D.1 Introduction

Background. Trusted computing, or the Trusted Execution Environment (TEE), is the foundational technology that ensures confidentiality and integrity in modern computing. Over the past few decades, a considerable amount of research has been carried out to search for practical ways to achieve trusted computing, e.g., using a formally verified operating system (OS) [40], virtual machine monitor (VMM) or hypervisor [15, 46, 49, 67, 69, 84], system management mode (SMM) [75], and BIOS [70], or hardware assistance [48]. Increasingly, hardware technologies for TEE (e.g., TPM [57], ARM Trusted Zone [63], Intel TXT, and AMD SVM [73])—already widely deployed in consumer and enterprise products— have rapidly matured, especially with the commoditization of Intel Software Guard Extensions (SGX) [3, 32, 50], which became market available in August 2015 in Intel's Skylake CPU.

At a high level, Intel SGX allows an application or part of an application to run inside a secure *enclave*, which is an isolated execution environment. Intel SGX hardware, as a part of the CPU, protects the enclave against malicious software, including the operating system, hypervisor, or even low-level firmware code (e.g., SMM) from compromising its integrity. This *isolation* enabled by Intel SGX is particularly useful in cloud computing environments, where customers cannot control the infrastructure owned by cloud providers. Consequently, initial exploration into secure execution has begun for cloud settings. Haven [4] pioneered the idea of enabling unmodified application binaries to run on Intel SGX by utilizing an OS library [58]. VC3 [65] suggested privacy-aware data analytics in the cloud. As witnessed by these early efforts, we believe that Intel SGX can be used to achieve unprecedented security for many cloud applications (and beyond), thereby safely taking advantage of the flexibility, elasticity, and economy-of-scale provided by cloud infrastructures.

At a low level, Intel SGX is fundamentally an extension to the x86 instruction set architecture (ISA) that enables an application to instantiate one or more isolated enclaves. SGX is implemented largely as two components within a regular CPU: a memory controller that regulates external memory accesses and also performs encryption/decryption of their contents; and an attestation engine that is mainly in charge of remote attestation and sealing of enclaves, which enables verification of enclave integrity. Accordingly, Intel SGX consists of two large sets of new instructions; those utilizing memory management for *isolation*, and those providing security for *attestation*, marked respectively as `MEM` and `SEC` in Table 1).

P	Type	Instruction	Description	V
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 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 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 page to a new location	r2

Table 1: SGX Instruction Overview. P: Privileged (ring 0) instructions; U: User-level (ring 3) instructions; V: Version; r1: Revision 1 [35]; r2: Revision 2 [36]; MEM: Memory management related; EXE: Enclave execution related; SEC: Security or permissions related.

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 each memory access, such that only code inside the enclave can access its own enclave region. That is, memory access from both non-enclaves and different enclaves is prohibited. Note that the memory access policy on non-enclave regions remains the same, i.e., a 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 the Enclave Page Cache (EPC). To defend against known memory attacks such as memory snooping, memory content in the EPC is encrypted by the Memory Encryption Engine (MEE). The memory content in

the EPC is decrypted only when entering the CPU package, where the code and data are protected by the *enclave mode*, and then re-encrypted when leaving the CPU back to the EPC memory region.

For programmers to use these hardware features, SGX introduces a set of instructions (Table 1) and data structures (Table 2) to support enclave and EPC-related operations. Instructions are classified into user-level instructions (ring 3) and privileged instructions (ring 0). The families of user-level and privileged instructions are called `ENCLU` and `ENCLS`, respectively. For example, the user-level instruction `EENTER` allows the host program to transfer control to an existing enclave program, while `ECREATE` is a privileged instruction that allocates available EPC pages for a new enclave.

Instruction		Description
`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 2: Hardware Level Data Structure in Intel SGX.

Motivation. The security of SGX programs does not come for free, but rather depends upon correct usage of SGX instructions. Unfortunately, it is non-trivial for programmers to correctly use such instructions. Worse yet, there is no way to guarantee or validate the correctness of SGX programs. Application programmers should therefore avoid using individual instructions, and instead utilize higher-level abstractions that thoroughly handle the security pitfalls or subtleties of the SGX programming. Although SGX is being integrated into commodity hardware, APIs and system infrastructure that are designed to utilize SGX need to be developed. This includes operating system support, packaging and distribution, debugging and monitoring, and even toolchains for the development of SGX programs. For example, many SGX instructions require kernel-level privileges (ring 0), but system call interface and operating system service/support for SGX has not yet been implemented.

While it may appear that Intel should have led such efforts to develop systems and software support, we note that there is a large demand for a community-driven alternative. If we consider the development and adoption of Intel VT-x, we notice that there are many successful open platforms built around these technologies, such as the Xen and KVM projects. At the same time, closed-source solutions, such as Microsoft HyperV and VMware vSphere, also exist. Because commercial hardware or software vendors might not always be willing to release source code utilizing SGX, and recognizing the benefit of open-source systems utilizing hardware-based ISA extensions, we believe there is a need to develop SGX support in an open manner. We believe an open-source ecosystem surrounding SGX will foster continued research by the security community, and ensure that these extensions are used to further secure x86-based computing environments.

SGX enclave programs will certainly be attacked from various vectors, such as a malicious OS, side channel attacks, or remotely exploitable vulnerabilities. In particular, since SGX programs will be running in a completely hostile environment, it is unclear how to defeat malicious operating system attacks such as system call level attacks (e.g., Iago [14] attack), the Cuckoo attack [56], or controlled side channel attacks [80]. Furthermore, SGX-based bugs are critical in terms of security, and the compromise of an enclave program becomes more critical than before. Also, SGX makes it impossible to analyze or even detect such compromises, and for end users a compromised execution is completely indistinguishable from a correct execution. Despite recent progress in secure software engineering, especially the use of formal methods to rigorously identify software vulnerabilities, it is still not possible to build bug-free software. Therefore, SGX programs will not be vulnerability free and could still be exploited. Unfortunately, traditional mitigation techniques such as ASLR will be incompatible with SGX programs because randomizing the layout of code can lead to attestation failure. As such, it is also imperative to investigate how to secure the SGX programs—especially from the enclave itself—since they cannot trust the underlying OS.

SGX can also be abused by malicious software, especially considering the fact the SGX enclave code will be invisible to the operating system. Recently, there were discussions surrounding the development of malware that can create enclaves and execute code that is undetectable and unanalyzable due to SGX blackboxing [32, 61, 62]. For example, once a system is compromised, malware can exploit Intel SGX-provided functionalities to create a botnet [22]. As such, it is unclear how the detection of malicious software running in the enclave should be performed. For example, if malware is running in the enclave, when and how does an OS kill the enclave program? Since the enclave program will appear as a giant block of code to the OS, what kind of accountability (e.g., the number of executed system calls, the number of used EPC pages, the CPU time slice) should an OS provide regarding the enclave program execution? We would like to answer these questions in this proposed research.

Research Objectives. This proposed research aims to fill the gap between what it needs from end-users, system administrators, and application developers, and what the low-level SGX instructions provide. Our approach will enable end-users and administrators to safely utilize SGX hardware and allow programmers to develop a secure enclave program without worrying about the subtle corner cases that SGX inherently contains. To be more specific, we plan to explore three research thrusts:

- **R1. Developing abstractions, libraries, and tools in support of SGX execution**. Making SGX practical requires end-to-end support from operating systems, compilers, loaders, debuggers, packaging, distributions, and programming abstractions. We will devise the best practical approach by precisely studying the pros and cons of existing alternatives, and openly develop our idea for the GNU/Linux platform. This result will not only enrich the general understanding of trusted computing and its adoption in SGX, but also contribute to the open source community.
- **R2. Securing enclave programs**. We will formulate concrete threat models of SGX programs so we can build SGX infrastructure and libraries. Specifically, we will investigate a self-defense scheme that guarantees the security of enclave programs under an adversarial OS and meanwhile investigate how to materialize traditional defense mechanisms (e.g., ASLR) for enclave programs.
- **R3. Defending against SGX malware**. Since it is impossible to trace or analyze the execution of the enclave program, malicious software can also abuse SGX to protect itself. We will explore potential accounting schemes that monitor the execution of enclave programs at different levels of granularities, thereby helping users safely maintain running enclave programs. We also plan to devise a practical policy-based scheme that allows developers to specify the runtime behavior of an enclave program.

Research Outcomes. The overall outcome we anticipate from this project is to produce systems, tools, libraries, and techniques that support and manage SGX program execution, ultimately fostering an initial movement from the open source community toward building a healthy ecosystem to secure SGX programs. As part of our effort, we will integrate our software design and artifacts into the GNU/Linux platform, and most importantly share our insights, experiment results, and software itself in top-tier systems and security venues.

Broader Impacts. This research will develop a toolchain and systems support to enable GNU/Linux application programmers to securely use SGX, while also providing a holistic solution to secure SGX programs against various attacks from a hostile OS as well as remote vulnerability exploits. It will also defend against the malicious use of SGX. These results are not only applicable to the broader scope of software development in general, but also are of interest to security, operating systems, compilers, and architectures. In addition to the broader academic impact, the proposed work is also of particular interest to the open source community as well as to the systems and security industry. We have included a collaboration letter from Intel, indicating their strong interest in this proposed research.

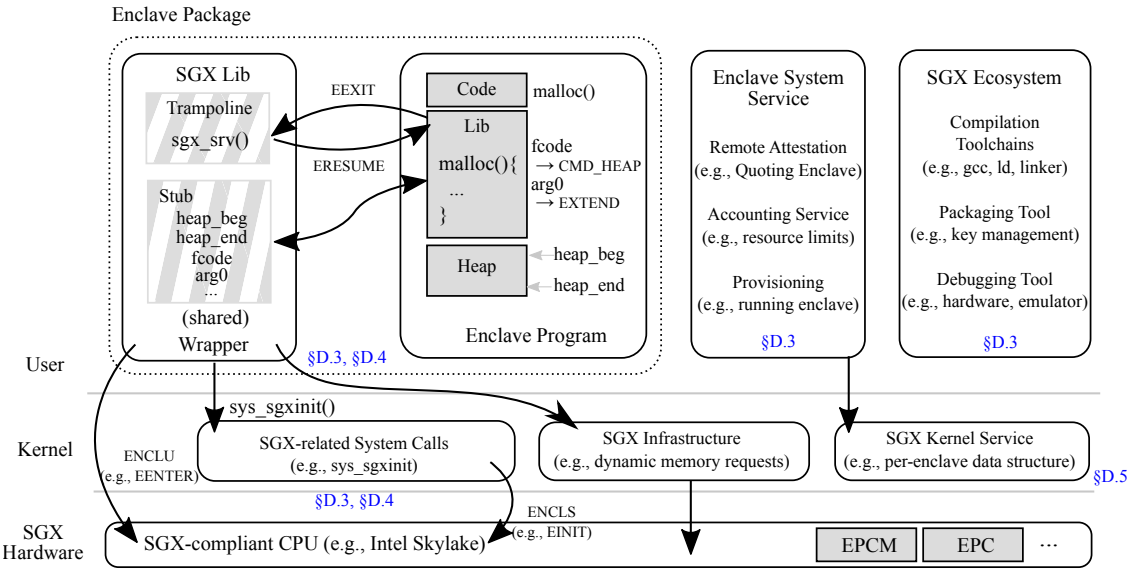

Figure 1: An Overview of the Systems and Software Support to Execute and Manage SGX Programs.

The proposed research results will be integrated into the trusted computing curriculum with lecture notes, hands-on labs, and online tutorials. These materials will be made freely available to the public. The PIs are also engaged in outreach activities, including partnering with the local (HBCU) colleges, organizing cybersecurity workshops in K-12 summer camps, organizing SGX workshops in academic conferences, and collaborating with systems and security companies.

D.2 Overview of the Proposed Research and Our Preliminary Result

This proposed research has three main goals: (1) enabling SGX program execution and management by building the corresponding systems and software support; (2) investigating techniques to secure the SGX programs running in the hardware protected enclave; and (3) defending against the malicious use of SGX.

Figure 1 shows our preliminary design of the SGX ecosystem, which consists of six major components (three located in user space and three located in kernel space) required to execute and manage enclave programs. An enclave package in user space contains the enclave program as well as the SGX runtime library that wraps the low-level SGX instructions and provides a trampoline for the enclave program. This runtime library directly interacts with the hardware, the OS kernel SGX-related system calls, and the SGX memory management infrastructures such as EPC page allocation and loading. SGX system calls will provide the system-call-level abstractions to initiate the enclave, add the EPC page, etc. There is also an enclave system service in user space that provides various functionalities regarding enclave program execution, such as provisioning, accounting, and remote attestation (e.g., launching a Quoting Enclave [32]). This enclave system service interacts with the OS kernel service. The last user space component is the SGX Ecosystem, which contains various tools for SGX program creation, packaging, and debugging. We present the design of these components in §D.3.

While SGX provides for the detection of integrity and confidentiality violations of code and data for an enclave program, we envision that there will be various attacks against SGX enclave programs. One instance is the recently demonstrated controlled side channel attack [80], in which an attacker can infer application behavior by controlling and observing the page fault of an application. There will also be other attacks from malicious OSes, such as the Iago attack [14] and Cuckoo attack [56]. Meanwhile, enclave programs may still contain conventional memory corruption bugs (e.g., buffer overflow) that attackers can exploit. We will explore security mechanisms to ensure the safety of enclave programs by strictly following two principles: 1)

security by construction and 2) the principle of least privilege. The details of our proposed research in this direction are presented in §D.4.

SGX is designed to prevent privileged software from analyzing the enclave program. Malicious software will likely use SGX to defeat malware analysis [22, 62]. Meanwhile, when SGX is introduced to the desktop ecosystem, keyloggers and screen readers will trivially sniff enclave programs' secrets. Therefore, in this proposed research we would also like to investigate how to detect the malicious use of SGX. We will look into both anomaly detection and misuse detection, and design new threat models regarding SGX malware. We will also develop support for the OS kernels to incorporate other possible trusted software or hardware components to secure I/O in a desktop environment. We provide our research plan for defending against malicious use of SGX in §D.5.

Preliminary Result. Both PI Kim and PI Lin are among the pioneers of SGX research. In the past a few years, PI Kim has led the efforts to develop OpenSGX (`https://github.com/sslab-gatech/opensgx`), which is an open source emulator of the new SGX instruction set and allows researchers to evaluate their prototypes when using the SGX platform [1]. As of today, OpenSGX has been used by more than 20 research groups worldwide including the group led by PI Lin. OpenSGX has built a strong foundation in developing systems and software support for SGX program execution.

PI Lin also has been working on investigating SGX for new security applications. In particular, PI Lin is among the first few research groups to have full access to the SGX platform from Intel, and he has led efforts to investigate the analysis of SGX enclave programs, especially for malware analysis applications. He also has looked into securing cryptographic keys using SGX, as well as protecting other secrets such as gaming data in enclave programs.

PI Kim and PI Lin also have been collaborating to develop SGX use cases. One such a use case is to secure a TOR network node by using OpenSGX to protect cryptographic keys [2]. Another is to study the system-level behaviors of enclave programs.

More importantly, the knowledge and early experiences from building the basic development platform for the SGX emulator and designing their applications have equipped the PIs with a deep understanding of SGX's advantages and disadvantages. Based on these experiences and strong support from Intel, we believe our team can advance the current state of the art in secure SGX execution, management, and defense.

D.3 R1: Developing Systems and Software Support for SGX Program Execution

Proper use of SGX requires support from the OS kernel with corresponding system calls and management services (§D.3.1), user-level runtime libraries and service routines (§D.3.2), and enclave program development toolchains (§D.3.3). In this section, we describe the justification of our design decisions with technical challenges and our initial ideas to overcome them in the proposed work.

D.3.1 R1.1 Operating Systems Support

Since SGX includes a number of privileged instructions (e.g., `EINIT`), they must be executed by OS kernels. This implies that an operating system, an untrusted entity with respect to SGX, must be involved to provide a service (e.g., through system calls) to use these instructions.

These privileged instructions are necessary for an OS to initiate an enclave, allocate enclave pages (EPC pages), and maintain the address mapping (e.g., fetching EPC information in the BIOS and maintaining the related page table). We also believe the OS needs to support quality of service (QoS) for enclave programs. For example, how can we allow users to kill a suspicious enclave program when it is so privileged that the user cannot see its execution? The OS should provide an interface for users to enforce policy and keep track of

[1] The details describing how OpenSGX works are currently under submission.

[2] The academic paper describing the details of protecting a TOR node is also under submission.

Instruction	Description

`int sys_init_enclave(`
`  void *base_address,`
`  unsigned int n_of_pages,`
`  tcs_t tcs,`
`  sigstruct_t *sig,`
`  einittoken_t *token)`
	Allocate, add, measure 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

	Related leaf commands: `ECREATE`, `EADD`, `EEXTEND`, `EINIT`
`void sys_terminate_enclave(` `  int keid)`	Terminate an enclave Enclave ID (maintained inside the kernel)
	Related leaf commands: `EREMOVE`
`epc_t sys_add_epc(` `  int keid)`	Allocate a new EPC page to the running enclave. Enclave ID (maintained inside the kernel)
	Related leaf commands: `EAUG`

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

	Related leaf commands: `N/A`

Table 3: List of potential system calls to support, execute, and manage enclave programs in commodity operating systems.

each enclave instance. In addition to these management issues, a set of other features should be implemented for proper functionality, such as dynamic EPC page allocation and expansion. Our team first devised a list of four system calls to execute, terminate, and maintain enclave programs, as summarized in Table 3.

(A) **Bootstrapping.** Not all memory pages can be used as EPC. When booting the kernel, the OS needs to know the mapping of enclave pages; a user can configure how many EPC pages can be allocated via BIOS. Upon booting, the OS kernel obtains a contiguous chunk of EPC and its physical address. Then, the OS can use the EPC region to initialize an enclave.

(B) **Enclave initialization.** Intel SGX provides four instructions related to enclave initialization: `ECREATE` (create an enclave), `EADD` (add a page), `EEXTEND` (extend EPC page measurement), and `EINIT` (initialize an enclave). Since system call numbers have a limited maximum value, we should introduce as small a number of system calls as possible. Therefore, we propose to design one system call, namely, `sys_init_enclave()`, to initiate, extend, and measure an enclave program that requires privileged SGX instructions.

(C) **Dynamic EPC page allocation.** The Intel SGX revision 2 [4, 36] introduces a mechanism to dynamically expand enclave memory by using `EAUG` and `EACCEPT`. Based on these two instructions, we propose to provide `sys_add_epc()` to dynamically allocate additional EPC pages for an enclave that requires more memory. When an enclave needs a new EPC page, the OS allocates a free EPC page via `EAUG`. Then, the enclave should invoke `EACCEPT` to accept the new page into its own enclave region.

Research Challenge 1. *A hostile OS can attack the enclave program by controlling EPC page allocation and deallocation. How can we defeat this attack?*

We propose to perform an integrity check upon the execution of the `EACCEPT` instruction (recall we will provide wrappers to all the user-level SGX instructions), to thwart such an attack. The details of this integrity check are presented in §D.4.1.

(D) **Enclave Program Execution Accounting.** Finally, the operating system should be in charge of authorization, fairness, and execution of requested enclave programs to fully take advantage of SGX features. For instance, to support multiple enclaves concurrently, the OS kernel needs to maintain a per-enclave structure that describes the execution context of each enclave. This would include the enclave ID and

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

Table 4: List of APIs to be implemented in `sgxlib`.

the contents of `TCS` and the stack size, similar to the `task_struct` in Linux. The structure also needs to contain debugging and accounting information (e.g., the number EPC page allocated, the CPU time slice executed, etc.).

Research Challenge 2. *What is the best way to provide contextual information about enclave executions to users? How effective is the information in determining abnormal behavior of enclave executions, and how efficiently can our infrastructure keep track of them?*

As such, we propose to design a `sys_stat_enclave()` system call for this purpose. When an enclave is created, we need to keep track of the new enclave by assigning an identifier (`keid`) in the kernel 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 utility application can later query the collected accounting information through `sys_stat_enclave()`.

D.3.2 R1.2 Runtime Libraries and User Level Enclave System Service Support

We also anticipate there is a need for a system runtime library that provides necessary functions for SGX programmers to use and execute enclave programs. Therefore, we propose to design such a library (called `sgxlib`) with the goal of (1) facilitating enclave programming and (2) minimizing the attack surface between the enclave and its *potentially malicious* host process. Table 4 lists APIs that need to be implemented by `sgxlib`. Below we describe how we should design the `sgxlib` and its security considerations.

Research Challenge 3. *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 a new attack surface (e.g., a malicious host can return a crafted input to the enclave).*

We propose to 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()`). While this is an engineering challenge, we plan to utilize an automatic source code instrumentation tool (e.g., LLVM [43] or CIL [52]) to speed up our development.

Research Challenge 4. *Although an enclave can legitimately access memory shared outside the enclave, this is not a recommended practice since a malicious host or operating system can potentially modify non-enclave memory. Consequently, how do we handle non-enclave memory for an enclave program?*

We propose a stricter form of communication protocols 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 that is tractable for enforcing the associated security properties.

More specifically, our stricter communication is one-way and entirely driven by the requesting enclave. For example, to request a socket for networking (see Figure 2), the enclave first sets up the input parameters in *stub* (e.g., sets `fcode` to `FSOCKET` in Figure 2), and then invokes a predefined handler, *trampoline*, by exiting *enclave mode* (i.e., by invoking `EEXIT`). Once the OS processes the enclave request, it stores the result or return values to `stub`, and enters *enclave mode* 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*).

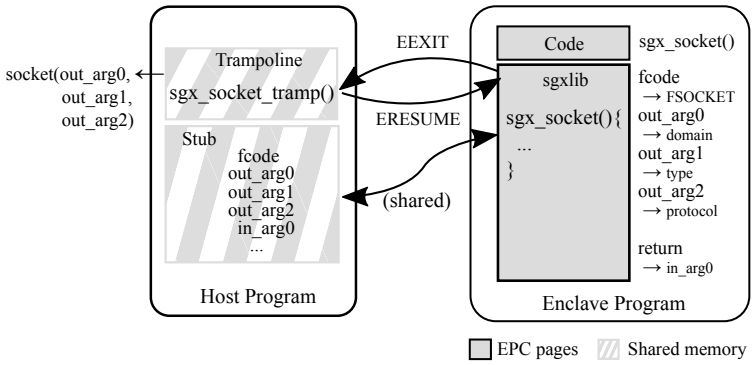

Figure 2: Interface defined for communicating with the non-enclave host program that performs the delegated calls to the operating system. In this figure, an `sgxlib` function, `sgx_socket()`, running inside the enclave, requests a socket system call via *trampoline* and *stub*, which are pre-negotiated between the enclave and its wrapper when packaged together.

Dynamic Memory Allocation. In addition to the runtime libraries, we also need to design and implement other user libraries necessary for the execution of enclave programs. One such a library is for the handling of dynamic memory allocation. Although it is permissible for an enclave program to use dynamically allocated user memory, it can break the enclave isolation feature. To avoid this, we propose a customized dynamic memory allocation API, `sgx_malloc()`, that behaves similarly to glibc `malloc()` [79], but that only allocates memory from the enclave heap (pre-allocated EPC pages, see Figure 3). `sgx_malloc()` manages the enclave heap by maintaining heap pointers, which are initially set to the heap with the aid of the OS during the first initial `sgx_malloc()` call. When the pre-allocated heap area becomes full, `sgx_malloc()` leverages dynamic EPC page allocation (via `sys_add_epc()`) to extend the enclave heap. With `EAUG`/`EACCEPT`, 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 trick the enclave into adding another EPC page.

Remote Attestation. We also need to support remote attestation, a crucial execution step in SGX. To this end, we propose an API `sgx_remote()` with which programmers can generate a remote attestation request in an enclave program. 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 serve a special enclave (called a "quoting enclave") that verifies a target enclave through intra-attestation, we also need to provide `launch_quoting_enclave()`. The overall procedure of remote attestation will be implemented based on the SGX specification [3].

D.3.3 R1.3 SGX Toolchain and Ecosystem Support

The final systems component for SGX is the toolchain for application development. Currently, there is no official document of the SGX application binary interface. The toolchain we propose includes a compiler, loader, debugger, and a new packaging tool packer.

Compiler. We will not directly modify any existing code base of mainstream compilers such as `gcc` or `clang`. Instead, we will use them to compile an enclave program. Because we will provide wrappers to SGX instructions, as long as we eventually link to these wrappers (whose object code is produced by the Intel assembler) we do not have to modify the compilers. However, we do have to take special care while compiling an enclave program. Specifically, one key feature of an SGX binary is that it should be easily relocated to EPC. According to the SGX specification, the `EADD` instruction loads code and data into the EPC by direct memory copying, which implicitly assumes that developers take care of program relocation by themselves. To ease developers' efforts in handling program relocation, we propose a build script to automatically adjust compilation options to make enclave code and data easily relocatable at runtime.

Loader. An SGX binary loader needs to determine the memory layout of code, data, stack, heap, and necessary data structures on the EPC region during initialization of an enclave. Similar to 'loader' in the OS, our SGX binary loader should obtain 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, and other necessary data structures (see Figure 3). Then, our SGX binary loader forwards the memory layout information to the `sys_init_enclave` system call to initiate the enclave.

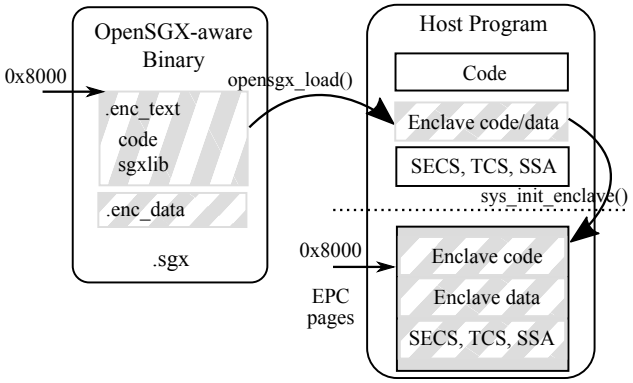

Figure 3: Loading process performed by our loader. First, `.enc_text` and `.enc_data` sections are loaded in to host memory. The SGX loader then forwards two sections along with stack, heap, and other necessary data structures to EPC via `sys_init_enclave()`.

Research Challenge 5. *SGX requires that application code and data be placed on an enclave page cache (EPC), a reserved, encrypted area of memory, and execution must stay within EPC. When executing a binary on EPC, an SGX instruction can allow one to copy a normal page onto an EPC page. Therefore, we have to enable our dynamic loader to provision the code, data, and stack sections on EPC and to also handle the EPC relocation appropriately. This is non-trivial. Meanwhile, such dynamic code loading into the enclave also introduces obstacles for remote attestation. We have to address these additional challenges.*

Debugger. To debug an enclave program, Intel SGX CPUs have hardware support for a single step of an enclave program. Therefore, we need to make `gdb` (our base platform) aware of this feature. To this end, in addition to enabling the single step inside an enclave, we propose to modify `gdb` and introduce four new commands. These 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`.

SGX Binary Packer. Once the binary is compiled and linked with our library, we also need to generate other help files such as the hash of the code, which will be used for attestation measurement. To this end, we need to develop an SGX binary packing tool. It will generate a configuration file (a .conf file) that contains the program measurement (a hash value), a signature signed by a specified RSA key, and other enclave properties that are required to validate the program during enclave initialization.

We also expect that it will be helpful to specify the system calls (e.g., the system call name, parameters, etc.) that will be used by the enclave program in the configuration file. Much like smartphone apps that ask for permissions [25, 26, 38], if the enclave can explicitly ask for permission to execute system calls, this will allow the OS to enforce security policies. This is especially helpful when considering that enclave programs

Type	Interface	Attack surfaces	In-enclave usage/check
MEM	sgx_malloc() → <out>addr	1) incorrect pointers, 2) incorrect EPC add.	EACCEPT verifies the status of a new EPC
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 wo/ trusted I/O
TIME	sgx_time() → <out>time	1) arbitrary time	Validate time from the NTP server
RAND	sgx_rand() → <out>rand	1) arbitrary value	Relying on rdrand inst
IO	sgx_write() → <out>len	1) arbitrary reported len	No general way to prevent wo/ 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)
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 5: Consideration of Iago attack in primitive `sgxlib` functions that are to be implemented by using the shared trampoline between OS 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).

can be attacked, and that there could be malware running in an enclave. We plan to develop automatic techniques to derive the system call specifications from the enclave programs and enforce them with system call level sandboxing [29, 59, 74]. More details are discussed in §D.5.

D.4 R2. Securing SGX Enclave Programs

Research Challenge 6. *An enclave program inevitably must interact with the underlying OS to request the services it provides (e.g., system calls), as well as the resources it manages (e.g., virtual pages, random numbers). This makes the enclave programs particularly vulnerable to a number of layer-below attacks. In particular, enclave programs rely on the support of an underlying OS, but the security model of the enclave programs excludes the OS from the TCB. Such an unconventional dependency makes various attack vectors, e.g., Iago and side-channel attacks, which are often considered impractical in a traditional setting, immediate and practical, especially in a cloud environment.*

In this research thrust, we would like to revisit how to defeat these attacks. Within the context of SGX, we want to defeat Iago attacks (§D.4.1) and controlled side channels (§D.4.2), and also achieve secure I/O under a hostile environment (§D.4.3). In addition, enclave programs can contain exploitable vulnerabilities such as buffer overflows, and typical threats and attack vectors still exist. We would like to investigate techniques to mitigate typical vulnerability exploitation as well (§D.4.4).

D.4.1 R2.1 Defeating Iago Attacks

To provide enclaves the ability to communicate with the underlying OS, it is unavoidable to introduce additional attack surfaces. Iago attacks [14], in which the operating system or the underlying hypervisor (e.g., [28, 39]) manipulates the return values and actions performed by the system call, are introduced when enclaves interact with the OS. However, by taking precautions inside an enclave, potential attacks can be prevented. In our research, we intend to prevent Iago attacks by systematically protecting the enclave from malicious system call results using the enclave library `sgxlib`. Specifically, we propose to defend against Iago attacks in three broad aspects: dynamic memory allocation, network and I/O, and non-determinism/resources. Note that Intel SGX does not prevent denial-of-service attacks or guarantee strong privacy (e.g., IP address), but provides strong isolation and confidentiality. With this in mind, as outlined in Table 5, we inspect the potential attack surface in communication between the enclave program and underlying kernel and discuss self-defenses from the enclave program itself in each category of attack.

- **Memory-related operations (marked `MEM`):** Since SGX revision 2, enclave programs can dynamically request EPC pages at runtime, which would provide the operating system with a potentially powerful attack vector. However, Intel SGX takes this into account and provides the `EACCEPT` instruction, which performs basic validation on newly assigned EPC pages (e.g., non-enclave pages or pre-allocated EPC pages). This thwarts a major source of memory-related attacks. We would like to investigate this feature and use it to defeat memory-related Iago attacks.
- **Network and I/O services (marked `IO`, `NET`):** Two principles are considered to prevent network- and I/O-related attacks: *encryption* and *fail-stop model*. Since a malicious OS may attempt to read or manipulate I/O from an application, to guarantee the confidentiality of packets, we believe enclave programs should encrypt all outgoing packets and validate the integrity and authenticity of all incoming packets. Upon any failure, the enclave stops its execution, which can dramatically reduce the potential attack surface in handling all errors and corner cases. We will also go into more details below about securing other I/O such as keystrokes and screen outputs in §D.4.3.
- **Non-determinism and resources (marked `DBG`, `TIME`, `RAND`):** Enclave programs often need time and randomness to provide rich experiences to users, but such data cannot simply be requested from the OS. To prevent Iago attacks, we will investigate the various ways this data can be obtained. For instance, we can fetch time values from trusted parties(e.g., an encrypted packet from known NTP servers) or randomness from a trusted CPU (e.g., the `rdrand` instruction).

D.4.2 R2.2 Defending Against Controlled Side Channel Attacks

Given the awkward TCB of enclave programs (i.e., no trust of the underlying OS), the OS has the capability to infer enclave application behavior through controlled channels. For instance, an OS can control the CPU execution of an application (e.g., the scheduling of for multi-threaded applications, which may lead to different behavior if there are dependencies among the threads), as well as the memory usage of an application (e.g., a page fault). Most recently, Xu et. al. [80] showed that analyzing an application in advance can enable an attacker to later reconstruct with surprising accuracy the data and control flow of an application by analyzing only its patterns of page faults. Since enclaves must still request memory, any applications not specifically designed to avoid leaking information in this manner may be vulnerable to revealing private information even if the entire application is placed in an enclave. To defeat such controlled side channel attacks, we plan to investigate defense from within the enclave program itself. We believe there are two basic strategies to self-defend against the controlled page fault side channel:

- **Enclave code rewriting**. Since controlled side channel attacks try to infer a user's input, it is possible to rewrite binary code such that the memory access pattern does not depend on sensitive data. Prior studies have demonstrated that with a special compiler, it is possible to mitigate the timing-based side channel [18]. We plan to use our experience in binary code rewriting (e.g., [77, 78, 82]) to defeat this attack.
- **Page fault noise injection**. The other strategy is to inject page fault noise. For instance, we can control the binary code to trigger a large volume of page fault noise that is not related to user input. We can also randomize the binary code at runtime to trigger unobserved page faults. We plan to investigate these alternatives and test their effectiveness.

There are also other potential controlled side channel attacks, e.g., the patterns in an application's system calls to the operating system. Since we cannot prevent enclave programs from using system calls, injecting system call noise may be one option to mitigate controlled side channel attacks. We plan to investigate this defense as well.

D.4.3 R2.3 Securing Keystrokes and Screen I/O

Research Challenge 7. *Intel SGX is an ideal model for the cloud, as it has a very restricted set of the I/O channels (usually just network communication). To use Intel SGX in a desktop-like environment, it is essential to es-*

tablish a secure channel between users and the enclave program via the keyboard and monitor. But such I/O is out of the scope of the TCB. How can we create a secure channel between the enclave programs and end users?

Since enclaves themselves can only run in ring 3, they are not well adapted to protecting I/O drivers. For instance, the keyboard driver can be completely malicious (i.e., a keylogger), and the screen output buffer can be intercepted by a malicious video card driver, etc. However, securing these inputs and outputs is paramount, especially when SGX is pushed to desktop applications, such as those used for DRM, online gaming, online shopping, online banking, etc.

Securing these forms of I/O requires that the endpoints be trusted or be able to attest that they can be trusted. From a client/server perspective, I/O is mainly viewed from the perspective of network communications, whose trust has already been addressed via encryption and remote attestation. Keyboards and screen buffers must also provide trust. One way this can be addressed is with special trusted hardware devices. There are a few commodity hardware devices already available on the market: Intel Protected Audio and Video (PAVP) [34] and Intel Identity Protection Technology (Intel IPT) [9]. PAVP [34] provides an encrypted data path and hardware accelerated decoding for high-definition video and audio playback. Intel IPT [9] provides a trustworthy display, called the Protected Transaction Display (PTD), so the display to users can be protected from malicious applications or compromised operating systems. We will investigate the use of these trusted hardware to build a secure channel between end-users and the enclave programs.

D.4.4 R2.4 Mitigating Vulnerabilities in Enclave Programs

Research Challenge 8. *SGX does not automatically secure vulnerable enclave programs; typical threats and attack vectors such as buffer overflows still remain. In fact, the compromise of an enclave program becomes more critical because SGX makes it impossible to analyze or even detect such compromises, and for end users, a compromised execution is completely indistinguishable from the correct execution (when an attacker also hides the system call footprints).*

Despite recent progress in model checking and formal methods [8, 16, 37, 40, 44], we cannot produce vulnerability-free software. There will be still an eternal war of memory errors between attackers and defenders [71] that applies to enclave programs as well. Even worse, previous OS-level defenses will become ineffective because fine-grained program behavior is invisible to the OS. While address space layout randomization (ASLR) may appear to mitigate this class of vulnerability, randomizing the layout of program code will lead to attestation failure for an enclave program. Therefore, we must search for new or revisit existing techniques to secure an enclave program from the enclave itself. Since we will not trust the underlying OS, we must use the self-defense mechanism in the fight against memory exploits. There are two practical strategies in this direction:

- **Control flow integrity (CFI)**. Since all control flow hijacks lead to a violation of control flow, CFI [1] or more generally various software guards (e.g., [2, 7, 10, 11, 19, 24, 64]) are proposed to defend against these attacks. While initial CFI approaches required access to the debugging symbols of a program, recent progress has made CFI largely practical. For instance, CCFIR [83] rewrites the binary with the relocation information from the COTS binary and enforces a practical CFI model. BinCFI [85] proposes a number of disassembling and rewriting strategies and can successfully build a CFI model for large COTS binaries. In this proposed research, we would like to revisit these practical CFI systems with the new constraints of handling cross enclave control flow transfers and also self-detecting violations by the enclave program itself.
- **Binary code randomization**. Most control flow hijacks need to know to where control flow transfers, and reuse this code to construct the exploit. As such, randomizing the layout of program code can significantly mitigate these attacks. ASLR has been shown to be effective in this direction [5, 68, 72]. However, remote attestation will fail if we directly attest the randomized code in the enclave. Instead, we need to derandomize the code and compute its attestation hash value. We plan to address this

challenge by developing a self-randomized ASLR scheme inside the enclave. Our prior effort has demonstrated that we can randomize instruction addresses every time we load a program, defeating code reuse attacks [78]. There are also several other efforts along this (re-)randomization direction (e.g., [6, 20, 21, 30, 31, 55]). We plan to integrate these re-randomization techniques while still addressing the attestation issue for enclave programs.

D.5 R3. Defending Against Malicious Use of SGX

Research Challenge 9. *We have discussed that an SGX program can be compromised because of exploitable vulnerabilities. There are also cases in which the SGX programs themselves can be entirely malicious. For instance, malware can now be executed inside an enclave that is undetectable and unanalyzable due to SGX blackboxing [62]. This will be especially problematic when SGX is widely available in desktop computers. On the other hand, while SGX is designed to defend attacks such as cloud providers in leaking or tampering with the sensitive code and data, cloud providers also have to make sure that there is no malicious SGX program that causes damages to their infrastructures. Therefore, it is also critical to study how to defeat malicious use of SGX.*

We believe there are two strategies to defeat the malicious use of SGX. One is to detect malicious software based on certain signatures (§D.5.1), and the other is to enforce the access control of enclave program execution based on explicit policies (§D.5.2).

D.5.1 R3.1 Detecting Malware Execution in the Enclave

Traditional intrusion detection systems utilize signature matching to detect the presence of malicious software, as well as anomaly-based detection using benign behavior profiles [12, 42, 53, 66]. We plan to design similar techniques to extract signatures of enclave program execution.

We cannot look at any control flow related information inside the enclave. Instead, we can only look at those operating system observable behaviors, such as system calls (including arguments and return values) [41] and their sequences [33, 76] or graphs [54]. We can also look at unique features in SGX program execution, such as the use of EPC pages, CPU time slice, the frequencies of page fault, etc. Therefore, we plan to investigate combining system call and SGX execution features to build a profile for malicious enclave programs (if we have a priori knowledge of their existence) as well as profiles for benign enclave programs.

D.5.2 R3.2 Preventing Malware Execution in the Enclave

The execution of an enclave program provides a perfect opportunity for the OS to enforce an execution policy (such as only allowing the execution of a system call that is registered). While it is challenging to acquire such a model through program analysis, we can ask programmers to explicitly register or develop tools (e.g., as in [13, 17, 45]) to help SGX programmers in expressing these policies. Considering there is no legacy SGX code at this moment, we believe this is a practical way of executing the SGX program. Each enclave program will have an associated system call execution policy file that specifies which system call (perhaps including its parameters) the enclave program will execute. The OS will enforce the execution of these system calls [27, 60]. In fact, the recent program execution model in mobile apps has already adopted a similar approach, in which an app has to explicitly express its resource use [25, 26, 38].

More broadly, we can also adopt the existing code producer and verifier execution model (such as the proof-carrying-code [51]) for SGX programs. For instance, we can restrict how an SGX program is produced (e.g., by a special compiler) and signed, and then verify certain properties of the code before executing it in the enclave. In fact, such an approach has been used in several existing systems. For example, Google's Native Client (NaCl) [81] requires a special compiler to modify the client programs at the source level. Microsoft's CFI [1] and XFI [24] requires code-producers to supply a *program database* (PDB) file with their released binaries. PittSFIeld [47] and SASI [23] require code-producers to provide `gcc`-produced assembly code.

Task	Success Metrics and Evaluation Plan	Time Line (RA-month)							
		Y1		Y2		Y3		Y4	
		G	U	G	U	G	U	G	U
R1.1	Linux kernel patches that support enclave program execution in the most recent kernel will be delivered and evaluated by the open source community.	12	12						
R1.2	A user level runtime library (with a lot of `glibc` code refactoring) and enclave system services (e.g., remote attestation) will be developed and also evaluated by the open source enclave program developers.	12	12						
R1.3	The SGX toolchain and ecosystem (e.g., packaging) support will be developed including compiler, loader, debugger, and binary packer. This will also be tested by open source enclave program developers. Meanwhile, we will also refactor the existing SPEC2006 benchmark and execute them in our platform to evaluate their performance overhead.		12	6	12				
R2.1	An `sgxlib` will be developed with special care of defending Iago attacks. This library will be tested with a variety of layer below attacks from OS kernels and hypervisors. We will use the benchmarks in [14, 28, 39] for the evaluation.			12	12				
R2.2	An enclave binary code rewriting, and a page fault noise injection scheme will be developed, and evaluated with benchmarks in [80] to defeat the controlled side channel attacks.			6	12	6			
R2.3	A scheme using trusted hardware to secure the keystrokes and screen I/O will be developed, and tested with keyloggers and screen buffer readers we will develop.					6	12		
R2.4	A binary level control flow integrity scheme, and a binary code randomization scheme will be developed and tested with the vulnerable enclave programs.					12	12		
R3.1	An enclave program execution profile will be developed, and then an intrusion detection model will be built based on the profile. Malicious or benign enclave program will be developed and tested with our detection model.							12	12
R3.2	A system call sandbox for enclave program execution will be developed, along with a system call policy derivation tool and enclave code verifier. They will also be tested with real enclave programs such as the SPEC2006 benchmark we refactored in R1.3.							12	12

Table 6: Tentative project time line and the detailed evaluation plan (Note that Yi denotes the i-th Year, G stands for Georgia Tech, and U stands for UT Dallas).

Therefore, when designing SGX malware execution prevention system, we would also like to investigate the model of restricting how the SGX code is produced, and then we can decline the execution of the code that does not pass the code verifiers. Such an approach will also bring many other advantages. For instance, we may be able to verify whether an enclave program contains incorrect logic that can expose secret key to memory outside the enclave during the verification process.

D.6 Broader Impacts: Education and Outreach

Curriculum Development. The systems security courses taught by both PIs will benefit greatly from the proposed research. In particular, PI Kim introduced the concept and recent research progress surrounding SGX into the course "Building Secure Systems (CS 8803)". In fall 2014, a group of graduate students initiated OpenSGX as part of their course project. The team leader, Soham Desai, joined the Intel Security Team after his graduation. PI Kim continues to improve current lab materials (limited to SGX itself) by introducing new security applications as part of this proposed work.

PI Lin also has been teaching a systems security class (CS 6332) since 2012. Given the significance of SGX, PI Lin has already developed course materials related to SGX. These include basic concepts to real implementation and hands-on projects, and were taught in fall 2015. The outcome of this research, especially the programming abstractions and toolchains, will serve as the enabling vehicle for student hands-on projects. PI Lin plans to have an SGX project using the OpenSGX platform developed by PI Kim in his fall 2015 class.

Outreach to undergraduates, women, and minorities. The University System of Georgia (USG) recently started an ambitious effort to increase the cyber security workforce across all units of the USG system (29 campuses). These include multiple HBCU colleges, including Spelman and Morehouse colleges, and research universities. Georgia Tech is expected to play a leading role in developing effective programs that deploy cyber security content to a diverse set of programs. Dr. Kim will lead the systems security content

development in this area, including curriculum development and an "educating the educators" program to maximize the number of impacted students. As part of this work, the course material and labs regarding SGX will be developed and introduced to the local security community.

PI Lin has a history of successfully mentoring undergraduate students in research. One undergraduate advisee (Garrett Greenwood) is a co-author of an NDSS 2014 paper, and another undergraduate student, Devin Willey, won the first place prize in the UT Dallas CS Department's First Annual Undergraduate Research Expo in 2015. Dr. Lin is also currently supervising one female Ph.D. student. Finally, Dr. Lin plans to organize cybersecurity workshops in K-12 summer camps, describing the recent advances in securing the cloud including SGX.

Outreach to the research and development community. The PIs plan to organize an SGX centralized workshop or tutorials at conferences such as ACM CCS. Given the importance of cloud computing today, there is a pressing need to adopt techniques such as SGX for secure cloud computing. Organizing an annual SGX workshop can bring together both academic researchers and industry practitioners to share their perspectives on SGX development, and update participants with the most recent progress in this field.

PI Kim is an active contributor to the open source community; not only did he initiate and lead the OpenSGX project, but he also works closely with practitioners. For example, the security group PI Kim leads was invited to the 2015 Linux Plumbers Conference to present another open source project that enables reboot-less kernel updates. To foster successful deployment and contribution to the community, the developed design, technical reports, and software artifacts developed through this proposed work will be publicly available.

D.7 Evaluation and Management Plan

We anticipate this project will require four years to complete. Table 6 outlines a tentative schedule of anticipated yearly foci for each specific research task, the corresponding evaluation plan, and the dedicated months in terms of the graduated research assistant (RA) from each institute. The details on how the two PIs collaborate and manage the project are presented in the attached **Collaboration Plan**.

D.8 Results From Prior NSF Support

Taesoo Kim: Dr. Kim is the PI of *"SaTC-EDU: EAGER: Big Data and Security: Educating The Next-Generation Security Analysts"* (NSF DGE 1500084, 06/2016-05/2018, $300,000). **Intellectual merit**: the identification of a set of basic principles and effective techniques of applying data analysis to security education, and the contribution to research and practice in data analysis and security. **Broader impacts**: newly designed, developed, and evaluated publicly available resources such as lab materials for teaching data-driven security analysts, as well as the training of better technical quality of security analysts.

Zhiqiang Lin: Dr. Lin is the PI on *"CAREER: A Dual-VM Binary Code Reuse Based Framework for Automated Virtual Machine Introspection"* (NSF CNS 1453011, 09/2015-08/2020, $535,054). **Intellectual merit**: the development of a novel dual-VM binary code reuse-based framework to automatically bridge the semantic gap for virtual machine introspection, as well as the enabling of a number of new applications such as automated out-of-VM guest management. **Broader impacts**: better and automated protection of computer systems against advanced malware attacks, as well as training of new security students and researchers. Lin is also co-PI on *"UTD SFS Renewal: Growing a Cybersecurity Community through SFS Scholarship Program at UTD"* (NSF DGE 1433753, 09/2014-08/2019, $3.9M). **Intellectual merit**: Recruitment and training of students (27 for 1433753) in our cyber security education programs and the development of a new concentration track in this direction. **Broader impacts**: successful placement of students in federal government, and various outreach activities to students in Texas.