# Crypto API Toolkit for Intel® SGX

## Developer Reference

Version 1.5

September 2019

Notice: This document may contain information on products in the design phase of development. Intel may make changes to specifications and product descriptions at any time, without notice. The information here is subject to change without notice. Designers must not rely on the absence or characteristics of any features or instructions marked “reserved” or “undefined.” Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. Do not finalize a design with this information. Revised information will be published when the product is available. Verify with your local sales office that you have the latest datasheet before finalizing a design.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL’S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT, AS WELL AS ANY WARRANTY ARISING FROM COURSE OF PERFORMANCE, COURSE OF DEALING, OR USAGE IN TRADE.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL’S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.
The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel technologies’ features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure. Check with your system manufacturer or retailer or learn more at intel.com.

Intel®, Intel® Core™, the Intel logo, and Intel® Xeon™ are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

## Contents

# 1. Introduction

Crypto API Toolkit for Intel® SGX Developer Reference document is for ISVs and OEMs seeking to use the SDK to harden crypto and hash operations in their applications.

## 1.1 Terminology

| Term | Description |
|---|---|
| **SGX** | Intel® Software Guard Extensions |
| **Enclave** | A binary containing set of instructions that execute within SGX |
| **PKCS** | Public Key Cryptography Standards |
| **PKCS11** | Programming interface to create and manipulate cryptographic tokens |
| **API** | Application Programming Interface |
| **ISV** | Independent Software Vendor |
| **CSP** | Crypto Service Provider |
| **AES** | Advanced Encryption Standard |
| **AES-GCM** | AES – Galois Counter Mode |
| **AES-CTR** | AES – Counter Mode |
| **AES-CBC** | AES – Cipher Block Chaining Mode |
| **SHA256** | Secure Hashing Algorithm 256 bits |
| **HMAC-SHA256** | Hash Message Authentication Code – SHA – 256 bits |
| **SHA512** | Secure Hashing Algorithm 512 bits |
| **HMAC-SHA512** | Hash Message Authentication Code – SHA – 512 bits |
| **RSA** | Rivest–Shamir–Adleman |
| **PSS** | Probabilistic Signature Scheme |
| **Attestation** | The process of demonstrating that a software executable has been properly |
| **DCAP** | Data Center Attestation Primitives |
| **EPID** | Enhanced Privacy ID |
| **ECDSA** | Elliptic Curve Digital Signature Algorithm |
| **ECC** | Elliptic Curve Cryptography |

# 2. Technology Overview

The Crypto API Toolkit for Intel® SGX aims at enhancing the security of ISVs' and OEMs' data protection applications by exposing enhanced and optimized interfaces that run the cryptographic operations securely within Intel® SGX. The operations are exposed and supported via PKCS11 API for Linux.

## 2.1 Integration Design Principles

- All the crypto operations in Crypto API Toolkit for Intel® SGX will be exposed via a set of PKCS11 provider.
- Secret keys within the enclave will never be exported in the clear.

# 3. Architecture Components Overview

Fig. 1 provides a high-level overview of the components and their interaction in the overall solution. Please note that this document details the implementation of Crypto API Toolkit for Intel® SGX using the PKCS11 API in Linux. For any questions/clarifications related to Windows implementation, please contact your Intel representative.

## 3.1 High-level Components Overview

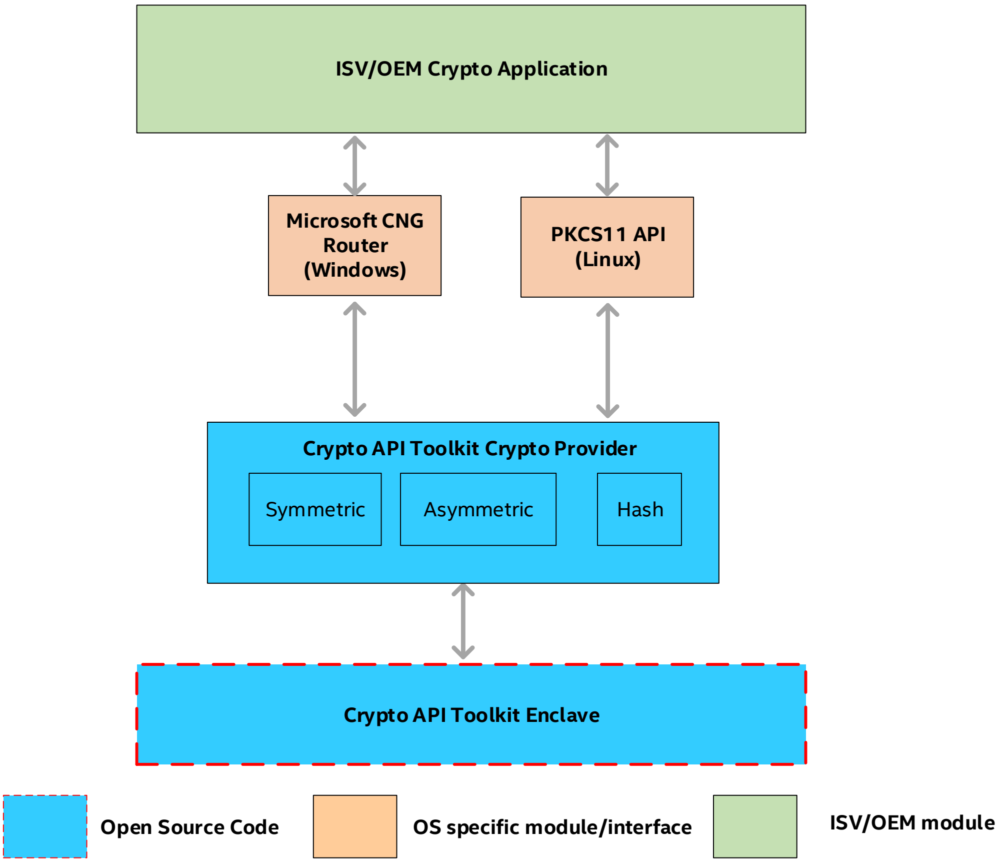
ISV/OEM Crypto Application
Microsoft CNG Router (Windows)
PKCS11 API (Linux)
Crypto API Toolkit Crypto Provider
Symmetric
Asymmetric
Hash
Crypto API Toolkit Enclave
Open Source Code
OS specific module/interface
ISV/OEM module

*Fig. 1: High Level Architecture*

- **Crypto API Toolkit Crypto Provider**: This is a PKCS11 crypto provider that exposes the symmetric, asymmetric crypto and hash functionalities of the PKCS11 specification. This component proxies calls from the application to the enclave.
- **Crypto API Toolkit Crypto Enclave**: This Crypto API Toolkit for Intel® SGX enclave implements the crypto and hash functionalities exposed by the crypto provider interface.
- The ISV/OEM components include the necessary application(s) or libraries that interact with Crypto API Toolkit for Intel® SGX through the standard PKCS11 interface.

# 4. Supported APIs and Mechanisms

Crypto API Toolkit for Intel® SGX implements the APIs according to PKCS v2.40 specification. In addition, Crypto API Toolkit for Intel® SGX also extends/adds additional mechanisms to the existing PKCS11 APIs to pass keys securely for HMAC operations. This document omits the next level details of the supported APIs, mechanisms and attributes. The developers are strongly encouraged to go through the PKCS11 specification for the details.

## 4.1 Unsupported APIs

Please refer to the PKCS v2.40 specification for details on the APIs and examples. The following is a list of PKCS11 APIs that are currently not supported by Crypto API Toolkit for Intel® SGX.

```
C_GetOperationState

C_SetOperationState

C_CopyObject

C_GetObjectSize

C_DigestKey

C_SignUpdate

C_SignFinal

C_SignRecoverInit

C_SignRecover

C_VerifyUpdate

C_VerifyFinal

C_VerifyRecoverInit

C_VerifyRecover

C_DigestEncryptUpdate

C_DecryptDigestUpdate

C_SignEncryptUpdate

C_DecryptVerifyUpdate

C_DeriveKey

C_SeedRandom

C_GenerateRandom

C_WaitForSlotEvent
```

## 4.2 Supported Algorithms

The following tables list the symmetric and asymmetric crypto algorithms, and the hash primitives supported by Crypto API Toolkit for Intel® SGX.

### 4.2.1.1 Symmetric Algorithms

| Name | Supported modes | Supported key size in bits |
|---|---|---|
| Advanced Encryption Standard (AES) | CTR, GCM, CBC | 128/192/256 |

### 4.2.1.2 Asymmetric Algorithms

| Name | Supported modes | Supported key size in bits |
|---|---|---|
| RSA | Includes RSA algorithms that use PKCS1, Optimal Asymmetric Encryption Padding (OAEP) encoding or padding, or Probabilistic Signature Scheme (PSS) plaintext padding | 1024/2048/3072/4096 |
| EC | Includes ECC curves P-256 and P-384 key pair generation and sign/verify. | 256/384 |
| ED | Includes curve ED-25519 key pair generation and sign/verify. | 256 |

### 4.2.1.3 Hashing Algorithms

| Name | Notes |
|---|---|
| Secure Hash Algorithm 256 (SHA256) | Includes HMAC-SHA256 |
| Secure Hash Algorithm 512 (SHA512) | Includes HMAC-SHA512 |

**Note:**

- Crypto API Toolkit for Intel® SGX supports a maximum of 64 keys each for symmetric and asymmetric. This can be modified in the code according to the ISV/OEM requirement provided there is enough memory for the enclave to run.
- For AES-CTR and AES-CBC modes of operation, Crypto API Toolkit for Intel® SGX restricts the support to only 16-byte IV values.
- For AES-GCM, Crypto API Toolkit for Intel® SGX restricts the tag size to 12 to 16 bytes.

## 4.3 Supported Mechanisms

As mentioned in the previous section, Crypto API Toolkit for Intel® SGX performs the supported symmetric, asymmetric and hash operations via the following mechanisms:

| Mechanism | Operation |
|---|---|
| CKM_AES_KEY_GEN | Generate an AES key |
| CKM_AES_CTR | AES crypto operations (including wrap and unwrap) with CTR |
| CKM_AES_GCM | AES crypto operations (including wrap and unwrap) with GCM |
| CKM_AES_CBC | AES crypto operations (including wrap and unwrap) with CBC |
| CKM_AES_CBC_PAD | AES crypto operations (including wrap and unwrap) with CBC padded |
| CKM_RSA_PKCS_KEY_PAIR_GEN | Generate an RSA keypair |
| CKM_RSA_PKCS | Asymmetric (RSA) crypto, wrap/unwrap & sign/verify operations with PKCS1 padding |
| CKM_RSA_PKCS_OAEP | Asymmetric (RSA) crypto, wrap/unwrap & sign/verify operations with OAEP padding |
| CKM_RSA_PKCS_PSS | Asymmetric (RSA) sign/verify operations with PSS padding with salt size of 32 bytes and SHA256 digest. |
| CKM_SHA256_RSA_PKCS | Asymmetric (RSA) signHash/verify operations with PKCS1 padding. The source buffer is hashed using SHA256 and the hash is signed. |
| CKM_SHA512_RSA_PKCS | Asymmetric (RSA) signHash/verify operations with PKCS1 padding. The source buffer is hashed using SHA512 and the hash is signed. |
| CKM_SHA256_RSA_PKCS_PSS | Asymmetric (RSA) signHash/verify operations with PSS padding with salt size of 32 bytes and SHA256 digest. The source buffer is hashed using SHA256 and the hash is signed. |
| CKM_SHA512_RSA_PKCS_PSS | Asymmetric (RSA) signHash/verify operations with PSS padding with salt size of 32 bytes and SHA256 digest. The source buffer is hashed using SHA512 and the hash is signed. |
| CKM_SHA256 | Hashing with SHA256 |
| CKM_SHA512 | Hashing with SHA512 |
| CKM_SHA256_HMAC_AES_KEYID | HMAC with SHA256 |
| CKM_SHA512_HMAC_AES_KEYID | HMAC with SHA512 |
| CKM_EXPORT_RSA_PUBLIC_KEY | Exporting an asymmetric public key |
| CKM_IMPORT_RSA_PUBLIC_KEY | Importing an asymmetric public key |
| CKM_EXPORT_EPID_QUOTE_RSA_PUBLIC_KEY | Exporting an asymmetric public key along with public key hash and quote using EPID attestation primitives |
| CKM_EXPORT_ECDSA_QUOTE_RSA_PUBLIC_KEY | Exporting an asymmetric public key along with public key hash and quote using DCAP (ECDSA attestation primitives) |
| CKM_EC_KEY_PAIR_GEN | Generate an EC key pair – EC p256 & p384 curves |
| CKM_EC_EDWARDS_KEY_PAIR_GEN | Generate an ED key pair – ED25519 curve |
| CKM_ECDSA | Sign/Verify using an EC key pair |

| | |
|---|---|
| CKM_EDDSA | Sign/Verify using an ED key pair |

The custom mechanisms that are implemented by Crypto API Toolkit for Intel® SGX are described below:

### 4.3.1 CKM_SHA256_HMAC_AES_KEYID

Crypto API Toolkit for Intel® SGX exposes the mechanism CKM_SHA256_HMAC_AES_KEYID for SHA256 HMAC operations. With this mechanism, the application can specify the key handle that it wants to use for doing the HMAC operation without the need to pass the key in the clear. The details of the mechanism are explained in the section Message digesting and HMAC Operations

### 4.3.2 CKM_SHA512_HMAC_AES_KEYID

Crypto API Toolkit for Intel® SGX exposes the mechanism CKM_SHA512_HMAC_AES_KEYID for SHA512 HMAC operations. With this mechanism, the application can specify the key handle that it wants to use for doing the HMAC operation without the need to pass the key in the clear. The details of the mechanism are explained in the section Message digesting and HMAC Operations

### 4.3.3 CKM_EXPORT_EPID_QUOTE_RSA_PUBLIC_KEY

Crypto API Toolkit for Intel® SGX can export the public key along with hash and a quote based on that using the mechanism 4.3.5 CKM_EXPORT_EPID_QUOTE_RSA_PUBLIC_KEY. The details are explained in the section Export the public portion of an asymmetric key along with a quote and hash using EPID attestation primitives

### 4.3.4 CKM_EXPORT_ECDSA_QUOTE_RSA_PUBLIC_KEY

For Intel® platforms that are targeted for the Data Centers, Crypto API Toolkit for Intel® SGX supports exporting the public key along with hash and a quote based on that using the mechanism CKM_EXPORT_ECDSA_QUOTE_RSA_PUBLIC_KEY using the SGX Data Center Attestation Primitives (DCAP). The details are explained in the section Export the public portion of an asymmetric key along with a quote and hash with Data Center Attestation Primitives (DCAP)

## 4.4 Supported Attributes

Crypto API Toolkit for Intel® SGX supports the following attributes:

| | |
|---|---|
| CKA_CLASS | CKA_KEY_TYPE |
| CKA_PRIVATE | CKA_SIGN |
| CKA_TOKEN | CKA_VERIFY |
| CKA_ENCRYPT | CKA_MODIFIABLE |
| CKA_DECRYPT | CKA_COPYABLE |
| CKA_WRAP | CKA_DERIVE |
| CKA_UNWRAP | CKA_LOCAL |

| | |
|---|---|
| CKA_MODULUS_BITS | CKA_LABEL |
| CKA_VALUE_LEN | CKA_VALUE_KEY_BUFFER |
| CKA_ID | CKA_LABEL |
| CKA_KEY_GEN_MECHANISM | CKA_PUBLIC_EXPONENT |
| CKA_MODULUS | CKA_EC_PARAMS |

## 4.4.1 Mandatory attributes to be set for key generation

| **Attribute** | **Key** |
|---|---|
| CKA_CLASS | AES, RSA, EC, ED |
| CKA_KEY_TYPE | AES, RSA, EC, ED |
| CKA_MODULUS_BITS | RSA (public key) |
| CKA_VALUE_LEN | AES |
| CKA_VALUE_KEY_BUFFER | AES (raw key import) |
| CKA_EC_PARAMS | EC/ED |

## 4.4.2 Limitations and Restrictions

| **Attribute** | **Information** |
|---|---|
| CKA_DERIVE | Since Crypto API toolkit does not support key derivation, CKA_DERIVE cannot be set to CK_TRUE. Default value is CK_FALSE. |
| CKA_COPYABLE | Since Crypto API toolkit does not support copying objects, CKA_COPYABLE cannot be set to CK_TRUE. Default value is CK_FALSE. |
| CKA_PRIVATE | Crypto API toolkit does not support changing CKA_PRIVATE attribute via C_SetAttributeValue. |
| CKA_TOKEN | Crypto API toolkit does not support changing CKA_TOKEN attribute via C_SetAttributeValue. |

# 5. Flows

Crypto API Toolkit for Intel® SGX provider layer depends on the SGX enclave for performing key generation, crypto and hash operations. The enclave module depends on Intel® Software Guard Extensions SSL library (a.k.a. SGX-SSL - OpenSSL built for SGX) for crypto functionalities (except AES-CTR) and Intel® IPP Cryptography library for hash/digest/HMAC and AES-CTR operations.

## 5.1 Typical Flow

In a typical flow, the application will,

1. Load the Crypto API Toolkit for Intel® SGX dynamic library (or link to the dynamic library and can optionally perform steps 2 & 3).
2. Get the pointer to the function list via C_GetFunctionList.
3. Load the function list.
4. Call C_Finalize to complete any previous sessions, if required.
5. Call C_Initialize to initialize the PKCS11 library.
6. The application can enumerate the slots and get info about the token using C_GetSlotList and C_GetTokenInfo.
7. If the token has not been initialized, the application will initialize the token using C_InitToken and get info of the slot using C_GetSlotInfo.
8. The application can get the list of mechanisms supported and the information of those mechanisms by using C_GetMechanismList and C_GetMechanismInfo
9. Open session(s) to interact with the token. The application might need to initialize PIN, set PIN and login based on the operations to be performed.
10. Perform crypto including key[pair] generation, encrypt/decrypt, wrap/unwrap keys, sign/verify and hash/digest/HMAC operations.
11. Perform cleanup operations (destroying or wrapping the keys for usage later, freeing up memory etc.,)
12. Close the session(s).
13. Finalize the provider.
14. Unload the library.

Please refer to the PKCS v2.40 specification for details on the APIs and examples for the various operations. The below sections explain the operations with custom/additional mechanisms that Crypto API Toolkit for Intel® SGX implements.

## 5.2 Wrapping and Unwrapping Keys

The keys created inside the Crypto API Toolkit for Intel® SGX enclave can be wrapped and persisted. For the purpose of this discussion, the terms ***wrap and export*** will be used interchangeably as well as the terms ***unwrap and import***. The wrapped keys can be used in the same session or later (if persisted) by importing the exported key blob. Crypto API Toolkit for Intel® SGX does not export the keys without wrapping them or binding them to the platform (no secret key export). PKCS11 specification talks in detail about wrapping and unwrapping the keys. Crypto API Toolkit for Intel® SGX assures that the keys can get unwrapped only inside the enclave and does not come out in the clear.

**IMPORTANT**

The following restrictions are imposed by Crypto API Toolkit for Intel® SGX for the wrapping and wrapped keys.

1. A wrapped key can get unwrapped only inside the enclave. It cannot be extracted to come out in the clear.
2. A key used for wrapping cannot be used for encryption or decryption.
3. If the key that is used for wrapping another key, was used for encrypting data earlier, that encrypted data cannot be decrypted post the wrapping operation.

## 5.2.1 Wrapping and unwrapping a symmetric key

An application can do a custom wrapping and export the key (for example, the application might want to store the symmetric key by wrapping it with a symmetric key or an asymmetric key).

### 5.2.1.1 Wrapping a symmetric key with symmetric or asymmetric key

After performing the initialization (load library, initialize and init token) and after creating the keys (the key used for wrapping and the key to be wrapped), the application,

1. Sets the mechanism parameters based on the symmetric mode to be used (CTR, GCM or CBC) or asymmetric key. Below are the mechanisms and the parameters.

| | |
|---|---|
| AES-CTR | `typedef struct CK_AES_CTR_PARAMS {`<br>`    CK_ULONG ulCounterBits;`<br>`    CK_BYTE cb[16];`<br>`} CK_AES_CTR_PARAMS;` |
| AES-GCM | `typedef struct CK_GCM_PARAMS {`<br>`    CK_BYTE_PTR       pIv;`<br>`    CK_ULONG          ulIvLen;`<br>`    CK_ULONG          ulIvBits;`<br>`    CK_BYTE_PTR       pAAD;`<br>`    CK_ULONG          ulAADLen;`<br>`    CK_ULONG          ulTagBits;`<br>`} CK_GCM_PARAMS;` |

2. The application fills the structure and passes the address of the structure to the pParameter of `CK_MECHANISM`. It also fills the `ulParameterLen` to the size of the structure (For AES-CBC, it is only the IV which is directly filled as byte array and the size of the array is passed for ulParameterLen, and for wrapping using an asymmetric key, there are no parameters)
3. The application calls `C_WrapKey` with the mechanism, the object handles of the key to be wrapped and key to be used for wrapping and passes a `NULL_PTR` for the destination buffer.
4. If the key handles are valid, the enclave returns the size of the destination buffer required to hold the wrapped key.
5. If the call is successful, the application allocates the buffer as per the returned buffer size.

6. The application calls C_WrapKey with the same parameters, but with the allocated buffer for pWrappedKey parameter and the allocated size in pulWrappedKeyLen.
7. Crypto API Toolkit for Intel® SGX wraps the key and returns it in the pWrappedKey buffer.

#### 5.2.1.2 Unwrapping a symmetric key with symmetric or asymmetric key

After performing the initialization (load library, initialize and init token) and after creating the keys (the key used for wrapping and the key to be wrapped), the application

1. Sets the mechanism parameters based on the symmetric mode to be used (CTR, GCM or CBC) or asymmetric key. The mechanisms and the parameters are as listed in 5.2.2.1.
2. The application fills the structure and passes the address of the structure to the pParameter of CK_MECHANISM. It also fills the ulParameterLen to the size of the structure (For AES-CBC, it is only the IV which is directly filled as byte array and the size of the array is passed for ulParameterLen, and for wrapping using an asymmetric key, there are no parameters)
3. The application calls C_UnwrapKey with the mechanism, the object handle of key to be used for wrapping, the wrapped key buffer and its length along with the key attributes to be set (in the pTemplate parameter) on the key being unwrapped.
4. If the key handle is valid and unwrapping is successful, the enclave returns the unwrapped key handle in the phKey parameter. The key is unwrapped (imported) inside the Crypto API Toolkit for Intel® SGX enclave and never is visible outside of it.
5. The application can then start using this key handle for the purposes set in the template that is sent during the call to unwrap.

### 5.2.2 Wrapping and unwrapping an asymmetric key

For asymmetric keys, both public and private keys can be exported (wrapped) as platform bind as explained in the section above. The application might want to export the public key portion only (for example, verification of signature). The application might want to export the public key along with a quote from the enclave to verify the trust worthiness of the enclave as well as to provision a secret. Both these operations are explained below:

#### 5.2.2.1 Export the public portion of an asymmetric key

After performing the initialization (load library, initialize and opening the session) and after creating the key pair (whose public key is to be exported), the application

1. Sets the mechanism to CKM_EXPORT_RSA_PUBLIC_KEY.
2. The application calls C_WrapKey with the mechanism, the object handle of the key whose public portion is to be exported and passes a NULL_PTR for the hWrappingKey and the destination buffer.
3. If the key handle is valid, the enclave returns the size of the destination buffer required to hold the exported public key which is the total size of the actual public key buffer (modulus and exponent) and sizeof(CK_RSA_PUBLIC_KEY_PARAMS).
4. If the call is successful, the application allocates the buffer as per the returned buffer size.
5. The application calls C_WrapKey with the same parameters, with the allocated buffer for pWrappedKey parameter and the allocated size in pulWrappedKeyLen.

6. The Crypto API Toolkit for Intel® SGX enclave exports the public key and returns it in the pWrappedKey buffer (below format).

| | | | | |
|---|---|---|---|---|
| Size | CK_RSA_PUBLIC_KEY_PARAMS | | ulModulusLen | ulExponentLen |
| Data | ulExponentLen | ulModulusLen | modulus | exponent |

### 5.2.2.2 Export the public portion of an asymmetric key along with a quote and hash using EPID attestation primitives

After performing the initialization (load library, initialize and opening the session) and after creating the key pair (whose public key is to be exported along with the enclave quote), the application

1. Sets the mechanism to CKM_EXPORT_EPID_QUOTE_RSA_PUBLIC_KEY and fills the mechanism parameters in the structure CK_EPID_QUOTE_RSA_PUBLIC_KEY_PARAMS.

```
typedef struct CK_EPID_QUOTE_RSA_PUBLIC_KEY_PARAMS {
    CK_BYTE_PTR       pSpid;
    CK_ULONG          ulSpidLen;
    CK_BYTE_PTR       pSigRL;
    CK_ULONG          ulSigRLLen;
    CK_ULONG          ulQuoteSignatureType;
} CK_EPID_QUOTE_RSA_PUBLIC_KEY_PARAMS;
```

2. The application calls C_WrapKey with the mechanism details, the object handle of the key to be exported and passes a NULL_PTR for the hWrappingKey and destination buffer.

3. If the key handle is valid, the enclave returns the size of the destination buffer required to hold the exported public key and quote which is the total size of the actual public key buffer and sizeof(CK_RSA_PUBLIC_KEY_PARAMS) and the size of the quote itself. The quote is of the format sgx_quote_t and contains the hash of the public key and the quote generated based on this.

4. If the call is successful, the application allocates the buffer as per the returned buffer size.

5. The application calls C_WrapKey with the same parameters, with the allocated buffer for pWrappedKey parameter and the allocated size in pulWrappedKeyLen.

6. The Crypto API Toolkit for Intel® SGX enclave calculates the hash of the public key, generates a quote based on this as well as the other parameters passed (SPID, SIGRL etc.,) and returns it in the pWrappedKey buffer (in the below format).

| | | | | | | |
|---|---|---|---|---|---|---|
| Size | CK_RSA_PUBLIC_KEY_PARAMS | | ulModulusLen | ulExponentLen | sgx_quote_t | |
| Data | ulExponentLen | ulModulusLen | modulus | exponent | SHA256 hash of the public key | quote |

The following example shows how the quote from the enclave based on the public key's hash can be retrieved.

```
std::vector<CK_BYTE> customQuote(CK_MECHANISM_TYPE     mechanismType,
                                 CK_SESSION_HANDLE     hSession,
                                 CK_OBJECT_HANDLE      hKey)
{
    CK_RV                   p11Status          = CKR_GENERAL_ERROR;
    CK_MECHANISM            mechanism          = { mechanismType, NULL_PTR, 0 };
    CK_ULONG                quotePublicKeyLen  = 0UL;
    CK_ULONG                signatureType      = UNLINKABLE_SIGNATURE;
    bool                    result             = false;
    std::vector<CK_BYTE>    spid { 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, \
                                   0x99, 0x99, 0x88, 0x77, 0x66, 0x55, 0x44, 0x33 };
    std::vector<CK_BYTE>    sigRL;
    std::vector<CK_BYTE>    quotePublicKey;
    CK_MECHANISM_PTR        pMechanism((CK_MECHANISM_PTR)&mechanism);

    CK_EPID_QUOTE_RSA_PUBLIC_KEY_PARAMS quoteRSAParams =
    {
        spid.data(),
        spid.size(),
        sigRL.data(),
        sigRL.size(),
        signatureType
    };

    do
    {
        pMechanism->pParameter = &quoteRSAParams;
        pMechanism->ulParameterLen = sizeof(quoteRSAParams);

        p11Status = p11->C_WrapKey(hSession,
                                   pMechanism,
                                   NULL_PTR,
                                   hKey,
                                   NULL_PTR,
                                   &quotePublicKeyLen);
        if (CKR_OK != p11Status)
        {
            break;
        }

        quotePublicKey.resize(quotePublicKeyLen);
        p11Status = p11->C_WrapKey(hSession,
                                   pMechanism,
                                   NULL_PTR,
                                   hKey,
                                   quotePublicKey.data(),
                                   &quotePublicKeyLen);
        if (CKR_OK != p11Status)
        {
            break;
        }

        result = true;
    } while (false);

    if (!result)
    {
       quotePublicKey.clear();
    }

    return quotePublicKey;
}
```

The quote portion of this blob can be verified with the Intel Attestation Server using remote attestation. Please refer to https://github.com/intel/sgx-ra-sample for how to verify the quote. Please also refer to https://software.intel.com/en-us/articles/code-sample-intel-software-guard-extensions-remote-attestation-end-to-end-example for more details.

Once the quote verification is successful, the verification module can compute the SHA256 hash of the public key (present in the blob as shown above) and compare against the SHA256 hash in the blob. This public key may be used to provision a secret key into the enclave.

#### 5.2.2.3 Export the public portion of an asymmetric key along with a quote and hash with Data Center Attestation Primitives (DCAP)

**IMPORTANT**
This capability is supported only in the Intel® platforms that support Intel® SGX Data Center Attestation Primitives. For more details on the platforms that support this as well as how to use this feature (which includes quote provider & quote verification), please refer to https://software.intel.com/en-us/blogs/2018/12/09/an-update-on-3rd-party-attestation.

After performing the initialization (load library, initialize and opening the session) and after creating the key pair (whose public key is to be exported along with the enclave quote), the application

1. Sets the mechanism to CKM_EXPORT_ECDSA_QUOTE_RSA_PUBLIC_KEY and fills the mechanism parameters in the structure CK_ECDSA_QUOTE_RSA_PUBLIC_KEY_PARAMS.

```
typedef struct CK_ECDSA_QUOTE_RSA_PUBLIC_KEY_PARAMS {
    CK_LONG qlPolicy;
} CK_ECDSA_QUOTE_RSA_PUBLIC_KEY_PARAMS;
```

2. The application calls C_WrapKey with the mechanism details, the object handle of the key to be exported and passes a NULL_PTR for the hWrappingKey and destination buffer.
3. If the key handle is valid, the enclave returns the size of the destination buffer required to hold the exported public key and quote which is the total size of the actual public key buffer and sizeof(CK_RSA_PUBLIC_KEY_PARAMS) and the size of the quote itself. The quote is of the format sgx_quote_t and contains the hash of the public key and the quote generated based on this.
4. If the call is successful, the application allocates the buffer as per the returned buffer size.
5. The application calls C_WrapKey with the same parameters, with the allocated buffer for pWrappedKey parameter and the allocated size in pulWrappedKeyLen.
6. The Crypto API Toolkit for Intel® SGX enclave calculates the hash of the public key, generates a quote and returns it in the pWrappedKey buffer (in the below format).

| Size | CK_RSA_PUBLIC_KEY_PARAMS | | ulModulusLen | ulExponentLen | sgx_quote_t | |
|---|---|---|---|---|---|---|
| Data | ulExponentLen | ulModulusLen | modulus | exponent | SHA256 hash of the public key | quote |

The following example shows how the quote from the enclave based on the public key's hash can be retrieved.

```
std::vector<CK_BYTE> customQuote(CK_MECHANISM_TYPE     mechanismType,
                                 CK_SESSION_HANDLE     hSession,
                                 CK_OBJECT_HANDLE      hKey)
{
    CK_RV                   p11Status          = CKR_GENERAL_ERROR;
    CK_MECHANISM            mechanism          = { mechanismType, NULL_PTR, 0 };
    CK_ULONG                quotePublicKeyLen  = 0UL;
    CK_LONG                 qlPolicy           = SGX_QL_PERSISTENT;
    bool                    result             = false;
    std::vector<CK_BYTE>    quotePublicKey;
    CK_MECHANISM_PTR        pMechanism((CK_MECHANISM_PTR)&mechanism);

    CK_ECDSA_QUOTE_RSA_PUBLIC_KEY_PARAMS quoteRSAParams =
    {
        qlPolicy
    };

    do
    {
        pMechanism->pParameter = &quoteRSAParams;
        pMechanism->ulParameterLen = sizeof(quoteRSAParams);

        p11Status = p11->C_WrapKey(hSession,
                                   pMechanism,
                                   NULL_PTR,
                                   hKey,
                                   NULL_PTR,
                                   &quotePublicKeyLen);
        if (CKR_OK != p11Status)
        {
            break;
        }

        quotePublicKey.resize(quotePublicKeyLen);
        p11Status = p11->C_WrapKey(hSession,
                                   pMechanism,
                                   NULL_PTR,
                                   hKey,
                                   quotePublicKey.data(),
                                   &quotePublicKeyLen);
        if (CKR_OK != p11Status)
        {
            break;
        }

        result = true;
    } while (false);

    if (!result)
    {
       quotePublicKey.clear();
    }

    return quotePublicKey;
}
```

This public key, after verification, may be used to provision a secret key into the enclave.

**NOTE**
If the the enclave has only the public key available (imported) and the private key is not available as part of the key pair, the quote generation will fail.

#### 5.2.2.4 Import the public portion of an asymmetric key

The application after creating the RSA key pair (whose public key is to be exported), can perform the steps below to export the public portion of the RSA key that it has created.

1. Sets the mechanism to `CKM_IMPORT_RSA_PUBLIC_KEY`.

2. The application calls `C_UnwrapKey` with the mechanism, the public key buffer (in the format as mentioned above) and its length along with the key attributes to be set (in the `pTemplate` parameter) on the key being imported. The key handle to be used for unwrapping is set to `NULL_PTR`.
3. If the import is successful, the enclave returns the key handle for the imported public key in the phKey parameter.
4. The application can then start using this key handle for the purposes set in the template that is sent during the call to unwrap.

### 5.2.2.5 Import the private portion of an asymmetric key

Crypto API Toolkit provides a way for an application to import a private portion of an RSA key into the enclave securely. For example, a private RSA key created using openssl in a secure environment (for example, a key server) can be imported into Crypto API Toolkit. This private key, for security reasons, cannot be imported directly, but instead it (in PKCS8 format) can be wrapped with a symmetric key and this wrapped blob can be unwrapped inside Crypto API Toolkit enclave. To achieve this, the application can perform the below steps.

0. Application has the wrapped key blob (RSA private key wrapped with an AES key).
1. Set the mechanism to one of CKM_AES_CTR, CKM_AES_GCM, CKM_AES_CBC or CKM_AES_CTR_PAD. This should match the wrapping mechanism used to wrap the private key.
2. Call `C_UnwrapKey` with the above mechanism, the wrapped private key buffer and its length along with the key attributes to be set (in the `pTemplate` parameter) on the key being imported. The key handle to be used for unwrapping should be the one that's used to wrap the private key.
3. If the unwrap and import is successful, the enclave returns the key handle for the imported private key in the phKey parameter.
4. The application can now use phKey obtained from the above C_UnwrapKey call.

```
// Sample code that generates an RSA key using openssl and
// imports the private portion of it into Crypto API Toolkit
// via C_UnwrapKey call

#if OPENSSL_VERSION_NUMBER < 0x10100000L
    OpenSSL_add_all_algorithms();
    ERR_load_crypto_strings();
#endif
    ERR_load_BIO_strings();
    unsigned long e = RSA_F4;
    BIGNUM* bne     = BN_new();
    if (BN_set_word(bne,e) != 1)
    {
        std::cout << "BN_set_word Failed" << std::endl;
        exit(0);
    }
    RSA* rsa = RSA_new();
    if (RSA_generate_key_ex(rsa, 2048, bne, NULL) != 1)
    {
        std::cout << "RSA_generate_key_ex Failed" << std::endl;
        exit(0);
    }
    EVP_PKEY* pkey = EVP_PKEY_new();
    PKCS8_PRIV_KEY_INFO * p8inf = nullptr;
    if (EVP_PKEY_set1_RSA(pkey, rsa))
    {
        p8inf = EVP_PKEY2PKCS8(pkey);
    }
    p8inf = EVP_PKEY2PKCS8(pkey);
    if (!p8inf)
    {
        std::cout << "EVP_PKEY2PKCS8 Failed" << std::endl;
        exit(0);
    }
    int len = 0;
    if ((len = i2d_PKCS8_PRIV_KEY_INFO(p8inf, NULL)) < 0)
    {
        std::cout << "i2d_PKCS8_PRIV_KEY_INFO Failed" << std::endl;
        exit(0);
    }
    unsigned char *privateKey = new unsigned char[len];
    int copiedBytes = i2d_PKCS8_PRIV_KEY_INFO(p8inf, &privateKey);
    privateKey -= copiedBytes;

    // wrapPrivateKeyWithAESKey: Function that encrypts privatekey with the key pointed by hAESKeyHandle.
    std::vector<CK_BYTE> wrappedKey = wrapPrivateKeyWithAESKey(privateKey, hAESKeyHandle);
    // wrappedKey is the blob obtained by wrapping private key with AES key.

    CK_OBJECT_HANDLE importedKey =  CK_INVALID_HANDLE;
    CK_KEY_TYPE     rsaKeyType   = CKK_RSA;
    CK_OBJECT_CLASS keyClass     = CKO_PRIVATE_KEY;
    CK_UTF8CHAR     keyLabel[]   = "RSA key unWrapped with AES Key";
    CK_UTF8CHAR     id[]         = "1";
    CK_BBOOL        bTrue        = CK_TRUE;

    CK_ATTRIBUTE asymPrivateKeyAttribs[] = {{ CKA_DECRYPT,  &bTrue,      sizeof(bTrue) },
                                            { CKA_SIGN,     &bTrue,      sizeof(bTrue) },
                                            { CKA_UNWRAP,   &bTrue,      sizeof(bTrue) },
                                            { CKA_KEY_TYPE, &rsaKeyType, sizeof(rsaKeyType) },
                                            { CKA_CLASS,    & keyClass,  sizeof(keyClass)   },
                                            { CKA_LABEL,    keyLabel,    sizeof(keyLabel)-1 },
                                            { CKA_ID,       &id[0],      sizeof(id)         }
                                            };
    p11Status = p11->C_UnwrapKey(hSession, pMechanism, hAESKeyHandle,
                                 wrappedKey.data(),wrappedKey.size(),
                                 asymPrivateKeyAttribs,
                                 sizeof(asymPrivateKeyAttribs) / sizeof(CK_ATTRIBUTE),
                                 &importedKey);
    if (CKR_OK != p11Status)
    {
        exit(0);
    }
    // importedKey is the key handle to RSA private key imported inside enclave.
```

## 5.3 Message digesting and HMAC Operations

Crypto API Toolkit for Intel® SGX supports hashing/digesting of the message. The detailed flow to generate a message digest are explained in detail with example in PKCS11 v2.40 specification. In this section we will explain how to generate a digest with HMAC using the secret key (AES) already generated or present in the enclave.

After performing the initialization, the application generates or imports the symmetric key (AES) to be used for HMAC operation. The application then,

1. Sets the mechanism to CKM_SHA256_HMAC_AES_KEYID or CKM_SHA512_HMAC_AES_KEYID, based on the requirement.
2. Sets the ulKeyID parameter of the CK_HMAC_AES_KEYID_PARAMS to the AES key to be used for HMAC operation. This ulKeyID is the AES key handle that has been generated or imported earlier. The pMechanism's pParameter is set to point to this structure and its ulParameterLen is set to sizeof(CK_HMAC_AES_KEYID_PARAMS).
3. Calls C_DigestInit with this pMechanism followed by C_DigestUpdate.
4. Makes a call to C_DigestFinal with NULL_PTR for pDigest. The enclave will calculate the size of the buffer to hold the digest and return it via pulDigestLen.
5. The application allocates the memory for the buffer and calls C_DigestFinal with this buffer to get the message digest.

## 5.4 Asymmetric Sign and Verify

Crypto API Toolkit for Intel®SGX supports signing and verify data using an asymmetric key pair.
The flows supported are C_SignInit → C_Sign and C_VerifyInit → C_Verify. C_SignFinal and C_VerifyFinal are not supported.

The following mechanisms are supported for signing and verifying data.

| **Mechanism** | **Description** |
|---|---|
| CKM_RSA_PKCS | Signs a data buffer. |
| CKM_RSA_PKCS_PSS | Signs a data buffer. |
| CKM_SHA256_RSA_PKCS | Hashes the data buffer and then signs. Uses SHA256 for hash computation. |
| CKM_SHA512_RSA_PKCS | Hashes the data buffer and then signs. Uses SHA512 for hash computation. |
| CKM_SHA256_RSA_PKCS_PSS | Hashes the data buffer and then signs. Uses SHA256 for hash computation. PSS padding scheme uses SHA256 digest. |
| CKM_SHA512_RSA_PKCS_PSS | Hashes the data buffer and then signs. Uses SHA512 for hash computation. Please note that the PSS padding scheme supports only SHA256 digest. |
| CKM_ECDSA | Signs/verifies a data buffer with an EC key. |
| CKM_EDDSA | Signs/verifies a data buffer with an ED key. |

## 5.5 Power Transition Scenarios

Crypto API Toolkit for Intel® SGX does not explicitly save/restore keys, handles, or crypto contexts during power transition scenarios like sleep, hibernate and resume. It is the application’s responsibility to intercept the power events and handle them by closing the key(s) and session(s) and re-establishing the session on a resume. The application can make use of token objects to have the keys persistent through power transition scenarios.

# 6. Building the source

The source has been built and tested with CentOS 7.5, Ubuntu Desktop v16.04 and v18.04.

## 6.1 Pre-requisites

Crypto API Toolkit for Intel® SGX depends on

### 6.1.1 autotools (autoconf, automake, libtool)

On CentOS, autotools can be installed with the command

```
sudo yum install autotools-latest
```

On Ubuntu, autotools can be installed with the command

```
sudo apt-get install autoconf automake libtool
```

### 6.1.2 g++ compiler with C++11 support

### 6.1.3 Intel® SGX SDK for Linux.

- The latest Intel(R) SGX SDK can be installed from https://01.org/intel-software-guard-extensions/downloads for the operating systems supported.
- Install the driver, libsgx-enclave-common and SDK in that order.
- Please refer to Intel® Software Guard Extensions (Intel® SGX) SDK for Linux* OS Installation Guide for more details.
- For exporting the public key along with a quote based on its hash using DCAP, the SGX DCAP can be downloaded and installed from https://01.org/intel-softwareguard-extensions/downloads/intel-sgx-dcap-linux-1.1-release or https://github.com/intel/SGXDataCenterAttestationPrimitives.

### 6.1.4 Intel® SGX SSL add-on

- Please download the Intel(R) SGX SSL from https://github.com/intel/intel-sgx-ssl, follow the instructions there to install it. Please note that the current release of Crypto API Toolkit for Intel(R) SGX has been validated with Intel(R) SGX SSL that uses OpenSSL version 1.1.1c.
- The build scripts have been configured to build with the release versions of SGX SSL libraries.

## 6.2 Enclave configuration

### 6.2.1 HeapMaxSize

The enclave configuration XML is configured for 80MB as `HeapMaxSize`. The ISVs/OEMs should review this configuration file setting to make sure it satisfies their requeirements. For example, AES-GCM multipart decrypt requires the enclave to store the decrypted buffer internally until C_DecryptFinal is called. When the multipart decrypted buffer grows beyond the available size (`HeapMaxSize`), the enclave will fail (crash).

This maximum heap size includes the tag outputted in case of AES-GCM and the padding in case of AES-CBC.

### 6.2.2 Multithreading support

Support for multithreading will be enabled in a future release.

### 6.2.3 Build configuration

The enclave is configured to have **`DisableDebug`** set to 0 for the purpose of debug during development and integration. This means that enclave will be debuggable. The ISVs/OEMs must configure this tag `<DisableDebug>` to 1 for a production enclave. Please refer to the section **Enclave Project Configurations** in the Intel® Software Guard Extensions SDK for Linux* OS – Developer Reference for more information. Please also note that the provider that loads the enclave needs to be built with NDEBUG preprocessor macro that disables the SGX_DEBUG_FLAG (will be defined as 0).

**NOTE** The release binaries thus built will need to be signed with a whitelisted key for the enclave to load and function properly. The details for signing and whitelisting the enclave is presented in section 6.4 Signing and Whitelisting the Enclave

## 6.3 Build Steps

The enclave, the provider and the sample application are configured to be built with autotools.

The sequence of steps to build are below:

### 6.3.1 Configure

```
sh autogen.sh

./configure
```

The build can be configured using the following additional options.

| Option | Description |
|---|---|
| `--prefix=`*`/path-to-install/`* | Sets the installation directory. Defaults to */usr/local/* if this option is not specified. |
| `--with-toolkit-path=`*`/path-for-tokens/`* | Sets the location for tokens to be created. Defaults to */opt/intel/cryptoapitoolkit/* if this option is not specified. |
| `--with-sgxsdk=/path-to-sgxsdk-installation/` | The SGX SDK installation directory. Defaults to */opt/intel/sgxsdk/* if this option is not specified. |
| `--with-sgxssl=/path-to-sgxssl-installation/` | The SGX SSL installation directory. Defaults to */opt/intel/sgxssl/* if this option is not specified. |

| | |
|---|---|
| `--with-dcap-path=/path-to-dcap-include-path/` | The DCAP include directory. If this option is not provided while configuring, features depending on DCAP are not supported. |
| `--enable-import-raw-symkey-support= "yes"` | Setting this option to 'yes' allows importing a data buffer (raw key import) into Crypto API Toolkit. |
| `--enable-import-raw-symkey-support-for-hmac="yes"` | Setting this option to 'yes' allows the use of raw key imported as the secret buffer for HMAC operations. |
| `--enable-verify-with-ec-support` | Setting this option to 'yes' allows verification of signatures using EC keys. |
| `--enable-verify-with-ed-support` | Setting this option to 'yes' allows verification of signatures using ED keys. |

### 6.3.2 Build and Install

To build the Crypto API Toolkit for Intel® SGX, please run

```
sudo make
```

To install Crypto API Toolkit for Intel® SGX, please run

```
sudo make install
```

**IMPORTANT**

- ISVs/OEMs need to carefully evaluate their requirements, mainly from a security standpoint, and decide whether they require the support for importing raw key using `--enable-import-raw-symkey-support`
- Please note that the enclave is signed with a test signing key. A production enclave should go through the process of signing and whitelisting an enclave as explained in the next section.

## 6.4 Signing and Whitelisting the Enclave

For deploying the enclave for production, the unsigned enclave SO needs to go through production signing and whitelisting process. The process of signing the enclave is detailed in Intel SGX Developer Reference for Linux. The process of whitelisting and onboarding is detailed in https://software.intel.com/en-us/articles/intel-software-guard-extensions-product-licensing-faq.

Once the enclave is signed and whitelisted, Intel Developer Services will send/notify the submitted about the completion of whitelisting process and the updated whitelisted enclave bin (for Linux) can be downloaded from http://whitelist.trustedservices.intel.com/SGX/LCWL/Linux/sgx_white_list_cert.bin.

Once the enclave is whitelisted and the updated whitelist cert bin is available in the link above, the AESM service, when it starts and connected to the network will fetch it, after a reboot. The enclave load call (`sgx_create_enclave`) for the production signed enclave will fail with `sgx_status_t::SGX_ERROR_SERVICE_INVALID_PRIVILEGE`, if the AESM has not downloaded the updated whitelist cert binary. This can be handled by building the provider with the whitelist cert bin and manually registering the cert chain by using the SGX API `sgx_register_wl_cert_chain`.