at the native hardware level (e.g., multiple threads support) inside an enclave. Table XIII summarizes the characteristics of currently available TEE technologies.

Intel SGX. A number of projects have explored applications of Intel SGX in the cloud environment. Haven [5] pioneered the idea of enabling unmodified application binaries to run on Intel SGX inside the cloud. VC3 [42] suggested using SGX for ensuring privacy in data analytics in the cloud. Both projects utilized the Intel SGX emulator provided by Intel to develop software that works on top of Intel SGX. However, the emulator has been available only to the authors of both projects. To the best of our knowledge, no SGX emulator is publicly available to the general research community. Thus, our focus in this OpenSGX project is to develop an openly available platform upon which new research ideas involving TEE can be readily implemented and explored.

Kim et al. [27] explore how to leverage SGX to enhance the security and privacy of network applications, such as software-defined inter-domain routing, Tor anonymity network, and in-network functions. They use OpenSGX to demonstrate the feasibility of the design and characterize the overhead of adopting SGX into application design, which demonstrates the usefulness of the OpenSGX platform.

Isolated execution environment. Hardware-based trusted execution environments have been constructed in various contexts. For example, Flicker [34] utilizes Late Launch; SICE [4] uses multi-core architecture; OASIS [38] proposes a cost-effective CPU ISA extensions for TEE; TrInc [31] provides trustworthy computation by using TPM for distributed systems; SecureSwitch [46] uses BIOS, and Secure Executables [7, 51] extends the power architecture to build a trusted execution environment. For low-end embedded devices, TrustLite [29] and Tytan [8] enforce execution-aware memory protection in a flat memory model.

Software-based solutions that utilize hypervisors as TCB have been explored (e.g., Trustvisor [35], MiniBox [33], NOVA [45], Overshadow [12], and seL4V [28]). The goal of these projects is to provide a secure environment. In contrast, OpenSGX does not provide any security guarantee due to its emulation nature, but offers a rich platform that enables new research.

Trusted I/O and remote attestation. Since Intel SGX does not provide a secure way of communicating with users, an SGX-compatible solution to establish a secure channel between a user to the computer (e.g., secure I/O) is required. In this regard, the integrity of the peripheral's software is important; VIPER [32] attempts to verify the integrity of device firmware; NAVIS [16] enables a similar integrity check to network adapters; IOCheck [54] provides a framework to enhance the security of I/O devices; and Intel recently introduced Intel IPT [10] to establish a secure display and audio channel. These solutions can be made compatible to Intel SGX in general, and by using our OpenSGX, one can also explore such an interface for trusted I/O, similar to our preliminary proof-of-concept example presented in §VI-B.

IX. Discussion and Limitations

In this section, we summarize the limitations of OpenSGX, describe our concerns about Intel SGX and its approach in general, and attempt to clarify prevalent misconceptions about Intel SGX.

Limitations of OpenSGX. First and foremost, OpenSGX is *not* secure for any security-related projects. However, OpenSGX can be utilized or extended for easy development of Intel SGX such as toolchains or a library, precise profiling of SGX programs, and the exploration of potential research opportunities beyond the software boundary (e.g., hardware semantics or efficient memory encryption scheme) that the Intel SGX can not flexibly enable.

Limitations of Intel SGX. Intel SGX is an ideal model for the cloud, as it has a very restricted set of the I/O channels (usually just the network and local disk). To use Intel SGX in a desktop-like, interactive environment, it is essential to establish a secure channel between users and the enclave program. There are already a few commodity hardware available on the market: Intel Protected Audio and Video (Intel PAVP) [20] and Intel Identity Protection Technology (Intel IPT) [10]. Although both technologies can be used for creating an encrypted I/O path of video and audio to an enclave program [19, 20], their usage seems restricted to mobile devices that accept a user's inputs through a touch-screen interface. However, integration into typical input devices such as a keyboard or mouse still needs to be solved for wide adoption of Intel SGX.

Rutkowska [39] raises a similar concern that even if SGX succeeds in preventing malware from stealing critical user data directly from memory, the absence of a secure input and output can allow malware to potentially command the enclave to leak them.

Another misconception of Intel SGX is that its trust (i.e., remote attestation) can be constructed without any future reliance on Intel once an SGX-enabled device is deployed. However, to properly perform a remote attestation, the report certified by the quoting enclave should be checked for an EPID infrastructure provided by Intel (e.g., checking identity or revocation) [9].

Malicious use of Intel SGX. As recently highlighted in detail by a handful of researchers [13, 40], we have similar concerns in terms of safe use of Intel SGX; for example, irreversible malware might be possible by abusing the isolation property, so unanalyzable, guaranteed by the hardware. Worse yet, end-users or anti-virus software will not be able to distinguish from a compromised instance of OpenSGX and its normal run.

More specifically, we describe a few concrete scenarios showcasing how an enclave program can be abused once compromised (or after a private key is leaked). First, a botnet creator can establish a completely hidden or obfuscated communication channel between its operator by busing the remote attestation [13]; once malware runs inside the enclave, the operation initiates actual commands.

Second, the isolation will make traditional, popular signature-based anti-virus programs futile; one potential direction is to estimate the correct or expected behavior of enclave programs, but we believe these directions are ad-hoc, (i.e., not sound) and results in huge false positives in practice.

Finally, another concern, similarly raised by Joanna Rutkowska [39, 40], is that the use of SGX tends to make end-users rely too strongly on Intel. Except for the fact that Intel can launch an enclave without a platform-specific key, our reliance on the SGX might result in a single point of security failure; Intel can introduce a CPU backdoor by disclosing the private key without any hardware tempering or incorporating suspicious components.

X. Conclusion

Due to the wide adoption of the x86 architecture, its Software Guard Extensions (SGX) for trusted execution potentially has a tremendous impact on software security, enabling a wide range of applications to enhance their security and privacy properties. At the same time, the limitations of SGX need to be closely evaluated due to the possibility of new forms of attacks potentially surfacing. Unfortunately, the absence of an open platform for research, such as SGX hardware or an emulator, has been a significant barrier to exploring the promises and potential issues of SGX. The remarkable interest we have received from the security community during the early phase of development reflects that there is a strong demand for an open platform for SGX research.

To tackle the problem, we propose OpenSGX, a fully functional open source emulator for Intel SGX. In the process, we have developed a complete platform for SGX development that includes emulated hardware and operating system components, an enclave program loader, an OpenSGX user library, and debugging and performance monitoring support. Our evaluation of OpenSGX demonstrates that it can run non-trivial applications, such as the Tor anonymity network, and new ideas can be easily implemented and evaluated as a proof-of-concept using our framework. Finally, we believe that significant research opportunities exist in applying new ideas to each and every component of OpenSGX. We plan to make OpenSGX publicly available as open source and hope that OpenSGX serves as a vehicle for implementing new ideas in trusted execution environments.

XI. Acknowledgment

We thanks Patrick Bridges for implementing enclave library and remote attestation supports, Ron Rivest for insightful discussion, Jethro Beekman for checking hardware conformance with OpenSGX, and the anonymous reviewers for their helpful feedback. This research was supported in part by the NSF award (DGE-1500084); by the ONR grant (N00014-15-1-2162); by the DARPA Transparent Computing program under contract No. DARPA-15-15-TC-FP-006; by the ICT R&D program, MSIP/IITP [R-20150223-000167, R0190-15-2010, H7106-14-1011, 14-911-05-001]; and by NRF-2013R1A1A1076024.

References

[1] T. G. Abbott, K. J. Lai, M. R. Lieberman, and E. C. Price. Browser-based attacks on tor. In *Proceedings of the 7th International Conference on Privacy Enhancing Technologies*, 2007.

[2] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, pages 1–8, Tel-Aviv, Israel, 2013.

[3] ARM. Building a secure system ising trustzone technology, Dec. 2008. PRD29-GENC-009492C.

[4] A. M. Azab, P. Ning, and X. Zhang. SICE: A hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 375–388, Chicago, Illinois, Oct. 2011.

[5] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th Symposium on Operating Systems*

Design and Implementation (OSDI), pages 267–283, Broomfield, Colorado, Oct. 2014.

[6] S. L. Blond, P. Manils, C. Abdelberi, M. A. D. Kaafar, C. Castelluccia, A. Legout, and W. Dabbous. One bad apple spoils the bunch: exploiting p2p applications to trace and profile tor users. *arXiv preprint arXiv:1103.1518*, 2011.

[7] R. Boivie and P. Williams. SecureBlue++: CPU support for Secure Executables, 2013. RC25369, IBM Research Report.

[8] F. Brasser, B. E. Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl. TyTAN: tiny trust anchor for tiny devices. In *Proceedings of the 52nd Annual Design Automation Conference (DAC)*, 2015.

[9] E. Brickell and J. Li. Enhanced privacy ID: A direct anonymous attestation scheme with enhanced revocation capabilities. In *Proceedings of the 2007 ACM workshop on Privacy in electronic society*, pages 21–30, 2007.

[10] P. Carbin. Intel Identity Protection Technology with PKI (Intel IPT with PKI), May 2012. White Paper, Technology Overview.

[11] S. Checkoway and H. Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 253–264, Houston, TX, Mar. 2013.

[12] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–13, Seattle, WA, Mar. 2008.

[13] S. Davenport and R. Ford. SGX: the good, the bad and the downright ugly, Jan. 2014. https://www.virusbtn.com/virusbulletin/archive/2014/01/vb201401-SGX.

[14] R. Dingledine. Tor Project infrastructure updates in response to security breach. http://archives.seul.org/or/talk/Jan-2010/msg00161.html, January 2010.

[15] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *USENIX Security Symposium*, 2004.

[16] L. Duflot, Y.-A. Perez, and B. Morin. What if you canâĂŹt trust your network card? In *Recent Advances in Intrusion Detection*, pages 378–397. Springer, 2011.

[17] FortConsult. Practical Onion Hacking: Find the Real Address of Tor Clients. http://www.fortconsult.net/images/pdf/Practical_Onion_Hacking.pdf, 2006.

[18] J. Greene. Intel trusted execution technology. *Intel Technology White Paper*, 2012.

[19] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, pages 1–8, Tel-Aviv, Israel, 2013.

[20] Intel. Graphics Drivers Blue-ray Disc* Playback On Intel Graphics FAQ. http://www.intel.com/support/graphics/sb/CS-029871.htm#bestexperience, 2008. Accessed: 05/04/2015.

[21] Intel. Intel Software Guard Extensions Programming Reference (rev1), Sept. 2013. 329298-001US.

[22] Intel. Intel Software Guard Extensions Programming Reference (rev2), Oct. 2014. 329298-002US.

[23] Intel. Product change notification, Oct. 2015. PCN114074-00.

[24] Intel. SGX Tutorial, ISCA 2015. http://sgxisca.weebly.com/, June 2015.

[25] A. Ionescu. Intel sgx enclave support in windows 10 fall update. *Winsider Technical White Paper*, 2015.

[26] A. Johnson, C. Wacek, R. Jansen, M. Sherr, and P. Syverson. Users get routed: Traffic correlation on tor by realistic adversaries. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 337–348. ACM, 2013.

[27] S. Kim, Y. Shin, J. Ha, T. Kim, and D. Han. 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, Nov. 2015.

[28] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, 2009.

[29] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. Trustlite: a security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*, page 10. ACM, 2014.

[30] S. Le Blond, P. Manils, A. Chaabane, M. A. Kaafar, C. Castelluccia, A. Legout, and W. Dabbous. One bad apple spoils the bunch: Exploiting p2p applications to trace and profile tor users. In *Proceedings of the 4th USENIX Conference on Large-scale Exploits and Emergent Threats*, pages 2–2, 2011.

[31] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–14, Boston, MA, Apr. 2009.

[32] Y. Li, J. M. McCune, and A. Perrig. VIPER: verifying the integrity of PERipherals' firmware. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 3–16. ACM, 2011.

[33] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry. MiniBox: A Two-Way Sandbox for x86 Native Code. In *Proceedings of the 2014 ATC Annual Technical Conference (ATC)*, pages 409–420, Philadelphia, PA, June 2014.

[34] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proceedings of the ACM EuroSys Conference*, pages 315–328, Glasgow, Scotland, Mar. 2008.

[35] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of the 31th IEEE Symposium on Security and Privacy (Oakland)*, pages 143–158, Oakland, CA, May 2010.

[36] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, pages 1–8, Tel-Aviv, Israel, 2013.

[37] S. J. Murdoch. Introduction to trusted execution environments (tee), 2014. http://sec.cs.ucl.ac.uk/users/smurdoch/talks/rhul14tee.pdf.

[38] E. Owusu, J. Guajardo, J. McCune, J. Newsome, A. Perrig, and A. Vasudevan. OASIS: On achieving a sanctuary for integrity and secrecy on untrusted platforms. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, pages 13–24, Berlin, Germany, Oct. 2013.

[39] J. Rutkowska. Thoughts on Intel's upcoming Software Guard Extensions (Part 1), Aug. 2013. http://theinvisiblethings.blogspot.com/2013/08/thoughts-on-intels-upcoming-software.html.

[40] J. Rutkowska. Thoughts on Intel's upcoming Software Guard Extensions (Part 2), Sept. 2013. http://theinvisiblethings.blogspot.com/2013/09/thoughts-on-intels-upcoming-software.html.

[41] Samsung. White Paper: An Overview of Samsung KNOX, 2013. Enterprise Mobility Solutions.

[42] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy Data Analytics in the Cloud using SGX. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[43] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani. Moat: Verifying confidentiality of enclave programs. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1169–1184. ACM, 2015.

[44] L. Soares and M. Stumm. Flexsc: flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–8. USENIX Association, 2010.

[45] U. Steinberg and B. Kauer. NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the ACM EuroSys Conference*, pages 209–222, Paris, France, Apr. 2010.

[46] K. Sun, J. Wang, F. Zhang, and A. Stavrou. SecureSwitch: BIOS-assisted isolation and switch between trusted and untrusted commodity oses. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2012.

[47] Y. Sun, A. Edmundson, L. Vanbever, O. Li, J. Rexford, M. Chiang, and P. Mittal. Raptor: routing attacks on privacy in tor. In *USENIX Security Symposium*, 2015.

[48] Torproject. The chutney tool for testing and automating Tor network setup. https://gitweb.torproject.org/chutney.git/tree/README, 2015. Accessed: 05/15/2015.

[49] A. Vasudevan, E. Owusu, Z. Zhou, J. Newsome, and J. M. McCune. Trustworthy execution on mobile devices: What security properties can my mobile platform give me? In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing (TRUST)*, pages 159–178, Vienna, Austria, 2012.

[50] Wikipedia. C dynamic memory allocation — wikipedia, the free encyclopedia, 2015. URL http://en.wikipedia.org/w/index.php?title=C_dynamic_memory_allocation&oldid=658580417. [Online; accessed 13-May-2015].

[51] P. Williams and R. Boivie. CPU support for Secure Executables. In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing (TRUST)*, pages 172–187, 2011.

[52] P. Winter, R. Köwer, M. Mulazzani, M. Huber, S. Schrittwieser, S. Lindskog, and E. Weippl. Spoiled onions: Exposing malicious tor exit relays. In *Privacy Enhancing Technologies*, pages 304–331. Springer, 2014.

[53] P. Winter, R. Köwer, M. Mulazzani, M. Huber, S. Schrittwieser, S. Lindskog, and E. Weippl. Spoiled onions: Exposing malicious tor exit relays. In *Privacy Enhancing Technologies*, pages 304–331. Springer, 2014.

[54] F. Zhang. IOCheck: A framework to enhance the security of I/O devices at runtime. In *Dependable Systems and Networks Workshop (DSN-W), 2013 43rd Annual IEEE/IFIP Conference on*, pages 1–4. IEEE, 2013.