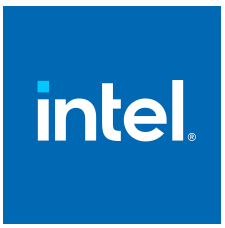


Intel® Software Guard Extensions (Intel® SGX)

Developer Guide

Legal Information

Revision History

Revision Number	Description	Revision Date
2.3	Intel® SGX Linux 2.3 release	September 2018
2.4	Intel® SGX Linux 2.4 release	November 2018
2.5	Intel® SGX Linux 2.5 release	March 2019
2.6	Intel® SGX Linux 2.6 release	June 2019
2.7	Intel® SGX Linux 2.7 release	September 2019
2.7.1	Intel® SGX Linux 2.7.1 release	November2019
2.8	Intel® SGX Linux 2.8 release	January 2020
2.9	Intel® SGX Linux 2.9 release	March 2020
2.9.1	Intel® SGX Linux 2.9.1 release	April 2020
2.10	Intel® SGX Linux 2.10 release	June 2020
2.11	Intel® SGX Linux 2.11 release	August 2020
2.12	Intel® SGX Linux 2.12 release	November 2020
2.13	Intel® SGX Linux 2.13 release	January 2021
2.14	Intel® SGX Linux 2.14 release	June 2021
2.15	Intel® SGX Linux 2.15 release	September 2021
2.15.1	Intel® SGX Linux 2.15.1 release	November 2021
2.16	Intel® SGX Linux 2.16 release	March 2022
2.17	Intel® SGX Linux 2.17 release	June 2022
2.18	Intel® SGX Linux 2.18 release	November 2022
2.19	Intel® SGX Linux 2.19 release	March 2023
2.20	Intel® SGX Linux 2.20 release	July 2023

Introduction

The Intel® Software Guard Extensions (Intel® SGX) Developer Guide provides guidance on how to develop robust application enclaves based on Intel SGX technology. This guide does not provide an introduction to the Intel SGX technology and it is not a secure coding guideline. This guide assumes that after assessing the benefits, costs and restrictions of developing with Intel SGX, you have decided to use this technology and now want to learn how to properly use it to develop sound application enclaves. With your knowledge of the Intel® SGX technology (see Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D) and experience on secure coding principles and practices, this guide will help you to develop your own application enclaves.

This document provides examples of many programming constructs and principles based on a hypothetical generic run-time system. The elements of this run-time system include the following:

- *Untrusted Run-Time System (uRTS)* – code that executes outside of the Intel SGX enclave environment and performs functions such as:
  - Loading and managing an enclave.
  - Making calls to an enclave and receiving calls from within an enclave.
- *Trusted Run-Time System (tRTS)* – code that executes within an Intel SGX enclave environment and performs functions such as:
  - Receiving calls into the enclave and making calls outside of an enclave.
  - Managing the enclave itself.
  - Standard C/C++ libraries and run-time environment.
- *Edge Routines* – functions that may run outside the enclave (untrusted edge routines) or inside the enclave (trusted edge routines) and serve to bind a call from the application with a function inside the enclave or a call from the enclave with a function in the application.
- *3rd Party Libraries* – for the purpose of this document, this is any library that has been tailored to work inside the Intel SGX enclave environment.

See the following Table Terminology for a definition of terms.

Terminology

Term	Definition

ECall	"Enclave Call" a call made into an interface function within the enclave
OCall	"Out Call" a call made from within the enclave to the outside application
Trusted	Refers to any code or construct that runs inside an enclave in a "trusted" environment
Trusted Thread Context	The context for a thread running inside the enclave. This is composed of: • Thread Control Structure (TCS) • Thread Data/Thread Local Storage – data within the enclave and specific to the thread • State Save Area (SSA) – a data buffer which holds register state when an enclave must exit due to an interrupt or exception • Stack – a stack located within the enclave
Untrusted	Refers to any code or construct that runs in the applications "untrusted" environment (outside of the enclave)

Enclave Programming Model

Intel® Software Guard Extensions (Intel® SGX) software, including an Intel SGX run time system, can be developed using standard tools and development environments. While the programming paradigm is very similar to conventional software, there are some differences in how the Intel SGX software is designed, developed and debugged to take advantage of the Intel SGX technology.

In this section, we compare the programming model available for developing enclaves and the programming model Independent Software Vendors (ISVs) are familiar with as the result of developing traditional applications for Android*, Linux*, OS X*, and Windows* operating systems. There are certain similarities that lower the barrier of entry to developers willing to adopt the Intel SGX technology. However, enclave writers must also be aware of the differences in how Intel SGX software is designed, developed and debugged to create robust enclaves. Features unique to Intel SGX such as attestation, provisioning and sealing are described in other sections of this document.

Enclave writers that understand the technology as well as the programming model it entails will extract the most benefit from Intel SGX. Developers must observe the following principles to develop application enclaves correctly. Failing to do so could result in a security vulnerability that could be exploited later on.

- An enclave is a monolithic software entity that reduces the Trusted Computing Base (TCB) for an application to a trusted runtime system, ISV code and 3rd party trusted libraries. A bug in one component may compromise the security properties of the enclave.
- The untrusted domain controls the order in which the enclave interface functions are invoked.
- When calling into an enclave, it is the untrusted domain who selects the Trusted Thread Context to be used within the enclave.
- There is no guarantee that the input parameters of a call into an enclave (ECall) or the return parameters from a call outside an enclave (OCall) will be what the enclave expects because the untrusted domain supplies them.
- The untrusted function invoked during an OCall may not perform the operations expected by the enclave.

- Anyone may load an enclave. Furthermore, an attacker may load an enclave with a program specifically developed to expose vulnerabilities in that enclave.

Enclave File Format

At a high-level, the Intel® SGX supporting software offers a programming model similar to what ISVs are used to from developing regular Android*, Linux*, macOS*, and Windows* applications, which is exposed through a DLL on Windows* OS, a Dynamic Library on macOS, and a Shared Object on Linux* OS and Android* OS.

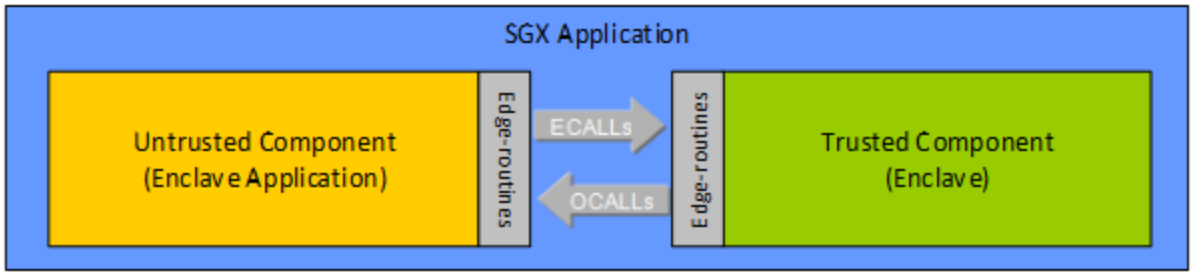
A regular DLL, Dynamic Library, or Shared Object file typically contains code and data sections corresponding to the functions and/or methods as well as the variables and/or objects implemented in the shared library. The operating system allocates a heap when the process that uses the shared library is loaded and a stack for each thread spawned within the process. Similarly, an enclave library file contains trusted code and data sections that will be loaded into protected memory Enclave Page Cache (EPC) when the enclave is created. In an enclave file, there is also an Intel SGX specific data structure, the enclave metadata. The metadata is not loaded into EPC. Instead, it is used by the untrusted loader to determine how to properly load the enclave in EPC. The metadata defines a number of trusted thread contexts, which includes the trusted stack, and a trusted heap initialized by the trusted runtime system at enclave initialization. Trusted thread contexts and trusted heap are required to support a trusted execution environment. The metadata also contains the enclave signature structure, which is a vital certificate of authenticity and origin of an enclave.

Even though an enclave can be delivered as a shared library file, defining what code and data is placed inside the enclave and what remains outside in the untrusted application is a key aspect of enclave development.

Enclaves, regardless on the number of trusted threads defined, must not be designed with the assumption that the untrusted application will invoke the ISV interface functions following a specific order. Once the enclave is initialized, an attacker may invoke any ISV interface function, arrange the calls in any order and provide any input parameters. Keep these ploys in mind to prevent opening an enclave up to attacks.

Enclave Trusted Computing Base (TCB)

Intel SGX Application



The first step in designing an Intel SGX enabled application is to identify the assets it needs to protect, the data structures where the assets are contained, and the set of code that operates on those data structures and then place them into a separate trusted library. Since the ISV knows the application best, the ISV should conduct a security analysis of the application and properly partition it making the decision about what code and data is placed in the enclave.

The code in an enclave is no different than code that exists as part of a regular application. However, enclave code is loaded in a special way such that once the enclave has been initialized, privileged code and the rest of the untrusted application cannot directly read data that resides in the protected environment, or change the behavior of the code within the enclave without detection. For this reason, even though identifying the secret processing components and the resources they use is an important step in any secure software development process, for using Intel SGX it is an essential activity.

Partitioning an application into trusted and untrusted components has additional implications from a security standpoint. It is generally accepted that a smaller memory footprint (smaller code and data) usually implies a lower chance of having defects in the final software product. It also implies simpler security analysis and safer software as a smaller attack surface can be exposed. Therefore, while it may be possible to move the majority of application code into an enclave, in most cases this would not be desirable. The TCB size should be a factor to consider when designing what goes inside an enclave. ISVs should attempt to minimize the enclave size, even though the Intel SGX architecture protects the enclave contents when the OS, VMM, or BIOS are compromised. The first generation of the Intel SGX architecture requires that all the functionality inside an enclave is statically linked in at build-time. This creates a performance/size trade-off which developers must carefully analyze as it impacts the TCB size. When using static library functionality, ISVs have two choices. They could provide a shim layer to call the functionality outside the enclave, or alternatively include the implementation of the library as part of the enclave. The first approach adds performance overhead compared to

normal library function calls. The second method causes an increase of the TCB size.

Partitioning also plays a key role in preparing an Intel SGX application to manage power events, see Section Power Management for additional details. The smaller the state information stored inside the enclave, the quicker the enclave will be able to backup this information outside the enclave and to recover from a power event.

Enclave Interface Functions (ECalls)

After defining the trusted (enclave) and untrusted (application) components of an Intel SGX enabled application, the developer should carefully define the interface between untrusted application and enclave. ISV trusted code is executed in the following scenarios:

- The untrusted application explicitly makes a call to an enclave interface function within the enclave, for example the application makes an ECall. Calling through an ISV interface function is the same as a regular application calling into a shared library.
- After a call made from within the enclave to the outside application (OCall) returns. Returning from an OCall is similar to what happens when a call from a shared library to another shared library returns; for instance after calling the Standard C library to perform an I/O operation. When an OCall returns, the trusted function that made the call outside the enclave continues execution.
- After an interrupts returns, the enclave code is also executed. However, the Intel SGX architecture ensures that execution within the enclave continues as if the interrupt never occurred. The same behavior is expected with interrupts that happen while a shared library function is executing.

An enclave must expose an API for applications to call in (ECalls) and advertise what services provided by the untrusted domain are needed (OCalls). The enclave writer defines the ECall and OCall functions that constitute the enclave boundary interface. Since ECalls expose the interface that an untrusted application may use, you should reduce the enclave attack surface by limiting the number of ECalls. You should also be aware that an enclave has no control on which ECall is executed, or the order in which ECalls are invoked. Thus, an enclave cannot depend on ECalls occurring in certain order. On the other hand, ISV interface functions can be invoked only after the enclave has been initialized, which means that:

- Any necessary address re-basing is performed successfully;
- Trusted global data, including security-centric data (for example, stack canary) are initialized successfully;
- Trusted thread context, including security-centric data (for example, stack guard pages) of the trusted thread the function is running on is initialized successfully;
- The implicit trusted initialization functions (for example, ISV global constructors) execute to completion.

As a special case of an ISV interface function, an ISV registered exception handler can only be invoked on a trusted thread where a supported enclave exception has happened and after the conditions above are met.

Enclave Inputs

Enclave inputs (and for this matter enclave outputs) can be observed and modified by untrusted code. The enclave writer must never trust any information coming from the untrusted domain and must always check ECall input parameters as well as OCall return values. When accepting inputs from outside the enclave, assumptions about the size and type of the values being passed in should be checked by the enclave software to assure correct behavior. After identifying the source and/or destination (remote entity, users, etc.) you should decide whether applying integrity protection and/or encryption with anti-replay and liveness protection checks are necessary to safeguard the information that at some point is exposed to the untrusted domain.

When an ISV interface function is invoked:

- The function arguments and any marshaled data of the pass-by-reference parameters are inside the trusted environment and not accessible to attackers;
- A read and/or write operation on the arguments, the return value and the marshaled reference, according to the parameter definitions specified by the enclave writer, will not compromise the ISV code/data confidentiality and integrity.
    - The argument, return value and the marshaled data are allocated and managed by the trusted runtime, not overlapping any ISV code or data.
    - The size of an argument, return value and the marshaled reference is as specified by the ISV (for example, the buffer size of the

marshaled data referenced by a pointer parameter is either specified by a constant, another parameter or a field in the fixed-size portion of the actual data).

Inputs Passed by Reference

Input arguments reside inside the enclave when the ISV interface function is invoked. However, when an input is passed by reference, only the reference (the pointer address) will be inside the enclave. The value referenced could be outside and change at any time. For instance, an attacker may change the value after the enclave code checks the function parameters.

The enclave writer must handle references or pointers with special care. An application may pass a pointer referencing a memory location within the enclave boundary, which may cause the enclave to inadvertently overwrite enclave code or data. Similarly, if the enclave software is not aware that a pointer references an untrusted location, the enclave may leak secrets. To prevent these issues, the enclave software must determine whether the memory region (specified by a pointer and size) is inside or outside the enclave linear range before dereferencing the pointer. Additionally, the enclave must ensure the data cannot be modified after it is checked. Developers should only pass through the enclave boundary interface pointers to objects of scope known inside the enclave. Thus pointers to C data structures are reasonable, but pointers to C++ objects are not.

Calls outside the Enclave (OCalls)

Enclaves cannot directly access OS-provided services. Instead, an enclave must do an OCall to an interface routine in the untrusted application. While calling outside adds a performance overhead, there is no loss of confidentiality. However, communication with the OS requires the release of data or the import of non-secret data, which needs to be handled properly.

Even though OCalls might be necessary sometimes, they are calls outside the enclave and therefore have associated some security risks.

- Enclave operations that require an OCall, such as thread synchronization and I/O, are exposed to the untrusted domain. An enclave must be designed in such a way that it prevents leaking side-channel information that would allow an attacker, who is looking at the untrusted functions called from an enclave, to gain insight into enclave secrets, see Section Protection from Side-Channel Attacks for additional information.
- An enclave must be prepared to handle the scenario where the OCall function is not performed at all. The return value from an OCall, which is an enclave input, comes from the untrusted domain and must not be relied upon. It might appear that an OCall has been successfully

completed when it has not. For instance, an attacker might drop an enclave's request to write sealed data to disk and tell the enclave the file was written successfully.
- An enclave cannot depend on nested ECalls occurring in certain order during an OCall. A developer may limit the ECalls that are allowed during a given OCall, since the state information (corresponding to the OCall in progress) can be stored inside the enclave. However, once an enclave makes an OCall there is no guarantee the untrusted domain will not recursively call into the enclave, and the enclave has no control over the order in which nested ECalls occur or the actual ISV interface functions invoked.

When an ISV function inside the enclave invokes an OCall:

- The OCall only exposes the OCall function arguments (including the referenced data) and the return value to the untrusted domain.
- When the OCall returns, the return value and any marshaled data of the pass-by-reference output parameters are inside the trusted environment (thus not accessible to an attacker) and the input-only function arguments (including the referenced data) are not changed. When the return value is a pointer, only the reference will be inside the trusted environment. The enclave software must check the data buffer referenced by the returned pointer like any other reference passed into the enclave.
- When the OCall returns, the trusted thread context is the same as before the OCall was made, except for the volatile registers and the output data on the trusted stack.

In certain scenarios, the enclave writer may avoid OCall functions by repartitioning the application and passing the information that an OCall is meant to obtain as an input parameter to an ISV interface function.

Nested ECalls (ECalls during OCalls)

You should be aware that when an OCall is made, it opens the door for nested ECalls. Once outside the enclave, an attacker trying to find vulnerabilities may invoke any ISV interface function exposed as an ECall to recursively call into the enclave. When an OCall is needed, you may reduce the surface attack blocking ISV interface functions such that nested ECalls are not allowed. For instance, you may store the state information (corresponding to the OCall in

progress) inside the enclave. However, an enclave cannot depend on nested ECalls occurring in certain order during an OCall. Initially, nested ECalls (ECalls during an OCall) are allowed and only limited by the amount of stack reserved inside the enclave. However, ISVs should be aware that such constructs complicates the security analysis on the enclave. When the need for nested ECalls arises, the enclave writer should try to partition the application in a different manner. If nested ECalls cannot be avoided, the enclave writer should limit the ISV interface functions that may be called recursively to only those strictly required.

NOTE:

The ISV interface functions can only be invoked after the enclave has been initialized. Thus nested ECalls are not allowed during the ISV global constructor functions.

Third Party Libraries

Earlier we mentioned that the enclave code must perform a thorough parameter checking for all the ISV interface functions; in other words, the enclave interface functions exposed to the untrusted domain. Such a recommendation also applies when working with a third party library. If a trusted library contains any function that is exposed, the ISV must confirm that the library provider also have this interface function check its input parameters exhaustively. However, if the top-level functions of a trusted library are meant to be called from inside the enclave only, or the trusted library is an enclavized version of an open-source implementation, the parameter checking might not be as strict. When a third party library does not sanitize its input parameters, and it is unpractical to change the third party code, then the enclave writer could add a wrapper that performs parameter checking to the API. This addition will not change the behavior or implementation of the third party library API, but removes the burden of validating the library again and simplifies future library updates.

Enclave Signature Structure

Establishing trust in software involved three main activities:

- Measurement: As an enclave is instantiated in a trusted environment, an accurate and protected recording of its identity is taken.
- Attestation: Demonstrating to other entities that a particular environment is instantiated in the correct manner.
- Sealing: Enabling data belonging to the trusted environment to be bound to it so that it can be restored only when the trusted environment is restored.

This section focuses on the first activity, measurement. Attestation and sealing activities are described in subsequent sections.

Enclaves include a self-signed certificate from the enclave author, also known as the Enclave Signature Structure (SIGSTRUCT). The enclave signature structure contains information that allows the Intel® SGX architecture to detect whether any portion of the enclave file has been tampered with. This allows an enclave to prove that it has been loaded in EPC correctly and it can be trusted. However, the hardware only verifies the enclave measurement when an enclave is loaded. This means that anyone can modify an enclave and sign it with his/her own key. To prevent this type of attack, the enclave signature structure also identifies the enclave author. The enclave signature structure contains several important fields that are essential for an enclave ability to attest to outside entities:

- Enclave Measurement – A single 256-bit hash that identifies the code and initial data to be placed inside the enclave, the expected order and position in which they are to be placed, and the security properties of those pages. A change in any of these variables will result in a different measurement. When the enclave code/data pages are placed inside the EPC, the CPU calculates the enclave measurement and stores this value in the MRENCLAVE register. Then the CPU compares the content of MRENCLAVE against the enclave measurement value in SIGSTRUCT. Only if they match with each other, the CPU will allow the enclave to be initialized.
- The Enclave Author's Public Key – After an enclave is successfully initialized, the CPU records a hash of the enclave author's public key in the MRSIGNER register. The contents of MRSIGNER will serve as the identity

of the enclave author. The result is that those enclaves which have been authenticated with the same key shall have the same value placed in the MRSIGNER register.

- The Security Version Number of the Enclave (ISVSVN) – The enclave author assigns a Security Version Number (SVN) to each version of an enclave. The SVN reflects the level of the security property of the enclave, and should monotonically increase with improvements of the security property. After an enclave is successfully initialized, the CPU records the SVN, which can be used during attestation. Different versions of an enclave with the same security property should be assigned with the same SVN. For example, a new version of an enclave with non-security-related bug fixes should have the same SVN as the older version.
- The Product ID of the Enclave (ISVPRODID) – The enclave author also assigns a Product ID to each enclave. The Product ID allows the enclave author to segment enclaves with the same enclave author identity. After an enclave is successfully initialized, the Product ID is recorded by the CPU, which can be used during attestation.

An enclave developer must provide the Security Version and Product ID of an enclave, as well as a signing key pair to generate the enclave signature structure. The CPU derives the identity of the enclave author from the public key whereas the private key is used to sign the enclave. The enclave measurement calculation must be performed based on the code and initial data to be placed inside the enclave, the expected order and position in which they are to be placed and the security properties of those pages. The code and initial data to be placed inside the enclave as well as the security properties of those pages are generated by the compiler, while their placement into the enclave is controlled by the enclave loader. Thus, the measurement calculation must follow the expected behavior of the enclave loader with regard to the manner of placing enclave code and initial data in the enclave.

Safeguarding the Enclave Signing Key

The enclave signing key is part of the enclave identity and it is critical to protect its secrets. An attacker who compromises the private signing key of an ISV might be able to:

- Write a malicious enclave that successfully attests to the identity of legitimate enclaves, and/or

- Write malware which uses the malicious enclave to compromise sealed data on individual platforms.

Proper key management practice should be employed to safeguard the private signing key, for example:

- Maintain minimum access to the private signing key.
- Use another enclave or a Hardware Security Module (HSM) to store the private signing key and perform enclave signing.
- Separate test signing from release signing using separate key pairs.

The SDK includes a tool for signing enclaves, called `sgx_sign`, that takes an enclave file and adds the enclave signature structure as required by the Intel® SGX architecture. This tool supports single-step test signing using a test signing private key configured on the local system, and two-step release signing that involves a signing facility/platform, where the release signing private key is protected. `sgx_sign` can also generate allowlisting materials from a signed enclave file.

Maintaining the Development Platform Clean

The ISV must maintain the development environment free from malware and other potential threads at all times. If the development platform is ever compromised, you cannot continue using the Intel SGX support software since it could be used to compromise the integrity of the enclaves built on that platform. At this point, the ISV must sanitize the platform before development can proceed.

Attestation

Attestation is the process of demonstrating that a piece of software has been established on a platform. In the case of Intel SGX, it is the mechanism by which a third entity establishes that a software entity is running on an Intel SGX enabled platform protected within an enclave prior to provisioning that software with secrets and protected data. Attestation relies on the ability of a platform to produce a credential that accurately reflects the signature of an enclave, which includes information on the enclave's security properties. The Intel SGX architecture provides the mechanisms to support two forms of attestation. There is a mechanism for creating a basic assertion between enclaves running on the same platform, which supports local, or intra-platform attestation, and then another mechanism that provides the foundation for attestation between an enclave and a remote third party.

NOTE:

The signing process involved in attestation takes place in such a manner that the relying party can be assured that it is communicating with a real hardware enclave and not some software emulation.

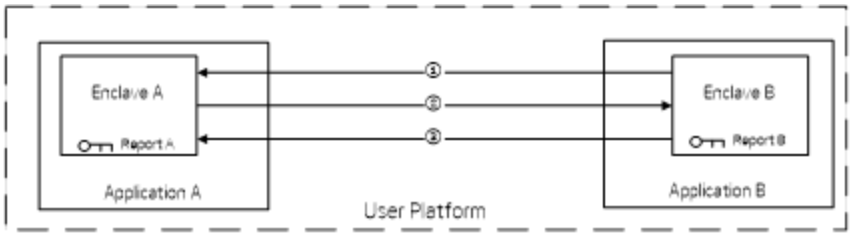
Local (Intra-Platform) Attestation

Application developers may wish to write enclaves which can co-operate with one another to perform some higher-level function. In order to do this, developers need a mechanism that allows an enclave to prove its identity and authenticity to another party within the local platform. Intel SGX provides a trusted hardware based mechanism for doing this. An enclave can ask the hardware to generate a credential, also known as report, which includes cryptographic proof that the enclave exists on the platform. This report can be given to another enclave who can verify that the enclave report was generated on the same platform. The authentication mechanism used for intra-platform enclave attestation uses a symmetric key system where only the enclave verifying the report structure and the enclave hardware creating the report know the key, which is embedded in the hardware platform.

An enclave report contains the following data:

- Measurement of the code and data in the enclave.
- A hash of the public key in the ISV certificate presented at enclave initialization time.
- User data.

- Other security related state information (not described here).
- A signature block over the above data, which can be verified by the same platform that produced the report.



Local Attestation Example

The figure Local Attestation Example shows an example flow of how two enclaves on the same platform would authenticate each other.

1. In the figure above, application A hosts enclave A and application B hosts enclave B. After the untrusted applications A and B have established a communication path between the two enclaves, enclave B sends its MRENCLAVE identity to enclave A.

   NOTE:
   Applications A and B can be the same application.

   There are two methods the application can use to retrieve the MRENCLAVE measurement for the enclave, either:

   - The application B retrieves the MRENCLAVE value from the enclave certificate for enclave B.
   - Enclave B supports an interface to export this value which is retrieved by creating a report with a random MRENCLAVE target measurement.
2. Enclave A asks the hardware to produce a report structure destined for enclave B using the MRENCLAVE value it received from enclave B. Enclave A transmits its report to enclave B via the untrusted application.
   - As part of his report request, enclave A can also pass in a data block of its choosing referred to as the user data. Inclusion of the user data in the report provides the fundamental primitive that enables a trusted channel to terminate in the enclave.

3. Once it has received the report from enclave A, enclave B asks the hardware to verify the report to affirm that enclave A is on the same platform as enclave B. Enclave B can then reciprocate by creating its own report for enclave A, by using the MRENCLAVE value from the report it just received. Enclave B transmits its report to enclave A.
4. Enclave A then verifies the report to affirm that enclave B exists on the same platform as enclave A.

Remote (Inter-Platform) Attestation

An application that hosts an enclave can also ask the enclave to produce a report and then pass this report to a platform service to produce a type of credential that reflects the enclave and platform state. This credential is known as quote. This quote can then be passed to entities off of the platform and verified to gain a remote provider's (also known as the relying party) or producer's trust.

Intel® SGX currently supports two types of remote attestation:

- Elliptic Curve Digital Signature Algorithm (ECDSA)
- Intel® Enhanced Privacy ID (Intel® EPID) Attestation

ECDSA Attestation

This method enables third-party attestation via the Intel® Software Guard Extensions Data Center Attestation Primitives (Intel® SGX DCAP). It is available on Intel® Xeon® Scalable processor-based server platforms including selected Intel® Xeon® E3 processors and Intel® Xeon-SP starting with 3rd Generation processors.

Features of ECDSA-based attestations:

- Takes advantage of 3rd Generation Intel® Xeon® Scalable processors for next-generation cloud-optimized platforms, 5G-ready networks, and next-generation virtual networks
- Provides flexible provisioning based on ECDSA certificates
- Allows for construction of on-premise attestation services
- Requires flexible launch control in supported Intel® platforms
- Provided under an open source licensing model

For detailed information, refer to the"Quoting Functions" section in the Intel® SGX Developer Reference.

EPID Attestation

This method supports Intel attestation via the Intel® Software Guard Extensions Platform Software (Intel® SGX PSW) package. It is available on selected client systems, selected Intel® Xeon E3 processors, and selected Intel® Xeon E processors. Ice Lake Xeon-SP (and the future Xeon-SP platforms) doesn't support EPID attestation.

Features of EPID-based attestations:

- Increased privacy protections
- Based on Intel EPID signatures
- Provisioning and attestation at workload runtime

Intel® Enhanced Privacy ID (Intel® EPID)

Attestation using standard asymmetric cryptographic signature algorithms has a well-known privacy concern when a small number of keys are used across the life of the platform. Because the key used for signing the quote needs to be associated with the hardware performing the quote operation, it allows third parties to collude and track which sites users have visited. To overcome this problem, Intel has introduced the use of an anonymous signature technique, known as Intel® Enhanced Privacy ID (Intel® EPID), for signing enclave quotes.

Intel EPID is a group signature scheme, which allows platforms to cryptographically sign objects while at the same time preserving the signer's privacy. With Intel EPID signature scheme, each signer in a group has their own private key for signing, but verifiers use the same group public key to verify individual signatures. Therefore, users cannot be uniquely identified if signing two transactions with the same party because a verifier cannot detect which member of the group signed a quote. In the case of Intel SGX, this group is a collection of Intel SGX enabled platforms.

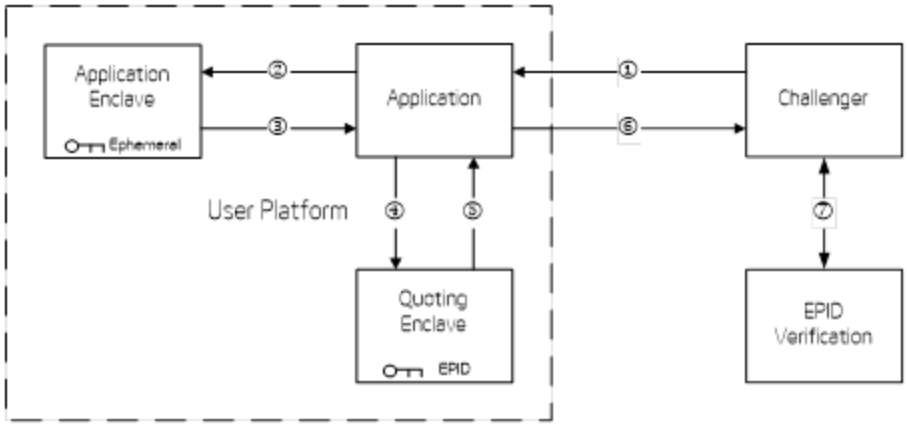
The Quoting Enclave

An Intel provided enclave known as the Quoting Enclave (QE) verifies the reports that have been created to its MRENCLAVE measurement value and then converts and signs them using a device specific asymmetric key, the Intel EPID key. The output of this process is called a quote, which may be verified

outside the platform. Only the QE has access to the Intel EPID key when the enclave system is operational. Therefore the quote can be seen to be emanating from the hardware itself but the CPU key is never exposed outside the platform.

Remote Attestation Process

The following figure shows an example of how an application, which has broken its processing into two component parts, provides attestation to a challenging service provider to receive some value added service from them.



Remote Attestation Example

The figure Remote Attestation Example shows the basic steps involved in canonical enclave attestation. Included in this diagram is the Quoting Enclave (QE). The steps in the figure are described below:

1. When the application needs a service from outside the platform, it first establishes communication with the service providing system. The service provider issues a challenge to the application to demonstrate that it is indeed running the necessary components of itself inside one or more enclaves. The challenge itself contains a nonce for liveness purposes.
2. The application requests a report from the application's enclave and passes in the nonce from the challenger.
3. The enclave generates a report structure and returns this structure along with a manifest to the application.
    a. The manifest contains those values which are included in the user data portion of the report.

   b. The manifest may include the nonce and an ephemerally generated public key to be used by the challenger for communicating secrets back to the enclave.
4. The report is delivered to the Quoting Enclave for signing.
   a. The Quoting Enclave authenticates the report.
   b. The Quoting Enclave converts the body of the report into a quote and signs it with the Intel EPID key.
5. The Quoting Enclave returns the quote structure requested.
6. The application returns the quote structure and any associated manifest of supporting data to the service challenger.
7. The challenger uses an Intel EPID verification service to verify the Intel EPID signature over the quote.
8. The challenger compares the enclave information in the quote against the trusted configuration and only renders the service to the application if the enclave information matches the trusted configuration. The challenger might enforce different trust policies, for example, only trusting a specific version of an enclave, identified by the measurement of the code and data in the enclave, or trusting all enclaves with a specific Product ID from a specific enclave author, identified by the hash of the public key in the ISV certificate. A trust policy must include enclave authorship and attributes check. For example, a debug enclave should never be trusted with any secret.

These steps serve as an example to illustrate one possible way that an enclave can be attested by a remote entity.

The trusted configuration mentioned in step 8 above is typically provided by the enclave author to the service provider. The mechanism for the service provider to acquire the trusted configuration is out of the scope of the remote attestation. One possible mechanism is that the service provider utilizes existing PKI infrastructure to verify the identity of the entity that's providing the trusted configuration information before accepting the trusted configuration information.

Privacy

Intel EPID name based (NB) Quotes only leave the platform encrypted with an Intel public key.

Malicious use of NB signatures (as unique IDs) can only occur if Service Providers (SP) collude somehow, for example, by lying about their identify or sharing private keys.

- License agreement between the SP and the Attestation Service will prohibit collusion, under penalty of the Attestation Service revoking an offending SP by no longer verifying attestations.

NB quotes are treated as unique identifiers; NB Quotes only being meaningful to a single Service Provider is not enough to waive this. Therefore, user opt-in is still required before transmitting them.

- License agreement between the SP and the Attestation Service will require the Intel® SGX application that communicates with the SP to be responsible for getting the user opt-in under penalty of the Attestation Service revoking an offending SP/application by no longer verifying attestations.
    - Opt-in needs to be “above and beyond” EULA acceptance.

Distinguishing between Running Enclave Instances

Intel SGX does not provide a direct mechanism (for example, through the automatically generated REPORT fields) to distinguish between two (or more) running instances of an enclave. Two running instances of an enclave cannot be distinguished by the automatically generated data in their REPORT’s alone. To do this, you must add a nonce to the protocol you use to establish trust in the underlying enclave. To establish trust in the underlying enclave, use the RDRAND functionality of the hardware and ensure this is submitted (directly or indirectly through a cryptographic hash) as part of the userdata field included in the REPORTs exchanged between enclaves. For more information of the RDRAND functionality, see Random Number Generation.

Secret Provisioning

A remote entity may provision some secrets to an enclave after remote attestation has been accomplished. Typically, secret provisioning is conducted through a secure channel. The secure channel establishment must be bound to the remote attestation process. Otherwise, the remote server might provision the secret to an entity other than the enclave that has been attested.

Step 3.b in the attestation flow referenced the ability to include a public key to facilitate the creation of a trusted channel. To accomplish this, in step 3 the enclave may wish to authenticate the server first, to ensure that it is about to receive a secret from a trusted entity. A known good root certificate can be embedded within the enclave code or initialization data for example, allowing the enclave to validate the server. Once the server has been authenticated, the enclave can generate an ephemeral public/private key pair and include the public key in the user data portion of the report.

After the enclave has been validated in step 7 of the attestation flow, the server can generate an encryption key E, and encrypt it with the enclave's public key P, and send P(E) over the channel to the application. The channel itself does not need to be protected, because the secret is encrypted.

Once P(E) has been received by the application, it can be passed to the enclave. Inside the enclave then, it can decrypt P(E) since it possesses the private key that is associated with this ephemeral public/private key pair, so now both the challenger and the enclave possess the encryption key E.

Similar to attestation, this is not the only way that a trusted channel can be established between an enclave and a remote entity to exchange a secret, just one example.

The verifier of the remote attestation must check the identity of the signer (MRSIGNER) before provisioning any secrets. The Intel SGX architecture does not verify the certificate chain when an enclave is instantiated. The hardware only verifies the enclave measurement (MRENCLAVE) and saves a hash of the ISV public key in the MRSIGNER field of the enclave signature structure. This means that anyone can modify an enclave and re-sign it. Similarly, the verifier must also check the enclave attributes to prevent provisioning any secret to a debug enclave, for instance.

Once a secure channel has been established, secrets can be provisioned to the enclave. The challenger can now encrypt a secret S with the key E, and send E(S) to the application, which in turn passes it to the enclave. The enclave can now use E to decrypt E(S), and now it possesses S. It would be

inconvenient, however to require the enclave to connect to the remote entity for secret provisioning every time the enclave is instantiated. Instead, the enclave may choose to store the secret in non volatile storage using the sealing techniques discussed in the next section. Even when the secret is sealed outside the enclave, the secret remains inaccessible to anyone but to the enclave that sealed it, and only on the platform on which it was sealed.

Secret provisioning is a critical feature enabled by the Intel® SGX technology. It allows building enclaves that are more robust than current Tamper Resistant Software (TRS). TRS typically provides security through obscurity, for instance it obfuscates secrets in the executable in an attempt to keep secrets safe from unauthorized observation. However, this approach simply makes it time-consuming, but not impossible, to extract secrets embedded in a TRS binary. Furthermore, it is a complex technique for developers to use and its practice is discouraged.

Debug (Opt-in) Enclave Considerations

Data provisioned to a debug enclave is not secret. A debug enclave's memory is not protected by the hardware so it may be inspected and modified using the Intel SGX debugging instructions. The enclave attributes, which include the debug flag, are contained in the report and quote that provide the enclave credentials. To protect all secrets provisioned to production enclaves, local and remote entities must check the enclave attributes and exchange special debug secrets during the development process but refrain from provisioning any secret to a debug enclave.

Disposal of Enclave Secrets

Enclave secrets may be safely stored outside the enclave boundary after such secrets are properly sealed. However, there are certain instances where a secret, such as the seal key, needs to be disposed of inside the enclave. The enclave writer must use the `memset_s()` function to clear any variable that contained secret data. The use of this function guarantees that the compiler will not optimize away the write to memory intended by this function call and thus ensuring the secret data is cleared. Using `memset_s()` is especially important when secret data is stored in a dynamically allocated buffer. After such a buffer is freed it could be reallocated and its previous contents, if they are not erased, may be leaked outside the enclave.

The implementation of `memset_s()` is not performance optimized so the use of `memset()` is appropriate to initialize buffers and clear buffers that do not contain secret data.

Sealing

When an enclave is instantiated it provides (confidentiality and integrity) protection to the data by keeping it within the boundary of the enclave. Enclave developers should identify enclave data and/or state that is considered secret and potentially needs to be preserved across the following events (when the enclave is destroyed):

- The application is done with the enclave and closes it.
- The application itself is closed.
- The platform is hibernated or shutdown.

In general, the secrets provisioned to an enclave are lost when the enclave is closed. But if the secret data needs to be preserved during one of these events for future use within an enclave, then it must be stored outside the enclave boundary before closing the enclave. In order to protect and preserve the data, a mechanism is in place which allows enclave software to retrieve a key unique to that enclave. This key can only be generated by that enclave on that particular platform. Enclave software uses that key to encrypt data to the platform or to decrypt data already on the platform. We refer to these encrypt and decrypt operations as sealing and unsealing, respectively as the data is cryptographically sealed to the enclave and platform.

Software Sealing Policies

When sealing data, the enclave needs to specify the conditions which need to be met when the data is to be unsealed. There are two options available.

Seal to the Current Enclave (Enclave Measurement)

Sealing to the current enclave uses the current version of the enclave measurement (MRENCLAVE), taken when the enclave was created, and binds this value to the key used by the sealing operation. This binding is performed by the hardware through the EGETKEY instruction.

Only an enclave with the same MRENCLAVE measurement will be able to unseal the data that was sealed in this manner. If the enclave DLL, Dynamic Library, or Shared Object file is tampered with, the measurement of the enclave will change. As a result, the sealing key will change as well, and the data cannot be recovered.

Seal to the Enclave Author

Sealing to the enclave author uses the identity of the enclave author, which the CPU stores in the MRSIGNER register at enclave initialization time, and binds this value to the key used by the seal data function. This binding is performed by the hardware through the EGETKEY instruction. The key used by the seal data function is also bound to the Product ID of the enclave. The Product ID is stored in the CPU when the enclave is instantiated.

Only an enclave with the same value in the MRSIGNER measurement register and the same Product ID will be able to unseal data that was sealed in this manner.

The benefit of offering this mechanism over sealing to the enclave identity is two-fold. First, it allows for an enclave to be upgraded by the enclave author, but does not require a complex upgrade process to unlock data sealed to the previous version of the enclave (which will have a different MRENCLAVE measurement) and reseal it to the new version. Second, it allows enclaves from the same author to share sealed data.

Enclave authors have the opportunity to set a Security Version Number (SVN) when they produce the enclave. This security version number is also stored in the CPU when the enclave is instantiated. An enclave has to supply an SVN in its request to obtain the seal key from the CPU. The enclave cannot specify a version beyond the SVN used when the enclave was created (ISVSVN). This would give the enclave access to a seal key to which it is not entitled. However, the enclave may specify an SVN previous to the enclave's ISVSVN. This option gives an enclave the ability to unseal data sealed by a previous version of the enclave, which would facilitate enclave software updates, for instance.

Sealing and Unsealing Process

The high level process for sealing data within an enclave is as follows:

1. Allocate memory within the enclave for the encrypted data and the sealed data structure which includes the payload consisting of both the data to encrypt and the Additional Authentication Data (AAD). AAD refers to the additional data/text that will be part of the MAC calculation but will not be encrypted (for example, it will remain plain text/data in the seal data structure). The AAD may include information about the application enclave, version, data, and so on.

2. Call the seal data API to perform the sealing operation. An example seal operation algorithm is:
    a. Verify the input parameters are valid. For instance, if a pointer to a sealed data structure is passed as a parameter, the buffer it points to must be inside the enclave.
    b. Instantiate and populate a key request structure used in the EGETKEY operation to obtain a seal key:
        i. Call EREPORT to obtain the ISV and TCB Security Version Numbers, which will be used in the key derivation.
        ii. Key Name: Identifies the key required, which in this case is the seal key.
        iii. Key Policy: Identifies the inputs required to be used in the key derivation. Use MRSIGNER to seal to the enclave's author or MRENCLAVE to seal to the current enclave (enclave measurement). Reserved bits must be cleared.
        iv. Key ID: Call RDRAND to obtain a random number for key wear-out protection.
        v. Attribute Mask: Bitmask indicating which attributes the seal key should be bound to. The recommendation is to set all the attribute flags, except Mode 64 bit, Provision Key and Launch key, and none of the XFRM attributes.
    c. Call EGETKEY with the key request structure from the previous step to obtain the seal key.
    d. Call the encryption algorithm to perform the seal operation with the seal key. It is recommended to utilize a function that performs AES-GCM* encryption/decryption, such as the Rijndael128GCM, which is available in the Intel® Integrated Performance Primitives Cryptography library.
    e. Delete the seal key from memory to prevent accidental leaks.
3. Save the seal data structure (including the key request structure) to external memory for future use within an enclave. The key request structure will be used in future enclave instantiation(s) to obtain the seal key required for the decryption process.

The high level process for unsealing data within an enclave is as follows:

1. Allocate memory for the decrypted data.
2. Call the unseal data API to perform the unsealing operation. An example unseal operation algorithm is:
    a. Verify the input parameters are valid.
    b. Retrieve the key request structure used in conjunction with the seal data structure.
    c. Call EGETKEY with the key request structure to obtain the seal key.
    d. Call the decryption algorithm to perform the unseal operation with the seal key.
    e. Delete the seal key from memory to prevent accidental leaks.
    f. Confirm that the hash tag generated by the decryption algorithm matches the tag generated during encryption.

Distinguishing between Enclave Instances

Enclave writers should be aware that even though two running instances of the same enclave can be distinguished at the time they attest, there is no Intel® SGX mechanism to prevent one enclave instance from having access to the sealed data of another enclave when both enclaves use the EGETKEY instruction. Both instances will return the same key value for the enclave – this is a basic premise for keeping data secret across power cycles. This is also true when the same enclave is executing in different virtual machines.

In order to provide SEAL key separation between enclave instances, including those executing in a virtualized environments (e.g. datacenter or cloud), Intel recommends that enclave writers use the KEYID field of the KEYREQUEST structure passed into the EGETKEY instruction to pass an enclave instance specific nonce. This will provide a different key even when the same enclave is in a different virtual machine.

Should you need to keep separate TCB's over different instances, it is recommended that the enclave writer assign a different identity to the enclave through the enclave signature mechanism.

For more information about differentiating between enclave instances and the attestation mechanism, see Distinguishing between Running Enclave Instances.

Data Migration across Platforms

Before the Intel SGX technology, the hardware platform was never part of the TCB for encrypting user data. This allowed the user to easily migrate their

data, even if it was encrypted, from one platform to another. Now the CPU is used to help determine the enclave's sealing key. Therefore, migrating a user's data from one platform to the next now requires careful planning.

If an application is moved from an old Intel SGX system to a new Intel SGX system (platform upgrade) or from one processor to another (CPU replacement in a system or load balancing in a cloud environment) the enclave will not be able to unseal the data in the new platform. Data migration typically requires a back-end server that verifies the identity of the enclave on the old system and the enclave on the new system, and facilitates the key exchange between the two systems to share the data. Regardless of the specific method that an ISV uses to migrate data, the seal key should not be shared outside an enclave because it could compromise all data previously sealed by the enclave.

Debug (Opt-in) Enclave Considerations

The Intel SGX architecture includes the debug flag, as well as other enclave attributes specified by the developer in the key request structure, in the seal key derivation. Two identical enclaves launched in debug and non-debug mode respectively, will get different seal keys. This mechanism protects the data sealed by a production enclave, since it cannot be unsealed by a debug enclave.

Processor Features

Aside from a few exceptions (described in Illegal Instructions within an Enclave), code executing within an enclave can execute most of the instructions available to software executing at Ring 3. This includes cryptographic acceleration instructions such as the Intel® Advanced Encryption Standard New Instructions (Intel® AES-NI) Set and the facility to generate trustworthy random values, rooted in the hardware.

Hardware Features

Functionality wise, hardware features can be assured using the enclave attributes. An enclave will fail at initialization if the platform does not support the requested attributes. However, some implementations may not support the same processor features. Software should consider that certain attribute may not be supported in all implementations before using these features inside the enclave.

Running an Enclave with Validated Features

An enclave writer typically depends on the compiler and libraries to utilize the appropriate Extended CPU feature instructions. This means that he/she does not know whether the generated enclave code utilizes any specific Extended CPU feature. The untrusted loader follows a conservative approach and attempts to enable all the Extended CPU Features available on the platform (supported by the CPU and enabled by the OS). However, an advanced enclave writer can override the default settings.

The Enclave Signature Structure (SIGSTRUCT) contains an ATTRIBUTES and ATTRIBUTEMASK fields. The entire ATTRIBUTES field, which includes the X-Features Request Mask (Extended CPU features mask or XFRM) subfield, is integral part of an enclave's identity (for example, its value is included in the reports generated by the Intel SGX platform, and arbitrary bits from this field can be included in key-derivation requests for keys). Together, the ATTRIBUTES and ATTRIBUTEMASK dictate what Extended CPU features must be enabled on the platform before the Intel SGX architecture initializes an enclave.

If a bit in SIGSTRUCT.ATTRIBUTEMASK is set to 1, the untrusted loader will have the corresponding enclave ATTRIBUTES and SIGSTRUCT.ATTRIBUTES bits match each other. This means that the corresponding X-Feature will be enabled or disabled based on the SIGSTRUCT.ATTRIBUTES bit and whether said X-Feature is enabled on the platform. If a specific Extended CPU feature

is requested (SIGSTRUCT.ATTRIBUTE is 1) but it is not enabled on the platform the enclave will fail to initialize. On the other hand, the Intel SGX architecture will disable any Extended CPU feature enabled on the platform that is not desired (SIGSTRUCT.ATTRIBUTE is 0). When a bit in SIGSTRUCT.ATTRIBUTEMASK is not set, then the untrusted loader will attempt to enable the corresponding Extended CPU feature (default settings).

To ensure that an enclave will only run with features that have been validated and prevent using a configuration that could compromise the enclave's behavior, set the ATTRIBUTEMASK bits corresponding to the appropriate X features to 1, and set the ATTRIBUTES bits to 1 or 0 depending on whether the specific Extended CPU feature should be enabled or disabled, respectively. Similarly, to guarantee that an enclave does not run in a future processor with a feature that is currently undefined the Intel SGX architecture requires setting the reserved ATTRIBUTEMASK bits to 1 and the reserved ATTRIBUTES bits to 0 (in SIGSTRUCT).

NOTE:

Do not rely on the enclave attributes to safeguard protected data. An attacker could sign an enclave with different attributes in an attempt to have the enclave crash and leak some secrets. In this scenario, however, the enclave will report a different MRSIGNER during attestation. As long as secrets are not provisioned to an enclave that has not been signed with the ISV key, a well-designed enclave that crashes because it is run with unexpected hardware features will not leak any secrets.

Random Number Generation

A good source of entropy is required to build a high-quality random number generator. The RDRAND instruction provides access to the hardware implementation of the underlying Digital Random Number Generator (DRNG). However, there are some circumstances when the RDRAND instruction may fail. When this happens, the recommendation is to try again up to ten times. Software vendors that have an existing Pseudo-Random Number Generator (PRNG) should use the RDSEED instruction to benefit from the high-quality entropy source of the Intel® Secure Key, rather than seeding the PRNG with some value contained in the enclave binary file, since an attacker would have access to it. Depending on uninitialized memory as a source of entropy to seed the PRNG is not a recommended either. Intentional references to uninitialized memory make the code difficult to understand and analyze and alone does not guarantee the randomness of the data collected. Additionally,

debugging tools will warn enclave developers about code where uninitialized memory is being used. However, tracking down the source of the uninitialized memory is not straightforward task.

Illegal Instructions within an Enclave

The following is a list of hardware instructions which are illegal within an enclave and will generate a #UD fault if executed:

1. Instructions which may VMEXIT if executed inside an enclave. Since it is not permissible for the VMM to update the enclave, they are not allowed.

   CPUID, GETSEC, RDPMC, RDTSC, RDTSCP, SGDT, SIDT, SLDT, STR, VMCALL, VMFUNC.

2. I/O instructions cannot be executed inside an enclave. These instructions could cause faults which cannot be handled by software.

   IN, INS/INSB/INSW/INSD, OUT, OUTS/OUTSB/OUTSW/OUTSD.

3. Instructions that may require a change in privilege levels.

   Far call, Far jump, Far ret, INT n/INTO, IRET, LDS/LES/LFS/LGS/LSS, MOV to DS/ES/SS/FS/GS, POP DS/ES/SS/FS/GS, SYSCALL, SYSENTER.

Developers should consider these instructions with respect to standard I/O and system functions, which depend on these HW instructions for their underlying implementation. Functions which gather host system attributes, perform I/O, or require a higher privilege level should be performed outside an enclave. In some cases, a developer may have access to trusted alternatives such as trusted time and trusted I/O. This functionality however is not provided directly by the Intel SGX architecture and it is currently out of the scope of this document.

CPUID Information

The CPUID instruction is also illegal inside the enclave. Thus software that retrieves CPUID information must do so outside the enclave. Therefore, this information cannot be assured from a security viewpoint and should be used carefully.

An enclave writer may write a custom untrusted function for gathering host system state, which may include CPUID values, system environment variables, and additional application attributes. The results from a specific CPUID leaf could then be preserved inside the enclave (via a specific ECall) to avoid the overhead associated with performing an OCall to execute the CPUID

instruction outside an enclave. The key point is that this information is gathered in the untrusted domain and thus the application enclave should design and validate for the scenario in which unexpected or inconsistent data is provided.

One additional consideration is the use of third party libraries which are dependent on the CPUID instruction and have not been modified for Intel SGX compatibility. In this case, the ISV must write a custom exception handler to catch the #UD fault caused by CPUID. In creating the custom exception handler, the ISV should:

1. Determine which CPUID leafs are required by the third party library.
2. Provide an initialization routine to gather the CPUID data needed by the third party library and cache it inside the enclave.
3. Write a custom exception handler for a #UD fault on a CPUID instruction and provide the results for the leaf requested in the failing CPUID instruction. The exception handler must also advance the Instruction Pointer to bypass the CPUID instruction. OCalls are not permitted in exception handlers and thus CPUID data must be obtained during initialization.

Programming for Performance

The security that is offered by the Intel® SGX architecture does not come for free in terms of impact of the performance on your application. Generally speaking Intel will seek to minimize the effect of the security checks and mechanisms that are required to support the security model offered by Intel SGX but some general awareness of where performance can be impacted will prove a useful tool to those seeking to get the best performance from their application.

Developers that understand the potential overhead in the areas described in this section and apply the recommendations made here, can successfully create applications that do not experience these addressable performance issues.

Enclave Creation

Enclave creation is the first area to consider. As the following discussion makes clear, enclave size greatly affects the time it takes to create an enclave because enclave measurement occurs to ensure the code loaded into the enclave is trusted.

During enclave creation, a series of EADD and EXTEND instructions are run to load and measure enclave pages.

- Each EADD instruction records EPCM information in the cryptographic log stored in the SECS and copies 4 Kbytes of data from unprotected memory outside the EPC to the allocated EPC page.
- Each EEXTEND instruction measures a 256 byte region (generating a cryptographic hash), which means it takes 16 invocations of EEXTEND to measure the 4KB page created by EADD. Each of the 16 invocations of EEXTEND adds to the cryptographic log information and to the measurement of the section.

Since cryptographic processing takes processor cycles, the time to create an enclave scales directly with the size of the enclave, because each additional 4KB page that an enclave uses results in cryptographic processing that occurs for the EADD instruction and cryptographic processing for the 16 EEXTEND instructions to measure that 4KB page. Creation of the enclave is visible to the application.

If you are concerned that the enclave creation time for your application is impacting its overall performance, consider the following approaches to reduce this impact:

- Reduce the size of your enclave. Scrutinize each code and data element currently in your enclave and move every element that does not reasonably need to be there to the untrusted part of your application. (You can use Intel® VTune™ Amplifier to help experiment with how long it takes to create various size enclaves.)
- Enclave Dynamic Memory Management (EDMM) available with Intel® SGX allows enclaves to expand after creation. When your OS supports EDMM, your application can create a smaller size enclave at first, then expand it later when additional enclave space is needed. This shifts some of the time to copy pages and measure regions from enclave creation time to a later point in time.
- Look for ways to hide the enclave loading time by having the application perform processing that occupies the user's attention.
- Avoid excessive enclave destruction/reloads to minimize repeating load overhead.

NOTE:

This discussion includes a subset of enclave creation actions. For details, see Constructing an Enclave in Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 3D.

Enclave Transitions

Transitioning control to/from enclaves is the second area to consider. The following discussion explains how the amount of data marshalled back and forth between the untrusted part of an application and the enclave greatly affects transition time to/from enclaves. Also discussed is balancing the time an application spends executing in an enclave versus how often that application enters/exits the enclave.

Transitions to an enclave and from an enclave resemble a context switch in many ways. When an EENTER instruction is executed to transition control into an enclave, register state and other information regarding the untrusted state is saved; then inside the enclave thread state and other information regarding the trusted state is loaded so execution can begin in the enclave. Much of this is performed by SDK generated code. A reverse process occurs on the

transition from an enclave, initiated by an EEXIT instruction. The trusted thread state information is saved; then untrusted register state and other information is loaded. Security checks are also performed during all transitions. Again, much of this is performed by SDK generated code. These actions constitute a fixed element of the overhead associated with transitioning to/-from enclaves.

However, there is also a variable element of the overhead associated with transitions at runtime. Parameters are marshalled from the untrusted part of the application to the trusted part, and return values are un-marshalled. In the trusted part, parameters from the untrusted part are un-marshalled and return values for the untrusted part are marshalled. Since parameters can vary greatly in size, if an application is passing large parameters between the two parts of an application, a noticeable overhead can be experienced.

If the overhead associated with transitions to/from enclaves for your application is impacting its performance, consider the following approaches to reduce this impact:

- Reduce the total size of parameters being passed. Examine each parameter currently being passed between the untrusted part of the application and the enclave and remove non-critical parameters. If possible, reduce the remaining parameters to their smallest reasonable size. You can use Intel® VTune™ Amplifier to compare transition time with frequency of transitions to help achieve a healthy performance balance.
- If the code in your enclave needs to operate on large data structures, you may wish to pass a pointer to the data structure into the enclave instead of the actual data. This is allowed using the `user_check` parameter with pointers in the Enclave Definition Language (EDL) file. However, there is a security risk because the Edger8r tool does not verify the pointer before passing it to the enclave when `user_check` is used. You as the developer must ensure that you are not exposing secrets in untrusted memory. You must also implement your own pointer verification if you choose this method. For details on the EDL `user_check` parameter, see this discussion in the Intel® Software Guard Extensions SDK Developer Reference: https://software.intel.com/en-us/node/708978. For details on the functions used to determine if a pointer and its associated data is inside or outside the enclave, see this discussion in the Intel® Software Guard Extensions SDK Developer Reference: https://software.intel.com/en-us/node/709040.
- If transition time is a concern, you may want to investigate other

approaches, such as implementing an Exit-less Service. With exit-less services, parameters are held in a buffer in the untrusted part of the application only; the trusted part polls the buffer looking for notification to perform its work on the data. So no transition occurs. This trades transition overhead for dedicating an enclave thread for additional polling, which may be appropriate in some cases. The following link allows you to download a paper describing exit-less based services: https://sites.google.com/site/silbersteinmark/Home/cr-eurosys17sgx.pdf.

NOTE:

- When an interrupt occurs while executing in an enclave, an Asynchronous Exit (AEX) occurs, which results in a transition from the enclave. An AEX has more overhead associated with it than a standard (non-Intel® SGX) interrupt context switch. Since interrupts are under the control of the OS, there is nothing you can do in your application to control this. But it's good to understand that interrupts may effect perceived application performance.
- This discussion describes a subset actions that occur during enclave transitions. For details, see Enclave Entry and Exiting in Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 3D.
- For more information on data marshalling, see the discussion of Proxy Functions in Part 7 of the Intel® SGX Tutorial Series.

Excessive Cache Misses

The third area to consider is the effect of cache misses, which in some situations can result in an overhead increase for Intel® SGX applications. The general actions (and overhead) associated with a cache write to system memory or a cache line fill in a non-Intel® SGX enabled system are very well known. However, Intel® SGX adds another dimension to cache misses, as all memory transactions that are outside the portion of the processor cache used for an enclave are protected. This protection adds some overhead when fetching cache lines from the memory. This overhead will be model-specific to the implementation of Intel® SGX. (A technical whitepaper describing the initial implementation can be found at: http://eprint.iacr.org/2016/204.pdf.)

Intel® SGX architecture adds two potential overhead elements to each cache miss beyond typical cache overhead:

- The time to perform integrity check/anti-replay check for each cache line not currently in the processor cache, and to update the data structure in system memory (if needed). This overhead depends on the memory access pattern.
- The time to encrypt/decrypt the actual data being moved between the cache and system memory.

This additional overhead could become significantly greater as the frequency of cache misses increases. (Note that memory access inside the enclave already in the cache are not impacted. And accessing memory outside the enclave from inside the enclave has little impact.)

If your application is experiencing overhead associated with a high number of cache misses, consider taking the following steps:

- Reduce the size of your enclave's data. Inspect your data to ensure that only critical elements are in the enclave. Less data means less encryption/decryption and less data structure checking by the Intel® SGX memory control/protection mechanism. You can use Intel® VTune™ Amplifier to inspect caching behavior in your application to help make tuning decisions.
- Consider the recommendations of the following sources to help create a more "cache friendly" application:

  https://software.intel.com/en-us/articles/resolve-cache-misses-on-64-bit-intel-architecture.

  https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

Excessive Writing of Pages

The fourth area to consider is how extensive page writes in Intel® SGX applications can increase overhead, and how to minimize that effect. Typically operating systems support paging, which includes some overhead; however, the overhead associated with paging in Intel® SGX is greater, as described below.

Intel® SGX uses secure storage, called the Enclave Page Cache (EPC), to stored enclave contents. Enclave pages are 4KB in size. When enclaves are larger than the total memory available to the EPC, enclave paging can be used by

privileged SW to evict some pages to swap in other pages. The CPU performs the following steps using the EWB instruction when the OS swaps out an enclave page:

- Read the Intel SGX page to be swapped out (evicted)
- Encrypt the contents of that page
- Write the encrypted page to unprotected system memory

Since this process has an inherent overhead associated with it, the more pages that are swapped out, the more often the overhead is incurred.

To prevent your application from experiencing the additional overhead associated with excessive page writes, do what you can to ensure enclave size is less than the EPC. By including only secrets and the code to operate on them in your enclaves, you can help minimize the chances of incurring paging overhead. You can use Intel® VTune™ Amplifier to inspect paging behavior in your application to help make tuning decisions.

NOTE:

- While developers have control over use of finite resources in their application design, users/usages also have significant control over how many applications are running. If users/usages end up causing a lot of enclaves to be run, enclave performance can be impacted, despite the efforts you put into performance optimization.
- The following technical forum discussion provides additional details related to paging of enclave code/data: https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/722444.

Additional Performance Notes

If your application is multi-threaded, look into optimizing data synchronization, locking, the threading model, and the memory allocation algorithm selected to improve performance.

- The Intel® SGX SDK (for Microsoft* Windows* and Linux*) has some synchronization and locking primitives that are already optimized.
- For heavily threaded applications, it may be better to select one memory allocation algorithm over another. The Intel® SGX SDK for Linux supports TCMalloc memory allocation algorithm, which can result in a sig-

nificant performance increase for many heavily-threaded applications over the default dlmalloc alternative.

Defense In-depth Mechanisms

The Intel® SGX software stack supports standard defense-in depth mechanisms such as stack probing, buffer overflow protection and, on Windows OS, safe structured exception handling. Enclave writers should set the compiler options such that by default enclaves are built with standard defense in-depth mechanisms available on a given platform. Regarding stack buffer overflow protection, developers must be aware that the commonly used compiler options only provide protection when the buffer meets certain criteria. For instance, Microsoft* Visual Studio compiler option `/GS` and GNU* compiler option `-fstack-protector` do not provide protection when the size of the buffer in stack is below certain threshold to avoid significant performance penalty. The enclave writer must evaluate whether this security check should be enabled in enclave functions that would remain unprotected otherwise (enclave interface functions, for instance) and apply more strict checking options, such as Visual Studio compiler option `/sdl` and GNU compiler options `-fstack-protector-all`, `-fstack-protector-strong`, and `-fstack-protector-explicit`, to specific modules. GNU compiler supports options `-fstack-protector-strong` and `-fstack-protector-explicit` in version 4.9.2. Address Space Layout Randomization (ASLR) is not supported within an enclave. However, the randomization of the load address of the enclave is dependent on the operating system. Different versions of Windows* may randomize (or not randomize) the location differently. A compromised loader or OS (both of which are outside the TCB) can remove the randomization entirely. The enclave writer should not rely on the randomization of the base address of the enclave.

The ideal enclave would also have a defense-in-depth mechanism that ensured that all sections containing executable code would also be non-writable. This would protect the enclave from an attack that managed to inject code into the enclave and then execute it. However, the nature of how an enclave is loaded impacts the ability of the enclave to ensure that all code pages are non-writable. The main point is that the image of the enclave must be loaded into the EPC before it can be relocated. Since relocations need to be performed after the EPC is loaded, any code pages containing relocations have to be loaded with write permission. This opens the door for the attack mentioned above. The best option to protect against this potential attack is for the code to contain no relocations. This can be done differently depending on the format of the linked image and whether it is a 32- or 64-bit enclave. The trusted libraries that are either part of the Intel® SGX support software or provided by a 3rd party should not contain any text relocations. In addition,

the tool provided to ISVs for signing their enclaves should output a warning if an enclave image contains any relocation in the .text sections, which means the final enclave will have writable code pages.

The Intel® SGX SDK includes limited data address randomization, where the addresses of all stack variables are randomized. The randomization comes at the expense of increased stack usage. Enclaves built with the Intel® SGX SDK that support address randomization should increase their stack size setting by 4 KB. To confirm your version of the Intel® SGX SDK supports address randomization, please check the Release Notes.

Unsafe C++11 Attributes

Developers should use C++11 attributes inside an enclave with care. The attribute noreturn, in particular, may cause a potential security risk. For instance, if a trusted function calls a noreturn function any clean-up code placed after the function call will be ignored.

```
[noreturn]]void foo(parameters…)
{
    ...
}
int ecall_function(parameters…)
{
    ...
    foo(...);
    // Clean-up code below will be ignored
    ...
    return 0;
}
```

Power Management

Modern operating systems provide mechanisms for allowing applications to be notified of major power events on the platform. When the platform enters the S3 and S4 power states (suspend to RAM and hibernate to disk), the keys are erased and all of the enclaves are destroyed. Enclaves that wish to preserve secrets across S3, S4, and S5 must save state information on disk.

The Intel SGX architecture does not provide a way of directly messaging the power down event into the enclave. The application may register a callback function for such events. When the callback function is invoked then the application may call the enclave specifically to save secret state to disk for preservation. However, the operating system does not guarantee that the enclave will be given enough time to seal all its internal state. Enclaves that wish to preserve state across power transition events must periodically seal enclave state data outside the enclave (on disk or the cloud). On re-instantiation of the application, the enclave is rebuilt from scratch and the enclave must retrieve its protected state (from disk or the cloud) inside the enclave.

To minimize the overhead caused by constantly sealing secrets and storing the encrypted data on disk or the cloud, the enclave writer should design an application enclave that keeps as little state information as possible inside the enclave, so the application can survive a power transition event smoothly.

Use of Large Addresses for 32-bit Enclaves

When an enclave writer develops a 32-bit enclave, the developer must be aware that the enclave may be loaded into a large address (defined here as an address greater than 2GB) or it may receive a pointer from a large address range. The enclave should be designed to cope with these scenarios and fail smartly.

32-bit applications are usually loaded by Windows OS and run below a virtual address of 2 GB. This means that an application developer could expect that the most significant bit in a valid pointer to be zero; and therefore perform a signed operation (subtraction, comparison, etc.) on that pointer without impacting the result. If that pointer were allowed to be greater than 2 GB, then the most significant bit would become the sign bit on a signed operation and the result of the operation may change. For example, the program flow for the code below would change based on whether the enclave is loaded to an address greater than 2GB. Because ptr1 and ptr2 point inside the enclave, they would become negative numbers.

Since the enclave itself cannot control whether the system is configured to support large addresses for 32-bit programs and the enclave cannot control where it is loaded or the inputs it receives, all 32-bit enclaves should expect that they can be loaded above the 2 GB limit or receive a pointer that references memory above this limit.

```
int * ptr1, ptr2;

// Perform some operation that initializes ptr1 and ptr2
to be inside the enclave

if ( (LONG_PTR)ptr1 > (LONG_PTR)ptr2 )  //Note: LONG_PTR
is signed
{
       //do something
}
else
{
       //do something else
}
```

The developer must also be aware that pointers may be subject to integer conversion rules when used in any arithmetic or comparison operation. These rules may convert a large address into a negative number and influence the

outcome of the operation; thereby potentially impacting the integrity of the security solution.

This is similar to preparing a 32-bit application to use the /LARGEADDRESSAWARE linker option in the Microsoft linker.

Threading Topics

The developer must be aware that when using multiple threads within an enclave certain conditions related to the Thread Binding Policy or how Thread Local Storage or Mutexes are used can potentially open an enclave up to attacks.

Thread Binding Policy

When an enclave writer develops an enclave which may employ more than one thread, the developer must be aware that untrusted code controls the binding of an untrusted thread to the Trusted Thread Context (composed of a TCS page, SSA, Stack, and Thread Local Storage Variables). Thus, the developer must follow the policies on using Thread Local Storage and thread synchronization objects within the enclave.

The developer may select one of the following Thread Binding Policies for an enclave:

- Non-Binding Mode: in this mode, the untrusted runtime (uRTS) selects any available Trusted Thread Context when a root call is made into the enclave. A root call is defined as an enclave call that is not nested within another enclave call (or does not occur within the context of an enclave out call). The uRTS then uses the same Trusted Thread Context for the duration of the enclave call. In other words, it will pick the same context for a nested enclave call. As the selection of the Trusted Thread Context is arbitrary in this mode, the trusted runtime within the enclave will initialize the entire Thread Local Storage data set on each root enclave call. This means that all Thread Local Storage variables will be reset at the beginning of each root enclave call.
- Binding Mode: in this mode, the uRTS binds an untrusted thread with a Trusted Thread Context within the enclave. This means that the uRTS always selects the same Trusted Thread Context for a specific application thread. Essentially, the uRTS binds an untrusted thread and a trusted thread together. In this mode, the trusted runtime does not reinitialize the Thread Local Storage data-set on each root enclave call.

NOTE:

Binding Mode is not supported in the Universal Windows Platform (UWP) version of the uRTS.

The Thread Binding Policy is stored both inside the enclave as a trusted parameter (which is also measured) and outside the enclave as an untrusted parameter in the uRTS. The trusted parameter affects whether the trusted runtime system re-initializes Thread Local Storage variables on each root enclave call; while the untrusted parameter determines how the uRTS selects a Trusted Thread Context to use for each call.

Thread synchronization objects, mutexes in particular, cannot be used safely across root enclave calls regardless the Thread Binding Policy. Synchronization objects maintain state information, such as a mutex ownership, within the enclave. Thus a thread must not exit an enclave returning from a root enclave call after acquiring the ownership of mutex.

Scenario 1

Application thread that reenters an enclave with a different Trusted Thread Context than the one used to acquire a mutex, cannot release the mutex, because the mutex ownership is mapped to the Trusted Thread Context.

Scenario 2

Application thread that enters an enclave with the Trusted Thread Context that was previously used by another thread to acquire a mutex, does not have the ownership of the said mutex.

These scenarios may only occur when the critical section protected by the mutex or other thread synchronization object is split across root enclave calls. Therefore, avoid such practice.

Since the uRTS selects which Trusted Thread Context to use for an enclave call, and the uRTS is untrusted code, the developer must be aware that an attacker can manipulate this selection. Thus, the attacker may switch the binding mode that the uRTS employs or even select a specific Thread Context for each call.

Protection from Side-Channel Attacks

Intel SGX does not provide explicit protection from side-channel attacks. It is the enclave developer's responsibility to address side-channel attack concerns.

In general, enclave operations that require an OCall, such as thread synchronization, I/O, etc., are exposed to the untrusted domain. If using an OCall would allow an attacker to gain insight into enclave secrets, then there would be a security concern. This scenario would be classified as a side-channel attack, and it would be up to the ISV to design the enclave in a way that prevents the leaking of side-channel information.

An attacker with access to the platform can see what pages are being executed or accessed. This side-channel vulnerability can be mitigated by aligning specific code and data blocks to exist entirely within a single page.

More important, the application enclave should use an appropriate crypto implementation that is side channel attack resistant inside the enclave if side-channel attacks are a concern.

NOTE:

The Intel® Advanced Encryption Standard New Instructions (Intel® AES-NI) Set is designed to be constant time to prevent timing based side channel attacks.

Programming Recommendations

As we mentioned in the beginning, this guide is not meant to be a secure coding guideline. However, we provide some recommendations based on the lessons learned from recent security research on Intel SGX and publications review.

Uninitialized Padding

To make the access of members of a structure more efficient, the compiler may pad certain C/C++ structs. Such padding is not initialized unless the developer explicitly calls memset on the entire structure. When an ECALL/OCALL returns/passes a struct as a parameter, the enclave may leak sensitive data through the struct padding. The edge-routines copy the entire struct rather than its individual members. This means the padding values, which may contain enclave stack/heap information, will be exposed to the untrusted application. This problem is not specific to Intel SGX and has impacted the security of the Linux kernel as well.

Recommendation

Use memset to initialize all structs that return/pass information in an ECALL/OCALL. Ensuring that secrets are cleared after use also (a fundamental secure coding guideline) avoids this issue as well, unless an OCALL somehow occurs while secrets are in use. Also, it can be difficult to ensure this considering the use of third-party/open-source libraries in enclaves.