Technical Report
1230

Leveraging Intel SGX Technology to Protect Security-Sensitive Applications

J.M. Sobchuk
S.R. O'Melia
D.M. Utin
R.I. Khazan

29 January 2018

Lincoln Laboratory
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LEXINGTON, MASSACHUSETTS

This material is based upon work supported by the Department of the Air Force under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001.

This report is the result of studies performed at Lincoln Laboratory, a federally funded research and development center operated by Massachusetts Institute of Technology. This material is based on work supported by the Department of the Air Force under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Department of the Air Force.

Massachusetts Institute of Technology
Lincoln Laboratory

Leveraging Intel SGX Technology to Protect Security-Sensitive Applications

J.M. Sobchuk
S.R. O'Melia
D.M. Utin
R.I. Khazan
Group 53

Technical Report 1230

29 January 2018

Lexington Massachusetts

This page intentionally left blank.

ABSTRACT

This report explains the basic process by which Intel Software Guard Extensions (SGX) can be leveraged into an existing codebase to protect a security-sensitive application. Intel SGX provides user-level applications with hardware-enforced confidentiality and integrity protections. These protections apply to all three phases of the operational data lifecycle: at rest, in use, and in transit. SGX shrinks the trusted computing base (and therefore the attack surface) of the application to only the hardware on the CPU chip and the portion of the application's software that is executed within the protected enclave. The SGX SDK enables relatively straightforward integration into existing C/C++ codebases while still ensuring program support for legacy and non-Intel platforms.

This page intentionally left blank.

ACKNOWLEDGMENTS

The authors would like to thank Walt Bastow, Kosta Feldman, Seth Toplosky, and Mark Yeager of MIT Lincoln Laboratory for their contributions to the project on which the SGX implementation methods described in this paper were put into practice.

This page intentionally left blank.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

LIST OF ILLUSTRATIONS

This page intentionally left blank.

LIST OF TABLES

This page intentionally left blank.

1. INTRODUCTION

1.1 THE NEED FOR HARDWARE PROTECTIONS

As cloud computing and other remote execution technologies become more widespread, service providers are increasingly running their programs on machines over which they have no physical control. If these programs involve the storage, processing, or transportation of sensitive data, the need arises to protect that data from being read or altered on untrusted machines. For example, stored user authentication information needs to be kept secure from a curious cloud service provider, digital rights management keys must not be leaked to an EULA-bending end user, and employee records must be resistant to a vulnerable node on a distributed platform. Intel SGX fulfills this necessity by providing hardware-backed protections to security-sensitive applications vulnerable to disclosure or modification by privileged software and hardware.

1.2 WHAT IS SGX?

1.2.1 Threat model

Intel SGX's threat model encompasses both software- and hardware-capable adversaries. It protects against unprivileged software, which has the ability to execute Ring 3 instructions and access memory that is mapped to it by the OS. It also protects against system-level and startup software, which can manage task scheduling and processor execution mode, access any visible memory on the platform, interface with hardware devices directly, and even control the initial system state. To address adversaries with direct access to internal hardware, SGX's hardware checks and memory encryption prevent memory snooping and bus tapping attacks.

However, side channel attacks against SGX-enabled programs are explicitly out-of-scope; it is the responsibility of the programmer to write safe code. For example, if a program executing with SGX protections includes sensitive data-dependent memory accesses such as in [1], an attacker can force cache misses in order to measure memory access time by the program and derive the sensitive data.

1.2.2 Enclave

Intel SGX is a set of hardware and CPU instructions that provides user-level (Ring 3) applications with hardware-enforced confidentiality and integrity protections. SGX allows developers to partition their applications into hardware-protected secure containers referred to as enclaves. An SGX enclave is an isolated container within the running application's address space. Code that executes within an enclave is tamper-resistant, and any sensitive data that is generated within or provisioned to an enclave is protected from snooping or disclosure outside the enclave. SGX is available starting in Intel's 6th generation of CPUs, codenamed Skylake.

Enclave code is compiled as a shared-object library appended with an RSA-signed certificate. The certificate contains data such as the enclave measurement and application author ID for integrity and authenticity purposes, ensuring that any enclave loaded into memory and executed is genuine and has not been modified by anyone other than its author.

1.2.3 Hardware keys

The SGX hardware requires persistent keys in order to deliver its promised functionality. Two CPU-unique symmetric keys are burned into the processor hardware at manufacture time: the root provisioning key, which is retained by Intel, and the root seal key, which is only known by the CPU onto which it is burned. [2] The root provisioning key is used to derive additional keys during enclave provisioning and attestation in order to demonstrate that the enclave is running on genuine SGX hardware. The root seal key derives keys to cryptographically seal data to the hardware and enclave instance. All other persistent keys in the system are derived from one or both of these keys when necessary.

1.2.4 Memory protection

Enclave memory is protected from the rest of the system using two mechanisms: hardware-enforced checks prevent enclave memory from being read or modified by non-enclave code within the CPU (in SRAM), while the hardware-based SGX Memory Encryption Engine (MEE) encrypts and authenticates all enclave memory before writing it to the (untrusted) system's main memory (DRAM). The MEE performs this encryption/decryption and authentication/verification of DRAM using hardware-based cryptographic keys that are randomly generated at power-on time, separate from the system's persistent hardware keys. [3]

1.3 BENEFITS

1.3.1 Protection from higher privilege levels

The primary security benefit that Intel SGX provides is the protection of enclave code and data from higher privilege levels, including the OS, virtual machine monitor (VMM), and even the hardware. In a traditional computing environment, a security-sensitive application that must maintain the confidentiality and integrity of its execution and data would need to trust that the OS, VMM, and hardware also maintain those security properties. If any of those entities snoop on the application and modify its code or data, the confidentiality and integrity of the application are broken. Likewise, if a user-level malicious application running on the same machine is able to exploit a vulnerability in the OS, escalate its privilege, or otherwise break isolation between itself and the security-sensitive application, the confidentiality and integrity are compromised once again. With SGX, the only entities that the application needs to trust are the CPU hardware and the code that is executing within the secure enclave. Therefore, as long as the application is running an enclave and handles its sensitive data securely, its sensitive data storage and processing remains confidentiality and integrity protected, even if it is running on a machine that has been compromised with privileged malware.

1.3.2 Shrinking of TCB and attack surface

Secure execution within an SGX enclave entails a massive reduction of the application's trusted computing base (TCB) and therefore its attack surface. Instead of taking steps to harden the hardware, VMM, OS, and entire application, with SGX, only the CPU hardware and the application code running within an enclave needs to be trusted and can be considered as part of the application's attack surface.

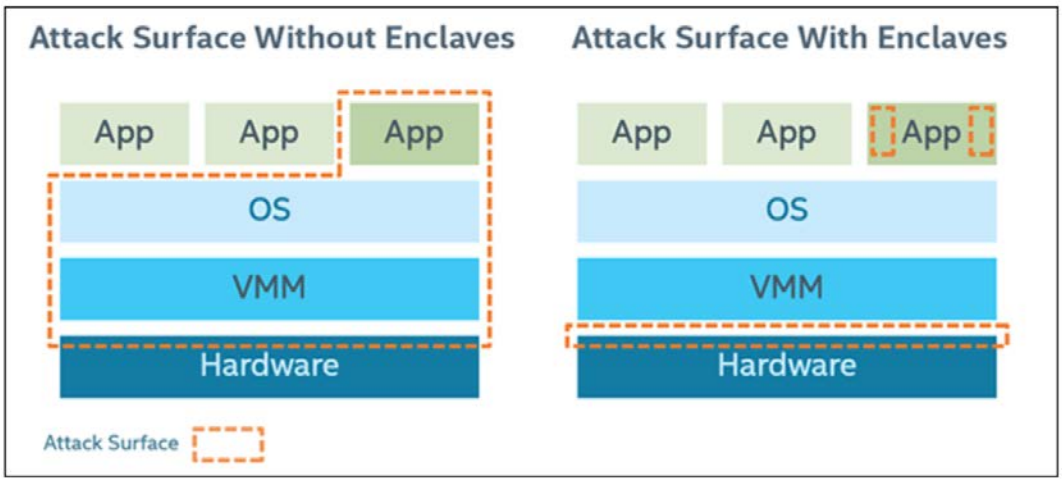

Figure 1. Attack surface of a security-sensitive application without SGX enclaves (left) and with SGX enclaves (right). [4]

1.3.3 Debug and test with simulation mode

The official SGX Software Development Kit (SDK) from Intel enables developers to build their applications in three different profiles (debug, pre-release, or release) under two different modes (software simulation or hardware). [5] Hardware debug enclaves can be inspected using the SGX debug instructions by an enclave-aware debugger. Hardware pre-release enclaves are also launched in enclave-debug mode, but with the same compiler optimization and debug symbol support as the release profile. This enables the enclave to be performance tested under similar conditions as the release build while still being debug-able. The simulation mode for all three profiles builds the application linked with the SGX simulation libraries, allowing the enclave to be built, tested, debugged, and executed on any non-SGX enabled platform.

1.3.4 Attestation and sealing

SGX provides mechanisms for entities to attest to the legitimacy of a running enclave, both locally (two enclaves running on the same machine verifying each other) and remotely (an entity on one machine verifying an enclave on a separate machine). Attestation guarantees the trustworthiness of the enclave to the challenging entity, which the entity may want to ensure before provisioning any sensitive data to the enclave.

SGX also provides the ability to seal data to a specific enclave for long-term storage in non-volatile memory. The data is cryptographically sealed using a key derived via SGX hardware before being written to disk and can only be unsealed by a running instance of the same enclave on the same machine (or optionally by an instance of any enclave signed by the same author running on the same machine). This protects the confidentiality and integrity of enclave data while making the data available across separate executions of the application.

1.4 CAVEATS

1.4.1 Ring 3 (no syscalls)

As a consequence of providing enclave protections only to user-level (Ring 3) applications, certain instructions are illegal within an enclave. [6] This includes all privileged instructions such as system calls, input/output instructions, as well as any instructions that may lead to a VMEXIT. Developers must ensure that their enclave code does not contain any illegal instructions. Some methods to accomplish this include designing the application to not include any illegal instructions within the enclave boundary or replacing illegal enclave code with safe alternatives.

1.4.2 Limited memory

The size of the physical protected enclave memory is currently limited to 128 MB. After subtracting the overhead required for the MEE, only about 96 MB is left for the application developer's enclave. [3] However, this can be seen as an advantage, since it forces the developer to keep the program's attack surface small. The newer SGX2 instruction set extension enables dynamic enclave memory management for programs that require a larger amount of enclave memory. [7]

1.4.3 Overheads

The protections that SGX provides incur some timing and storage overheads. Reading and writing to main memory experience an additional performance penalty compared to usual cache misses due to the encryption/decryption of data by the MEE. In experiments conducted by Intel [3], the average performance degradation due to the MEE is about 5.5 percent. Sealing and unsealing data for long term storage suffers a performance penalty in a similar fashion since the data must be encrypted or decrypted. Sealing also requires roughly 500 bytes of storage overhead per piece of sealed data to store the metadata that the SGX hardware needs in order to unseal it later. [8]

1.4.4 C/C++, Windows and Linux only

The official SGX SDK currently supports only C and C++ for enclave development. An unofficial SDK provided by China-based Baidu X-Lab supports the development of SGX enclaves in the Rust programming language, but this SDK has not been vetted by Intel. [9] Also, the official SDK is currently only available for Windows and Linux.

1.4.5 Licensing with Intel

The ability to run a production-ready (hardware release) enclave on actual SGX hardware requires the developer to obtain a commercial license from Intel. [10] The acquisition process involves providing a CA-signed certificate to Intel and demonstrating that the developer is employing secure software development practices, storing their signing key safely, and is not creating malware to be executed within enclaves. In addition, the license provides the developer with access to Intel's Production Attestation Service, which is required for remote attestation of production-level software. Without a commercial license from Intel, the developer can only run enclaves in software simulation or on hardware with support for inspection by a debugger, which do not deliver the full confidentiality and integrity protections of the hardware.

2. ARCHITECTING PROGRAM INTO SENSITIVE AND NON-SENSITIVE PARTS

The first and most important step in integrating SGX protections into an existing application is partitioning the codebase into security-sensitive and non-security-sensitive parts. The component(s) that are security sensitive must be placed within the trusted enclave while the rest of the code can reside in the untrusted application space. The developer should aim to shrink the size of the enclave code as much as possible – this reduces the size of code that may need to be formally analyzed and verified, which also decreases the size of the program's attack surface. Basically, the code that must be placed in an enclave is the answer to the following question: What is the bare minimum amount of code that requires handling sensitive data in the clear?

As an example, consider a feature-rich application that sometimes needs to perform some type of cryptographic processing using sensitive keys, and the confidentiality and integrity of those keys must be protected at all times. The cryptographic component of the application would reside within the trusted enclave, while the remainder of the code would continue to execute in the untrusted application space. Since all code that requires the use of key material resides within an enclave, the key material's confidentiality and integrity is protected by the SGX hardware.

This page intentionally left blank.

3. PROJECT SETUP AND ASSUMPTIONS

3.1 DEVELOPMENT ENVIRONMENT

This paper assumes that the software developer is working on a C++ project on a 64-bit Linux-based operating system (specifically Ubuntu 16.04). The developer does not use an integrated development environment (IDE), but instead writes and modifies code using a text editor and compiles binaries and libraries using the latest versions of g++ and GNU Make.

3.2 INSTALLING THE SDK

The SGX software development kit (SDK) can be downloaded from Intel's website [11] and is available for 64-bit versions of both Windows and Linux-based OSes. The SGX installation guide [12] goes into greater detail about the SDK installation process that is summarized here. In order to run the application using actual SGX hardware, SGX must be supported by the processor and enabled in the system's BIOS, then the SGX driver, platform software (PSW), and SDK must be installed in that order. To develop SGX applications on non-SGX-supported systems or to use only the software simulation mode, only the SDK needs to be installed.

The SDK includes source code for sample SGX programs in the SampleCode/ directory. Most of these projects organize their code into separate directories for the untrusted application and trusted enclave spaces and include all of the necessary files to build an SGX-enabled program. Therefore, it is recommended to emulate the directory structure and Makefiles from these sample projects, and copy the signing key files, if necessary, for software and debug enclaves.

3.3 TERMINOLOGY

The remainder of this document refers to the codebase in the following fashion. The entirety of the SGX-enabled application is referred to as the "program". The portion of the program that resides in the untrusted application space is referred to as the "application" or "application code," and the portion of the program that resides within the trusted enclaves is referred to as the "enclave" or "enclave code".

This page intentionally left blank.

4. SGX IMPLEMENTATION

This section details the methods by which SGX support can be added to an existing codebase. It covers how to build the enclave shared-object library, create edge routines to interface between the enclave and untrusted application, and modify enclave code to remove illegal instructions.

4.1 BUILDING WITH SGX

It is recommended for an enclave developer to model the project's Makefile after one that is bundled with one of the sample projects that are included with the SDK. The basic additions that need to be made to a non-SGX Makefile are covered below, and the reader can refer to a sample project's Makefile for a real-world implementation.

4.1.1 SGX SDK settings

The first section of the Makefile should have variables describing the SGX architecture (x86 or x64), mode (hardware or simulation) and profile (debug/pre-release/release). Several flag and path variables are set based on these options.

```
SGX_SDK ?= /opt/intel/sgxsdk
SGX_MODE ?= HW
SGX_ARCH ?= x64
SGX_DEBUG ?= 1

ifeq ($(shell getconf LONG_BIT), 32)
	SGX_ARCH := x86
else ifeq ($(findstring -m32, $(CXXFLAGS)), -m32)
	SGX_ARCH := x86
endif

ifeq ($(SGX_ARCH), x86)
	SGX_COMMON_CFLAGS := -m32
	SGX_LIBRARY_PATH := $(SGX_SDK)/lib
	SGX_ENCLAVE_SIGNER := $(SGX_SDK)/bin/x86/sgx_sign
	SGX_EDGER8R := $(SGX_SDK)/bin/x86/sgx_edger8r
else
	SGX_COMMON_CFLAGS := -m64
	SGX_LIBRARY_PATH := $(SGX_SDK)/lib64
	SGX_ENCLAVE_SIGNER := $(SGX_SDK)/bin/x64/sgx_sign
	SGX_EDGER8R := $(SGX_SDK)/bin/x64/sgx_edger8r
endif
```

Figure 2. Snippet of basic SGX settings defined at the beginning of the Makefile provided in the SDK's SampleEnclave project.

4.1.2 Using SGX libraries

Additional SGX libraries need to be linked with both the untrusted application and trusted enclave code in order to use the SGX API. For example, the Untrusted Run-time System (uRTS) library needs to be linked with the application and the Trusted Run-time System must be linked to the enclave. These libraries are responsible for tasks such as making calls into an enclave and managing the enclave while it is running. Either the actual versions of these libraries or the "simulation" versions must be linked depending on the SGX mode (hardware or software simulation).

```
ifneq ($(SGX_MODE), HW)
      Urts_Library_Name := sgx_urts_sim
      Uae_Library_Name := sgx_uae_service_sim
else
      Urts_Library_Name := sgx_urts
      Uae_Library_Name := sgx_uae_service
endif

App_Link_Flags := $(SGX_COMMON_CFLAGS) -L$(SGX_LIBRARY_PATH) \
-l$(Urts_Library_Name) -l$(Uae_Library_Name)
```

Figure 3. Setting linker flags for the application.

4.1.3 Enclave gcc flags

The enclave .so library requires several compiler and linker flags in order to be built correctly. Some examples include the –nostdinc and –fpie compiler flags (to not include the standard system header files and to compile position-independent code) as well as the –nostdlib and –Bstatic linker flags (to not include standard libraries and to link shared objects statically).

```
Enclave_C_Flags := $(SGX_COMMON_CFLAGS) -nostdinc -fvisibility=hidden -fpie \
-fstack-protector $(Enclave_Include_Paths)
Enclave_Cpp_Flags := $(Enclave_C_Flags) -std=c++03 -nostdinc++

Enclave_Link_Flags := $(SGX_COMMON_CFLAGS) -Wl,--no-undefined -nostdlib \
-nodefaultlibs -nostartfiles -L$(SGX_LIBRARY_PATH) \
      -Wl,--whole-archive -l$(Trts_Library_Name) -Wl,--no-whole-archive \
      -Wl,--start-group -lsgx_tstdc -lsgx_tstdcxx -l$(Crypto_Library_Name) \
-l$(Service_Library_Name) -Wl,--end-group \
      -Wl,-Bstatic -Wl,-Bsymbolic -Wl,--no-undefined \
      -Wl,-pie,-eenclave_entry -Wl,--export-dynamic  \
      -Wl,--defsym,__ImageBase=0 \
      -Wl,--version-script=Enclave/Enclave.lds
```

Figure 4. Example flags used for enclave compilation.

4.1.4 Create .edl file (edge routines)

Once the enclave developer has defined the boundary between the application and enclave, the edge routines that interface between the two must be defined. Any function that passes this boundary is considered an edge routine – functions that are called from the application into enclave are referred to as enclave calls (ecalls) while functions that are called from the enclave into the application are named outside calls (ocalls).

The function prototypes for these edge routines must be placed in an Enclave Definition Language (.edl) file, which will be used by the SGX Edger8r tool during compilation to generate proxy functions for the ecalls and ocalls. When a program calls an edge routine, it actually invokes the generated proxy function, which then handles the data marshalling and enclave border crossing. Edger8r generates four files to hold these proxy functions – two each for the application and enclave. As an example, if the .edl filename is Enclave.edl, Edger8r generates Enclave_u.c and Enclave_u.h for the application code and Enclave_t.c and Enclave_t.h for the enclave code, where the _u and _t represent the untrusted and trusted sides of these proxies.

Only objects of known scope can be passed as arguments to an ecall or ocall; therefore, C buffers and structures are acceptable but C++ objects are not. [13] [14] Any C++ object arguments should be converted to an equivalent C structure before being passed to an edge routine. For example, a C++ vector<uint8_t> argument in an ecall could be passed as a uint8_t* (along with an additional variable to represent its size) instead. Once inside the enclave, the buffer could be converted back to a vector if necessary.

All buffer arguments in edge routines must include a direction specifier and data count. This specifier enables the Edger8r tool to automatically generate data marshalling code for the buffer across the enclave boundary. The specifier can have values of [in], which copies the buffer into the edge routine; [out], which initializes a buffer of zeros in the edge routine and copies it to the calling code upon function completion; or [in/out], in which the data is copied back and forth. Additionally, the specifier can designate a buffer as [user_check], in which no data marshalling is performed automatically and no data size is required. In this case, the raw pointer is passed across the enclave boundary, and it is the developer's responsibility to marshal and access the data correctly.

Further information about the EDL syntax can be found at [15].

```
void KmCrypto::encryptAES256_GCM( const vector<uint8_t> &plainText, const
vector<uint8_t> &aad, const vector<uint8_t> &key, vector<uint8_t>
&rCipherText, vector<uint8_t> &rIv, vector<uint8_t> &rAuthTag )

void ecall_encryptAES256_GCM( [user_check] const uint8_t *plainText_buffer,
uint32_t plainText_size,
[user_check] const uint8_t *aad_buffer, uint32_t aad_size,
[user_check] const uint8_t *key_buffer, uint32_t key_size,
[user_check] uint8_t *rCipherText_buffer, uint32_t rCipherText_size,
[user_check] uint8_t *rIv_buffer, uint32_t rIv_size,
[user_check] uint8_t *rAuthTag_buffer, uint32_t rAuthTag_size )
```

Figure 5. Example of an encryption function prototype and its ecall counterpart defined in the EDL file.

4.1.5 Enclave signing

Once the enclave's shared object (.so) library is compiled, it must be signed with the developer's private key using the enclave signing tool, sgx_sign, included with the SDK. The sample projects' Makefiles handle this automatically using an included private key file for all build types except hardware release, where it is expected that the developer uses their own key and perform the signing manually in a secure environment.

Optionally, an enclave configuration file can be passed to the signing tool. [16] This XML file contains user-defined parameters for the enclave such as its security version number or product ID and is included as part of the signature calculation.

4.2 MODIFY APPLICATION AND ENCLAVE CODE

4.2.1 Include header files for libraries

Certain libraries from the SGX SDK need to be included in both the application and enclave in order to support SGX functionality. On the application side, sgx_urts.h (untrusted runtime system) and Enclave_u.h (untrusted proxy generated by Edger8r) are needed for SGX variable types, enclave creation and destruction, and edge routine calls. Additionally, other SGX library header files could be included, such as sgx_tseal.h for calculating the amount of memory to allocate for an output buffer that will contain sealed data.

On the enclave side, sgx_trts.h (trusted runtime system) and Enclave_t.h (trusted proxy generated by Edger8r) are needed for SGX variable types, trusted memory checking functions, and edge routine calls. Additionally, other SGX library header files could be included, such as sgx_tseal.h for sealing data to the enclave.

4.2.2 Initialize and destroy enclave

The untrusted application handles the initialization and destruction of enclaves. Launching an enclave is accomplished by calling the sgx_create_enclave() function provided by the SDK. This function takes in the enclave filename, debug flag, and optionally the enclave initialization token as input, and outputs the enclave ID (an unsigned 64-bit integer), updated token data if it has been altered, and optionally additional enclave attributes. The enclave ID is used as the first argument to all ecalls to specify the enclave in which the function is being executed. The token data stores attributes and measurements of the enclave that are checked and updated upon enclave creation to increase the enclave initialization performance.

An enclave can be terminated simply by passing its ID to the sgx_destroy_enclave() function via the application.

```
sgx_status_t sgx_status = SGX_ERROR_UNEXPECTED;
sgx_launch_token_t token = {0};
int updated = 0;

...
// Read token data from file and store in "token"
...

sgx_status = sgx_create_enclave(ENCLAVE_FILENAME, SGX_DEBUG_FLAG, &token,
&updated, &global_enclave_id, NULL);
```

Figure 6. Partial code to demonstrate call to sgx_create_enclave().

4.2.3 Call into/out of enclave

The developer can make an ecall into a particular enclave at any point between its creation and destruction. This is accomplished simply by calling the ecall as it is defined in the .edl file, except that the enclave ID is added as the first argument of the function.

The enclave code can make an ocall if it requires a function that can only be executed in the untrusted application. The ocall can be made by calling the function defined in the .edl file.

Since the application is untrusted, the developer cannot make any assumptions within the enclave regarding the I/O parameters of an ecall or ocall or the state of execution when an ecall is made or an ocall returns. Therefore, the developer must perform proper sanity checking within the enclave to ensure that no sensitive data leaves its borders.

4.2.4 Marshall I/O buffers in enclave

For variables that are passed by reference, if the direction specifiers [in], [out], or [in/out] are used in the ecall or ocall function prototype, data marshalling across the enclave boundary happens automatically and no action is required from the enclave developer. However, the developer must handle data marshalling of any [user_check] buffers manually. As an example use case, a developer may want to pass a buffer of encrypted data into an enclave function. If the developer designates that buffer with [user_check], the buffer should be copied into enclave space by the developer before any processing is performed on it.

```
void ecall_encryptAES256_GCM( uint8_t *key_buffer, uint32_t key_size, ... )
{
  // Copy arguments from untrusted space to trusted space
  // Also convert from C buffer to C++ vector:
  vector<uint8_t> key_vec(&key_buffer[0], &key_buffer[key_size]);

  // Continue processing
  ...
```

Figure 7. The key material for this encryption function is stored in a C++ vector. To pass it through the enclave boundary, it is converted to a buffer within the application and then manually copied back into a vector defined within protected enclave memory.

4.2.5 Handle sensitive data in enclave

The developer must write the enclave in a secure manner such that no sensitive data can ever leave the enclave boundary. Before unsealing sensitive data, the developer can call the sgx_is_within_enclave() function provided by the SDK to guarantee that the output buffer (which will contain raw sensitive data) is strictly within the enclave's protected memory. Once the processing of sensitive material is complete, the developer should call memset_s() to clear every variable that contains sensitive data. The use of memset_s() cannot be optimized away by the compiler, guaranteeing that the data is overwritten.

4.2.6 Remove illegal instructions

There are several ways to exclude illegal instructions from the enclave code. The most straightforward method is to partition the codebase such that the enclave portion does not include any illegal instructions in the first place. For example, if the cryptographic processing portion of a codebase does not contain any illegal instructions, it would be simple to encapsulate that portion within an enclave without making any major changes.

However, there are cases where this task is either impossible or impractical. For these cases, the most effective option may be to define ocalls in the .edl file. However, the developer must keep in mind that since the processing of an ocall function happens in the untrusted domain, the function may not perform the operations nor return the parameters that the enclave expects.

Yet another option involves replacing the illegal instructions with SGX-provided counterparts. For example, instead of making a system call to the OS's random number generator (RNG), which has the potential of issuing an illegal VMEXIT instruction, the developer could use the RNG provided by the SGX SDK (sgx_read_rand()).

The final option would be to simply remove the offending code from the program entirely. This option could be useful when the illegal instructions do not affect the functionality of the program. For example, print statements used for debugging purposes could be commented out of the code when building the SGX-enabled program.

5. PROTECTING SENSITIVE DATA

5.1 SEALING

As explained in Section 1.3.4, SGX provides the ability to cryptographically seal sensitive data before placing it in untrusted non-volatile storage. The sealing and unsealing processes are fairly straightforward and must occur within the enclave. To seal data, the developer needs to make two SGX library calls: sgx_calc_sealed_data_size() to determine how much memory to allocate for the output buffer and sgx_seal_data() to seal the data using the seal key derived in the SGX hardware. As expected, unsealing data works in a similar fashion: the developer calls sgx_get_encrypt_txt_len() to get the size of the output (unsealed) data and sgx_unseal_data() to unseal the data and write it to the buffer.

```
// Initialize output vector
uint32_t unsealedDataSize = sgx_get_encrypt_txt_len(
(sgx_sealed_data_t*) &sealedData.front());
vector<uint8_t> unsealedData(unsealedDataSize);

// Ensure that the output vector is completely within the enclave
if(sgx_is_within_enclave(&unsealedData.front(), unsealedDataSize) != 1)
{
  // Buffer is outside enclave!
  return ERROR_UNSEAL_DATA;
}

// Unseal the data and store it in "plaintext"
sgx_status_t result = sgx_unseal_data((sgx_sealed_data_t*)
&sealedData.front(), NULL, 0, &unsealedData.front(),
&unsealedDataSize);
```

Figure 8. Example of the data unsealing process. The enclave should use sgx_is_within_enclave() to ensure that unsealed sensitive data does not leave the enclave boundary.

5.2 REMOTE ATTESTATION

Remote attestation is the process by which a challenger verifies the legitimacy of an enclave that is running on a separate physical machine before provisioning sensitive data to it. [17] Remote attestation involves a three-party communication between the SGX-enabled program, remote challenger (referred to as a "service provider" by Intel), and Intel's Attestation Service (IAS). First, the program's enclave and remote challenger perform an Elliptic-Curve Diffie-Hellman (ECDH) key exchange to establish a secure channel. Next, the enclave sends a report of its authenticity to the challenger, who verifies it with the IAS. If the verification is successful, the challenger can then provision sensitive data to the enclave over the secure channel with confidence that the data will not be leaked to the surrounding untrusted system.

This page intentionally left blank.

6. OPTIMIZATIONS

This section describes several optimizations that developers can make in order to maximize the performance of their SGX-enabled program, make minimal changes to their program-specific code, and ensure that their programs are still compatible with legacy and non-Intel hardware.

6.1 SHIM LAYER

The most important optimization for a developer to leverage is to contain all of the program's SGX-specific code in a "shim layer". This shim layer is responsible for all SGX-related processing, including instantiating and destroying enclaves, making ecalls and ocalls, and sealing and unsealing data. The shim layer straddles the enclave boundary, with one half of it in the application space and the other half contained within the enclave. This makes the enclave boundary opaque to the surrounding program-specific code and allows the API between the application and enclave program code to remain the same, since all data marshalling happens within the shim layer. Logically, the program code on both sides of the enclave boundary interacts as if the shim is not even there.

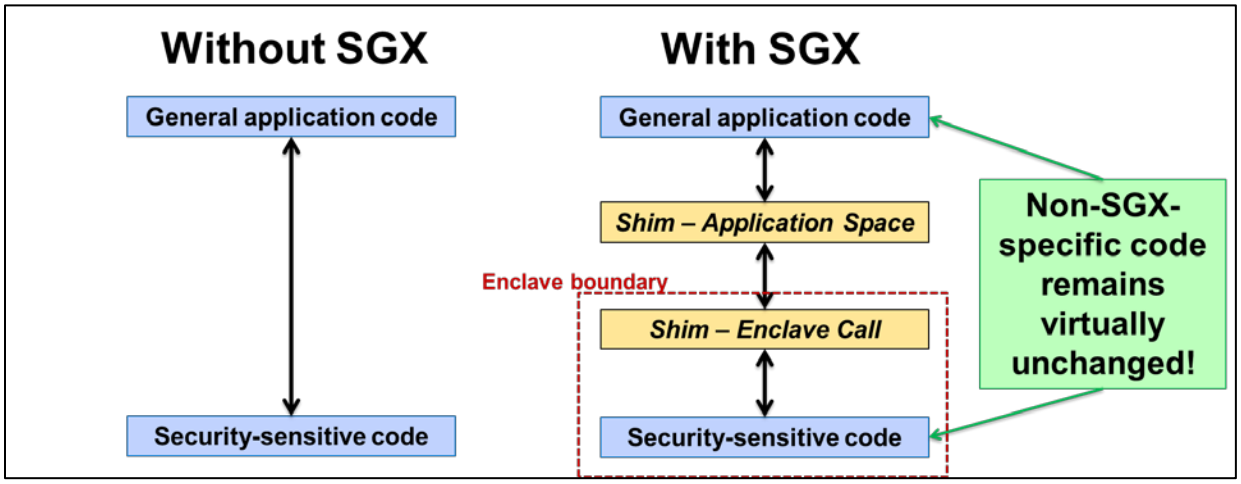

Figure 9. The API for the security-sensitive code is the same whether the "legacy" or "SGX-enabled" version of the program is built. The only aspect of the general application code and security-sensitive code that changes is the inclusion of SGX-specific header files.

6.2 DUAL CODE PATHS

An application should never crash or otherwise fail ungracefully solely because the platform does not support Intel SGX; therefore, the developer should include dual code paths based on whether or not SGX is supported. [18] The implementation of the dual code paths is left to the developer; based on the program and the sensitivity of the material the non-SGX path could do anything from implementing the full suite of security-sensitive code to displaying an error message and failing gracefully. The SDK includes functions to detect SGX support on the platform.

6.3 CODE REFACTORING

If the program code requires several jumps across the enclave boundary in a single function, the developer may choose to refactor that code to decrease the number of enclave border crossings. For example, the program may be structured to encrypt several pieces of data by calling the encryption function in a loop for each piece. If the loop is in the application space but the encryption function is within an enclave, each iteration of the loop involves all of the data marshalling, encryption key unsealing, and memory checking innate to an enclave call, adding overhead to the program. This code could be refactored to prepare all pieces of data ahead of time, pass them into the enclave all at once, perform the encryption, and return the result in bulk. This would involve only one enclave border crossing, decreasing the amount of overhead due to leveraging SGX protections in this program.

6.4 MACRO DEFINITIONS

If all SGX-specific code is implemented in the shim layer, the developer can use macro definitions at the start of source code files to decide whether to include it or not. A simple flag can be passed to the compiler to determine whether or not the shim layer is included, as shown in Figure 10. Since the API between the program-specific code is unchanged when implementing a shim layer, the function calls within the general application layer are identical whether the SGX or non-SGX programs are built.

```
// In general_app_code.h
#ifdef USING_SGX
  #include "secure_code_shim.h"
#else
  #include "secure_code_real.h"
#endif
```

Figure 10. If the USING_SGX macro is defined, the application code includes the shim layer's header file; otherwise it includes the header of the actual security-sensitive code.

6.5 ERROR HANDLING

If the program's enclave code includes explicit error checking and exception throwing, the developer must catch the exceptions within the enclave and pass a status flag back to the application indicating that an error occurred. Exceptions are not allowed to be thrown across the enclave boundary, but they can be thrown and caught within the enclave. Each ecall function body should be wrapped in a try/catch block so that any exceptions that are thrown within them are caught within the enclave. A status variable should be passed to each ecall from the application and set to the caught error code within the catch block. This status variable should be checked upon return from the ecall in the application's shim layer, and an exception should be thrown if the status indicates that an error occurred in the enclave. This emulates the exception throwing/catching functionality of the program-specific code in a non-SGX-enabled build of the program.

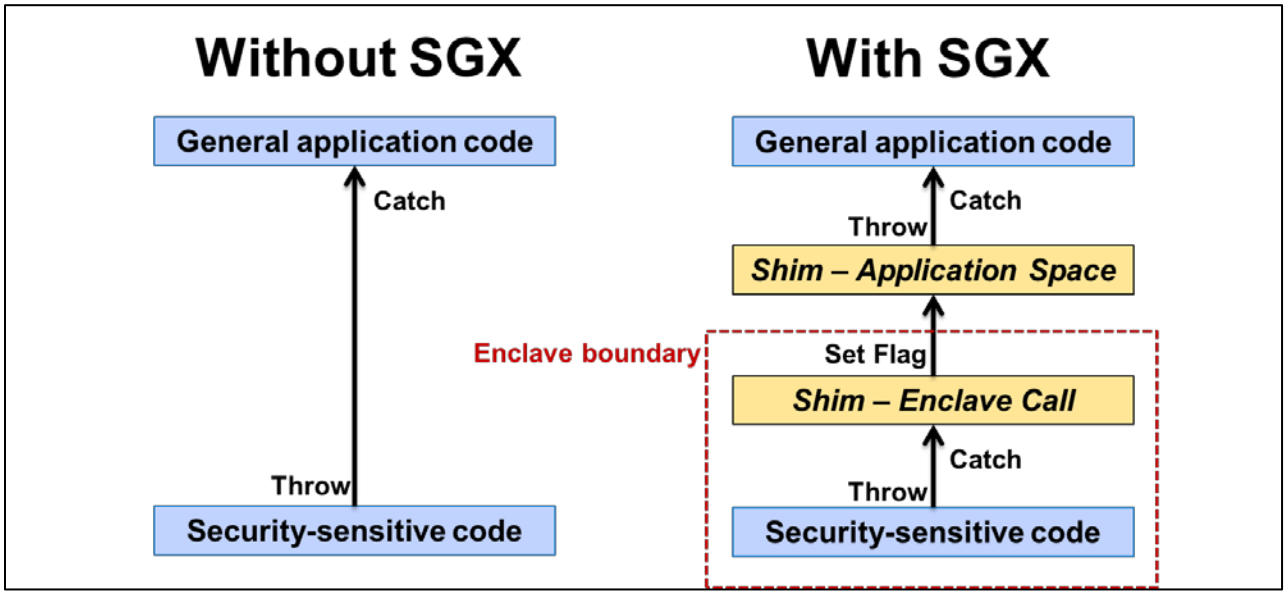

Figure 11. Exception throwing and catching can be emulated across the enclave boundary by setting a flag variable within the enclave when an error is caught and checking that flag once the ecall returns to the application.

This page intentionally left blank.

7. RELATED WORK

7.1 ARM TRUSTZONE AND AMD SECURE PROCESSOR

Arm TrustZone technology provides system-wide hardware isolation for trusted software. [19] It works by partitioning the computing space into hardware-separated secure and non-secure worlds, disallowing software in the non-secure world from accessing secure resources directly. Arm provides a reference implementation of low-level secure world software known as Arm Trusted Firmware [20] which can be used to launch a trusted OS and trusted applications through secure boot.

AMD's Secure Processor [21] is simply an on-chip Arm processor with TrustZone technology, and Apple's Secure Enclave Processor on iOS devices is essentially a highly customized version of TrustZone technology. [22]

7.2 INTEL TXT

Intel's Trusted Execution Technology (TXT) [23] provides hardware-based security by measuring each component of the boot process to ensure that the system was initialized securely and has not been altered from a known trusted configuration. Intel TXT establishes a "root of trust" in hardware that provides launch control policy and verified launch protections. It also enables the sealing of sensitive data and attestation of platform legitimacy. Intel TXT was launched in 2010 and is currently available in most Intel Xeon processors.

7.3 CRYPTO LIBRARIES IN SGX

Intel has implemented a cryptographic library that leverages SGX enclaves. It is based on the OpenSSL library and available on GitHub under an open-source license. [24] This library incorporates a shim layer as described in this whitepaper, enabling the API exposed by the library to be fully compliant with a subset of the unmodified OpenSSL API. This allows for straightforward compatibility with projects that already interface with the OpenSSL library.

The mbedtls library (previously known as PolarSSL) has been ported to use SGX enclaves. It provides an implementation of the TLS protocol suite and a variety of cryptographic primitives. It is also available for download on GitHub under the Apache License version 2.0. [25]

7.4 INTEL KPT

Intel Key Protection Technology (KPT) [26] augments Intel's QuickAssist Technology (QAT) hardware crypto accelerator with the run-time protection of Intel Platform Trust Technology (PTT) for persistent key storage at rest. It supports high-performance, scalable, security-enabled cryptography and key management and is intended to replace traditional hardware security modules in server environments. Intel introduced KPT with their latest Intel Xeon Scalable processors.

TABLE 1
Comparison of Hardware-Based Security Technologies

	Intel SGX	Arm TrustZone	Intel TXT	Crypto Libraries in SGX	Intel KPT
Hardware trusted?	CPU chip only	Yes	Yes	CPU chip only	CPU chip only
OS/VMM trusted?	No	Yes	Yes	No	No
Confidentiality and Integrity Protections	Yes	Yes	Yes	Yes	Yes
Trusted boot	No	Yes	Yes	No	No
Attestation	Yes	Yes	Yes	Yes	Yes
Built-in data sealing	Yes	No	Yes	Yes	No

8. CONCLUSION

Intel SGX provides secure applications with hardware-backed confidentiality and integrity protections. As long as the enclave is bug-free and is not susceptible to side-channel attacks, these protections apply to all three phases of the operational data lifecycle: at rest through sealing, in use through processing within an enclave, and in transit through ECDH key exchange during remote attestation. SGX shrinks the trusted computing base (and therefore the attack surface) of the application to only the hardware on the CPU chip and the portion of the application's software that is executed within the protected enclave, versus the entire hardware/VMM/OS/application stack in traditional computing environments. Its SDK enables relatively straightforward integration into existing C/C++ codebases and ensures program support for legacy and non-Intel platforms.

This page intentionally left blank.

REFERENCES

[1] M. Schwarz, S. Weiser, D. Gruss, C. Maurice and S. Mangard, *"Malware Guard Extension: Using SGX to conceal cache attacks,"* 2017.

[2] S. Johnson, S. Vinnie, R. Carlos, E. Brickell and F. Mckeen, *Intel Software Guard Extensions: EPID Provisioning and Attestation Services*, 2016.

[3] S. Gueron, *A Memory Encryption Engine Suitable for General Purpose Processors*, 2016.

[4] John M., "Intel® Software Guard Extensions Tutorial Series: Part 1, Intel® SGX Foundation," Intel, 7 July 2016. [Online]. Available: https://software.intel.com/en-us/articles/intel-software-guard-extensions-tutorial-part-1-foundation. [Accessed 12 September 2017].

[5] S. Johnson, "Intel SGX: Debug, Production, Pre-release what's the difference?," Intel, 7 January 2016. [Online]. Available: https://software.intel.com/en-us/blogs/2016/01/07/intel-sgx-debug-production-prelease-whats-the-difference. [Accessed 12 September 2017].

[6] "Illegal Instructions Within an Enclave," Intel, [Online]. Available: https://software.intel.com/en-us/node/703005. [Accessed 12 September 2017].

[7] B. Xing, M. Shanahan and R. Leslie-Hurd, "Intel SGX Software Support for Dynamic Memory Allocation inside an Enclave," ACM, 2016.

[8] "sgx_sealed_data_t," Intel, [Online]. Available: https://software.intel.com/en-us/node/709220. [Accessed 12 September 2017].

[9] B. X-Lab, "Rust SGX SDK v0.2.0," [Online]. Available: https://github.com/baidu/rust-sgx-sdk/releases/tag/v0.2.0. [Accessed 12 September 2017].

[10] D. Rao, "Intel SGX Product Licensing," Intel, 28 February 2016. [Online]. Available: https://software.intel.com/en-us/articles/intel-sgx-product-licensing. [Accessed 12 September 2017].

[11] "Intel SGX SDK," Intel, [Online]. Available: https://software.intel.com/en-us/sgx-sdk/download. [Accessed 12 September 2017].

[12] "Intel SGX Installation Guide v1.9," Intel, [Online]. Available: https://01.org/intel-software-guard-extensions/documentation/intel-sgx-sdk-installation-guide. [Accessed 12 September 2017].

[13] "Intel SGX Developer Guide - Inputs Passed by Reference," Intel, 21 June 2017. [Online]. Available: https://software.intel.com/en-us/node/702975. [Accessed 12 September 2017].

[14] "C++ Language Support," Intel, [Online]. Available: https://software.intel.com/en-us/node/709059. [Accessed 13 September 2017].

[15] John M., "Refine the Enclave with Proxy Functions," Intel, 28 November 2016. [Online]. Available: https://software.intel.com/en-us/articles/intel-software-guard-extensions-tutorial-part-7-refining-the-enclave. [Accessed 12 September 2017].

[16] "Enclave Configuration File," Intel, [Online]. Available: https://software.intel.com/en-us/node/708992. [Accessed 12 September 2017].

[17] "Remote Attestation End-to-End Example," Intel, 8 July 2016. [Online]. Available: https://software.intel.com/en-us/articles/intel-software-guard-extensions-remote-attestation-end-to-end-example. [Accessed 12 September 2017].

[18] John M., "How to Create Dual Code Paths," Intel, 28 October 2016. [Online]. Available: https://software.intel.com/en-us/articles/intel-software-guard-extensions-tutorial-series-part-6-dual-code-paths. [Accessed 12 September 2017].

[19] "SoC and CPU System-Wide Approach to Security," Arm Holdings, [Online]. Available: https://www.arm.com/products/security-on-arm/trustzone. [Accessed 12 September 2017].

[20] "ARM Trusted Firmware," Arm Holdings, [Online]. Available: https://github.com/ARM-software/arm-trusted-firmware. [Accessed 12 September 2017].

[21] "AMD Secure Technology," AMD, [Online]. Available: http://www.amd.com/en-us/innovations/software-technologies/security. [Accessed 12 September 2017].

[22] T. Mandt, M. Solnik and D. Wang, "Demystifying the Secure Enclave Processor," 2016. [Online]. Available: https://www.blackhat.com/docs/us-16/materials/us-16-Mandt-Demystifying-The-Secure-Enclave-Processor.pdf. [Accessed 12 September 2017].

[23] "Intel Trusted Execution Technology Whitepaper," 2012. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/trusted-execution-technology/trusted-execution-technology-security-paper.html. [Accessed 12 September 2017].

[24] "Intel SGX SSL," Intel Open Source Technology Center, [Online]. Available: https://github.com/01org/intel-sgx-ssl. [Accessed 12 September 2017].

[25] "mbedtls-SGX," bl4ck5un, [Online]. Available: https://github.com/bl4ck5un/mbedtls-SGX. [Accessed 12 September 2017].

[26] H. Tadepalli, "Intel Quick Assist Technology with Intel Key Protection Technology in Intel Server Platforms Based on Intel Xeon Processor Scalable Family," Intel, 2017.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
29-January-2018	Technical Report	

4. TITLE AND SUBTITLE

Leveraging Intel SGX Technology to Protect Security-Sensitive Applications

5a. CONTRACT NUMBER
FA8721-05-C-0002 and/or FA8702-15-D-0001

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Joseph M. Sobchuk, Sean R. O'Melia, Dan M. Utin, Roger I. Khazan

5d. PROJECT NUMBER
1971

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

MIT Lincoln Laboratory
244 Wood Street
Lexington, MA 02421-6426

8. PERFORMING ORGANIZATION REPORT NUMBER

TR-1230

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Space and Missile Systems Center
483 N. Aviation Blvd.
El Segundo, CA 90245

10. SPONSOR/MONITOR'S ACRONYM(S)
SMC/MCD

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A: Approved for public release distribution unlimited.

13. SUPPLEMENTARY NOTES

13. ABSTRACT

This report explains the basic process by which Intel Software Guard Extensions (SGX) can be leveraged into an existing codebase to protect a security-sensitive application. Intel SGX provides user-level applications with hardware-enforced confidentiality and integrity protections. These protections apply to all three phases of the operational data lifecycle: at rest, in use, and in transit. SGX shrinks the trusted computing base (and therefore the attack surface) of the application to only the hardware on the CPU chip and the portion of the application's software that is executed within the protected enclave. The SGX SDK enables relatively straightforward integration into existing C/C++ codebases while still ensuring program support for legacy and non-Intel platforms.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER *(include area code)*
unclassified	unclassified	unclassified	unclassified	42	

This page intentionally left blank.