



UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Strumenti per il confidential computing

Sistemi Embedded

Anno Accademico 2023/2024

Giuseppe Capasso matr. M63001498

Indice

1 Confidential computing	2
1.1 Trusted Execution Environment (TEE)	2
1.2 Intel Software Guards Extentions (SGX)	2
1.2.1 Enclave	3
1.2.2 Attestazione	4
1.2.3 SGX2	4
2 Strumenti	6
2.1 Classificazione	6
2.2 Gramine	6
2.2.1 Struttura di un'applicazione con Gramine	7
2.2.2 Remote attestation	8
2.3 OpenEnclave	10
2.3.1 Struttura di un'applicazione con Open Enclave	10
2.3.2 Attestazione	10
3 Sviluppo di un'applicazione sicura	11
3.1 Setup di laboratorio	11
3.1.1 Installazione Gramine	11
3.1.2 Installazione Open Enclave	13
3.1.3 Configurazione infrastruttura DCAP	14
3.2 Applicazione base: helloworld	20
3.2.1 Gramine	20
3.2.2 OpenEnclave	22
3.3 Remote attestation	27
3.3.1 Gramine	27
3.4 Applicazione avanzata: sqlite3	32
4 Sviluppi futuri: Intel Trusted Domains Extentions (TDX)	33
4.1 Virtualizzazione	33
4.2 Memory protection	33
4.3 Attestazione	34
Riferimenti	i

	Intel SGX	AMD SEV	ARM TrustZone
Target devices	Client PCs	Servers	Mobile devices
Trust anchor	CPU hardware e microcode	Platform security processor	TZ hardware e ARM trusted firmware
Cache side-channel protection	No	No	No
Multiple security domains	Si	Si	No
Security peripherals	No	No	Si

Tabella 1: Confronto tra Intel SGX, AMD SEV e ARM TrustZone

1 Confidential computing

1.1 Trusted Execution Environment (TEE)

I Trusted Execution Environment realizzano una forma di isolamento dei processi e delle applicazioni in hardware per rendere sistemi resistenti ad attacchi fisici e *side-channel*. I TEE sono pensati per essere utilizzati dai sistemi operativi (o hypervisor) in maniera alternativa a TPM e HSM. Ogni azienda propone una propria implementazione dei TEE che può andare dalla protezione di applicazioni in spazio utente (come Intel-SGX) a macchine virtuali sicure come (AMD-SEV e Intel TDX). In questo senso, ogni implementazione fa riferimento ad un *threat model* differente e può includere diversi elementi nella **Trusted Computing Base (TCB)**. La Tabella 1 riporta una panoramica delle principali tecnologie commerciali ed è tratta da [12]. I 2 approcci principali (*user-space isolation* e *VM isolation*) sono illustrati in Figura 1 (tratta da [12]) in cui viene mostrata anche la soluzione **Sanctum** sviluppata per Risc-V.

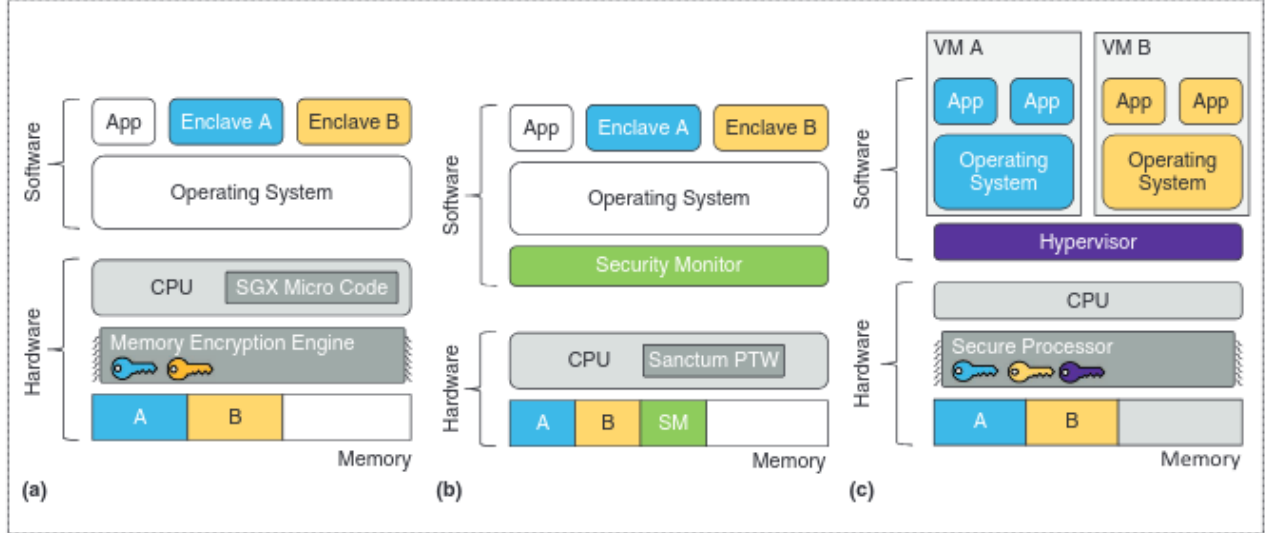


Figura 1: Implementazione TEE in diverse architetture. (a) Intel SGX. (b) Risc-V Sanctum. (c) AMD SEV

1.2 Intel Software Guards Extensions (SGX)

Intel SGX è la prima forma di confidential computing proposta da Intel con supporto hardware. Lo scopo principale di SGX è quello di fornire ad un'applicazione delle istruzioni dedicate per realizzare delle applicazioni *trusted*. In un'applicazione trusted, sia dati che istruzioni non lasciano la CPU in chiaro assicurando tre proprietà:

- **Confidenzialità:** un'altra applicazione non può leggere le istruzioni e i dati di un'applicazione trusted;

- **Integrità:** sia all'avvio che durante l'esecuzione dell'applicazione viene effettuato un controllo crittografico dello spazio di memoria.
- **Attestazione:** le applicazioni devono poter dimostrare la loro *trustworthiness* sia ad altre applicazioni (*local attestation*) che a sistemi remoti (*remote attestation*);

SGX aggiunge all'ISA dell'architettura classica 18 istruzioni con le quali si può creare una regione confidenziale, eseguire il processo di attestazione e verificare l'integrità dell'applicazione. SGX rende fattibile un threat model in cui sono considerati trusted solo la CPU intel e il codice in esecuzione nell'enclave assumendo che un attore malevolo possa prendere effettuare sia attacchi hardware (eg. *cold boot*) che al livello OS (eg. *ROP*).

1.2.1 Enclave

L'*enclave*[6] è la prima astrazione fornita da SGX e definisce un perimetro di sicurezza all'interno dello spazio di indirizzamento virtuale. La memoria all'interno dell'enclave è cifrata e accessibile solo dall'interno dell'enclave stessa. La memoria viene cifrata con un **MEE** (*Memory Encryption Engine*) la cui implementazione non è stata discussa ufficialmente.

Codice, dati e metadati di un'enclave vengono gestite da SGX in un'area riservata della DRAM detta **PRM** (*Processor Reserved Memory*) che contiene la **EPC** (*Enclave Page Cache*) a cui si accede solo attraverso le istruzioni SGX. La EPC ha una dimensione di 128/256MB e può essere vista come una *cache L3*, ma non impone alcun limite alla dimensione delle enclave visto che è supportato il processo di *eviction*. Il compito di gestire la tabella delle pagine è affidato al sistema operativo o hypervisor. Dato che non il gestore della memoria non è parte della TCB, SGX controlla le operazioni effettuate dal gestore della memoria attraverso la **EPCM** (*Enclave Page Cache Map*).

Un'applicazione trusted deve creare un'enclave con l'istruzione *ECREATE*. SGX assegna per ogni enclave una **SECS** (*SGX Enclave Control Structure*) in cui vengono conservati tutti i metadati per enclave. In particolare, i metadati sono memorizzati sottoforma di **attributi**.

Alla creazione, viene allocata una SECS insieme alla EPC. Con l'istruzione *EADD* vengono caricati codice e dati dell'applicazione: nello specifico vengono copiate pagine dalla parte non trusted del sistema a quella dell'enclave con un controllo validità. Per essere eseguita, l'enclave deve ottenere un **EINIT Token Structure**. Questo token viene fornito da un'enclave privilegiata detta (**LE**) *Launch Enclave* fornita da Intel e firmata con una chiave speciale all'interno dell'architettura SGX. Solo la **LE** può chiamare la *EINIT* che può impostare l'attributo **INIT** a 1. La terminazione viene fatta con *EREMOVE* che dealloca la EPC e la EPCM.

Quando un'enclave entra in esecuzione il processore opera in **enclave mode** con la quale riceve i privilegi di accedere alla EPC. Per poter transitare in enclave mode, il processore deve eseguire una *EENTER* e per lasciare la modalità esegue una *EEXIT*. *EENTER* non effettua un cambio di privilegio (si esegue sempre a **ring 3**), ma effettua un salto memorizzando il **RIP** (*Return Instruction Pointer*). *EENTER* è un'operazione sincrona. *EEXIT* può essere eseguita solo da un processore in enclave mode ed effettua un salto al RIP memorizzato durante il *EENTER* in maniera sincrona. Nel caso in cui ci sia un'eccezione, l'uscita avviene in maniera asincrona con *AEX* (*Asynchronous Enclave Exit*).

Limiti di SGX Intel SGX, pur offrendo un elevato livello di sicurezza, presenta alcune limitazioni intrinseche che devono essere considerate durante lo sviluppo di applicazioni. Queste limitazioni derivano principalmente dal design dell'architettura e dalle restrizioni imposte per mantenere l'integrità e la sicurezza dell'enclave. Di seguito sono elencate alcune delle principali limitazioni di SGX:

- **Limitazioni di memoria:** la dimensione della *Enclave Page Cache* (EPC) è limitata a 128/256 MB, il che può rappresentare un vincolo significativo per applicazioni che richiedono grandi quantità di memoria. Sebbene SGX supporti il processo di *eviction*, questo può introdurre overhead prestazionale;
- **Assenza di protezione contro attacchi side-channel:** SGX non fornisce protezione contro attacchi side-channel basati su cache, timing, o power analysis. Questi attacchi possono potenzialmente rivelare informazioni sensibili anche se l'enclave è isolata;
- **Limitazioni nelle operazioni di I/O:** le enclavi SGX non possono eseguire direttamente operazioni di I/O. Tutte le operazioni di I/O devono essere delegate al sistema operativo host, il che richiede un'uscita dall'enclave (*ocall*);

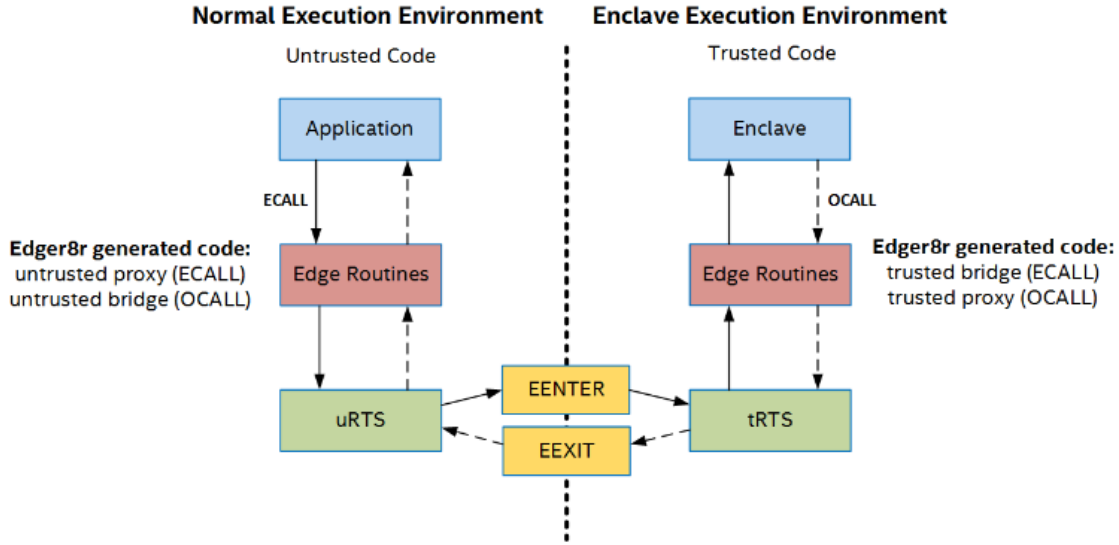


Figura 2: Ciclo di vita di un'enclave

Comunicazione con mondo esterno Date le limitazioni, SGX fornisce un meccanismo per interagire con l'input output attraverso le **Enclave Call (ECALL)** (per eseguire una funzione in ambiente sicuro) e le **Out Call (OCALL)** (esegue una funzione in spazio non *trusted*). Queste due tipologie di operazioni non comportano un cambio di privilegio strettamente, ma, ad esempio una **OCALL** consente di eseguire una system call (o di interfacciarsi con l'I/O). Questa sequenza di operazioni è una delle principali fonti di degradazione delle prestazioni in applicazioni I/O bound in SGX perchè il sistema spende molto tempo nella fase **EENTER** e **EEXIT**, come illustrato in Figura 2

1.2.2 Attestazione

L'attestazione è un processo fondamentale nel confidential computing che consente di verificare l'integrità e l'affidabilità di un'applicazione o di un ambiente di esecuzione. Questo processo permette a un'entità di dimostrare a un'altra entità che il proprio stato è sicuro e che non è stato compromesso. Esistono due tipi principali di attestazione:

- locale: avviene tra componenti all'interno dello stesso sistema;
- remota: coinvolge la verifica da parte di un'entità esterna.

In SGX, l'attestazione viene creata da un'altra enclave privilegiata detta **QE (Quoting Enclave)** gestita da Intel che può accedere alla chiave per firmare l'attestazione. Prima di procedere con l'attestazione remota, bisogna creare un canale di comunicazione sicuro con la quoting enclave e, pertanto, è previsto un sistema di attestazione locale.

Una volta creato, l'enclave genera un report, popolato i campi con la relativa SECS, con l'istruzione **ERREPORT**, lo firma con una chiave scambiata con la **QE** e allega anche un MAC per il controllo dell'integrità.

È bene notare che la chiave con cui viene firmato il report dalla **QE** non è presente all'interno di SGX all'uscita dalla fabbrica, ma viene generata utilizzando un **Provisioning Enclave** con l'istruzione **EGETKEY** il cui processo non è documentato, ma ha come input valori accessibili solo all'interno del processore come il **Secure Version Number (SVN)**.

L'attestazione può essere fatta con due procedimenti: **EPID** (Figura 3a, utilizzata da remoto) o con **DCAP** (Figura 3b). Le due figure sono molto simili, ma con **DCAP** il servizio Intel viene contattato solo una volta dal **Provisioning Enclave** e non ogni volta come in **EPID**.

1.2.3 SGX2

L'architettura descritta in §1.2.1 è implementata completamente in SGX1. In [13], è mostrata un'estensione della versione base di SGX a cui ci si riferisce come **SGX2**. Il componente principale è l'**Enclave Dynamic Memory Management (EDMM)** che permette di gestire dinamicamente le pagine di memoria dall'interno di un enclave

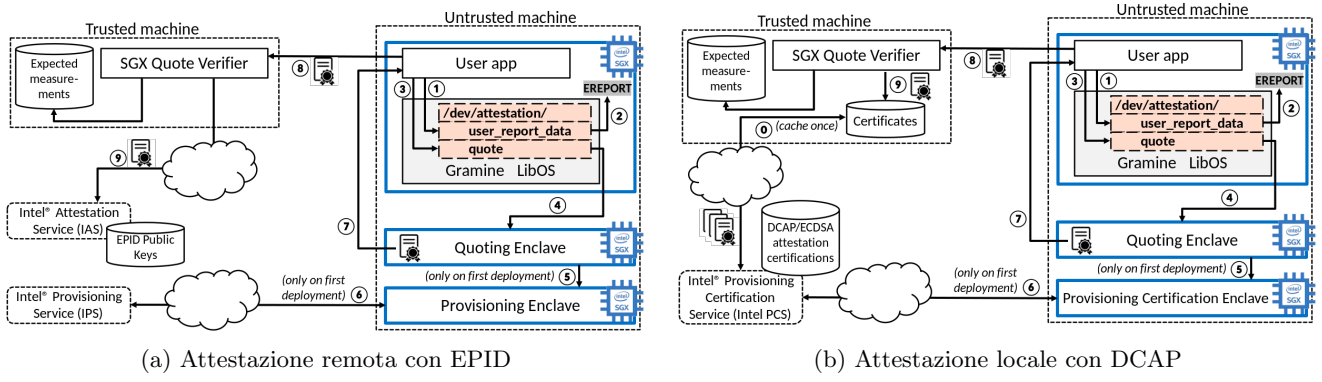


Figura 3: Confronto tra attestazione remota con EPID e DCAP

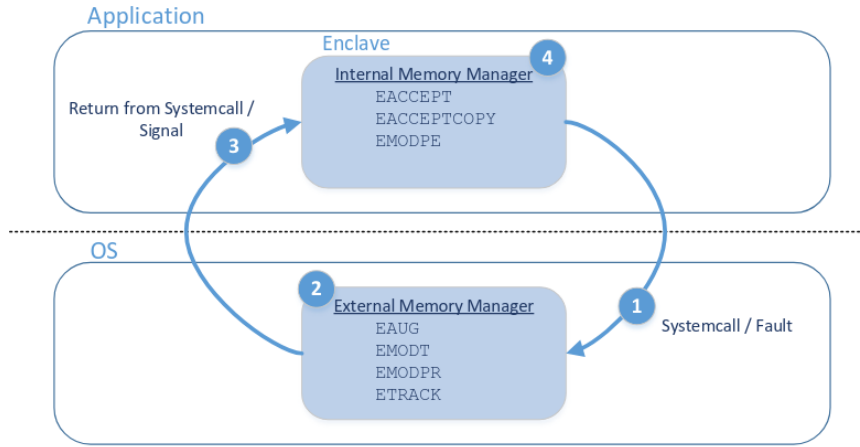


Figura 4: Gestione dinamica della memoria in SGX2

aggiungendo istruzioni per agire sulla secondo un architettura mostrata in Figura 4. In [13], sono illustrati i protocolli con cui *External Memory Manager* e *Internal Memory Manager* comunicano per effettuare modifiche sulla EPC.

2 Strumenti

2.1 Classificazione

Gli strumenti utilizzati per realizzare applicazioni *trusted* che operano in TEE sono diversi e possono essere classificati secondo diversi criteri:

- astrazione dall'architettura hardware: alcuni strumenti si interpongono al sistema operativo fornendo un insieme di ABI riscritte su cui si effettuano dei controlli o, come nel caso di *occlum* fornisce un insieme di servizi che le applicazioni possono usare;
- trasparenza con il codice sorgente: alcuni strumenti come *open enclave* richiede una ristrutturazione dell'applicazione in cui si distinguono chiaramente le parti sicure da quelle non sicure avendo come granularità la funzione;

Una panoramica degli strumenti analizzati è riportata in Tabella 2.

	Astrazione dall'architettura hardware	Trasparenza con il codice sorgente
Intel SGX SDK	Specifico per SGX, fornisce un insieme completo di API per sfruttare direttamente le capacità hardware di Intel SGX.	Richiede la divisione esplicita tra codice sicuro (enclave) e non sicuro, con un maggiore onere per lo sviluppatore.
Rust SGX	Specifico per SGX. Implementa un target diverso per il compilatore Rust	Supporta solo applicazioni scritte in Rust
Open Enclave SDK	Supporta diverse architetture hardware (ad es. Intel SGX, ARM TrustZone) attraverso un'astrazione comune.	Richiede una chiara distinzione tra le funzioni sicure e non sicure nel codice sorgente, con una ristrutturazione esplicita.
Gramine-SGX	Implementa un livello di compatibilità (libOS) per eseguire applicazioni non modificate in SGX, rendendole indipendenti dall'hardware sottostante.	Non necessita di modifiche al codice sorgente dell'applicazione, facilitando l'integrazione di software legacy.
Occlum	Fornisce un insieme di ABI riscritte e servizi di runtime per operare in TEE, semplificando l'implementazione e l'adattamento al TEE specifico.	L'applicazione non richiede modifiche significative al codice, poiché Occlum maschera le complessità dell'hardware sottostante.
Enarx	Astratto rispetto all'hardware, utilizza WebAssembly come formato intermedio, garantendo portabilità su diversi TEE.	Le applicazioni devono essere portate in WebAssembly, richiedendo un adattamento potenzialmente significativo al codice.
EGo	Specifico per SGX. Esegue applicazioni GO in un enclave.	Richiede modifiche minimali al codice. Non supporta il multiprocessing e altre funzionalità di sistema.
MysticOS	Sistema operativo leggero, basato su Open Enclave. Quindi si propone di essere <i>hardware-agnostic</i>	Comporta molte limitazioni per la comunicazione con il sistema operativo.

Tabella 2: Confronto tra strumenti per lo sviluppo di applicazioni trusted in TEE

2.2 Gramine

Gramine[1] (ex Graphene) è un **libOS** che fornisce un'astrazione di alto livello in ambiente Linux per l'esecuzione di applicazioni isolate. Gramine è stato introdotto pre-SGX, ma è stato integrato con il concetto di enclave con **Gramine-SGX**. L'idea principale è quella di rendere applicazioni sicure senza modificare il codice sorgente, ma di effettuare un *wrap* di un eseguibile in un ambiente isolato utilizzando SGX. L'astrazione principale di Gramine è il *picoprocesso*[4]. Gramine inietta un componente detto **PAL** (*Platform Abstraction Layer*) che si occupa di intercettare le *system call* dell'applicazione e gestirle in modo sicuro garantendo che solo le operazioni autorizzate vengano eseguite.

Di seguito, una panoramica di tutti gli elementi configurabili attraverso il *manifest file*.

- **loader**: Carica l'applicazione nell'enclave, gestendo l'eseguibile principale le variabili d'ambiente;
- **libos**: libOS emula un mini sistema operativo all'interno dell'enclave, permettendo all'applicazione intercettare le chiamate di sistema illegali all'interno dell'enclave ed implementa il protocollo previsto per comunicare tra mondo sicuro e non;
- **sgx**: contiene le impostazioni specifiche per l'enclave SGX, inclusa la modalità di debug, la dimensione dell'enclave e i file trusted;
- **fs**: definisce come i file e le directory sono montati all'interno dell'enclave, permettendo di specificare percorsi di file system virtuali e di configurare mount point cifrati per proteggere i dati sensibili;
- **sys**: gestisce le impostazioni di sistema per l'enclave, inclusa la dimensione dello stack e la configurazione di nomi di dominio runtime aggiuntivi;

2.2.1 Struttura di un'applicazione con Gramine

Come detto in §2.1, Gramine non prevede nessuna modifica al codice sorgente, ma prevede una specifica statica di tutti gli elementi dell'enclave attraverso un *manifest*. Il manifest contiene informazioni per configurare il libOS e per configurare l'enclave fornendo opzioni come *stack size* e *enclave size*. Inoltre, contiene una lista di tutti i *file* a cui può accedere l'enclave.

Il manifest è un *file* in formato *toml*. Attraverso l'utility *gramine-manifest*, il manifest può essere personalizzato usando un template *jinja* che supporta costrutti iterativi e condizionali e offre un meccanismo per iniettare variabili all'interno del file. In listato 1, è riportato un esempio minimale di manifest file. L'entrypoint è un eseguibile chiamato *program* che deve essere necessariamente specificato come *trusted_file*. Se l'enclave accede ad un file non *trusted* si verificherà un errore.

Listato 1: Esempio minimale di manifest file

```
1 libos.entrypoint = "/program"
2
3 loader.env.LD_LIBRARY_PATH = "/lib"
4
5 fs.mounts = [
6   { path = "/lib", uri = "file:{{ gramine.runtimedir() }}" },
7   { path = "/program", uri = "file:program" },
8 ]
9
10 sgx.trusted_files = [
11   "file:program",
12   "file:{{ gramine.runtimedir() }}/",
13 ]
```

A partire dal file processato (viene calcolato un *hash* (sha256) per tutti i file trusted), è possibile firmare il manifest con una chiave *rsa* a 3072 – *bit* ed eseguire l'enclave.

Un manifest file più complicato è mostrato in listato 2. I *mount point* possono essere di diverso tipo. Ad esempio, *tmpfs* è un file system in memoria privato all'enclave cifrato; *encrypted* specifica che i file sono cifrati su disco e vengono acceduti da gramine in maniera trasparente. I mountpoint cifrati possono specificare una chiave inserendola nel manifest (altamente insicuro) o indicare il nome di una chiave ottenuta con l'attestazione remota e devono essere dedicati per enclave. Quando non specificato, il mountpoint è di tipo *chroot* che in maniera simile a come avviene in Linux permette di collegare file all'interno dell'ambiente dell'enclave. SGX consente la presenza di file non controllati condivisi con l'host con l'opzione *sgx.allowed_files*.

Infine, è possibile configurare opzioni per la dimensione dello stack e dell'enclave.

Listato 2: "Manifest con allowed_files mountpoint di diversa e configurazione per l'enclave"

```
1 libos.entrypoint = "/program"
2
3 loader.env.LD_LIBRARY_PATH = "/lib"
4
5 sys.stack.size = "2M"
6 sys.enable_extra_runtime_domain_names_conf = true
```



```

7
8 sgx.debug = true
9 sgx.edmm_enable = {{ 'true' if env.get('EDMM', '0') == '1' else 'false' }}
10 sgx.enclave_size = "1G"
11
12 fs.mounts = [
13   { path = "/lib", uri = "file:{{ gramine.runtimedir() }}" },
14   { path = "/program", uri = "file:program" },
15   { type = "tmpfs", path = "/tmp" },
16   { type = "encrypted", path = "/encrypted/", uri = "file:encrypted/" },
17 ]
18 fs.insecure_keys.default = "<hardcoded-key>"
19
20 sgx.trusted_files = [
21   "file:program",
22   "file:{{ gramine.runtimedir() }}/",
23 ]
24
25 sgx.allowed_files = [
26   "file:shared/"
27 ]

```

2.2.2 Remote attestation

Come approfondito in §3.1.3, la procedura con EPID è deprecata in favore della *Data Center Attestation Primitives* (DCAP). Gramine costruisce API di alto livello basate sulla remote attestation per creare facilmente tunnel TLS e fornire alle *enclave* segreti. Gramine espone 3 livelli di astrazione per fare utilizzare l'attestazione remota:

- Low level interface: utilizza i pseudo files in Linux per eseguire le istruzioni SGX come EREPORT per generare quoting e report comunicando con le *architectural enclave*;
- RA-TLS: un insieme di due librerie utilizzate per aggiungere e verificare un SGX quote all'interno di un certificato X.509;
- Secret Provisioning: un API di alto livello per generare un segreto condiviso tra due applicazioni;

Per utilizzare l'attestazione, bisogna aggiungere *sgx.remote_attestation* nel manifest e impostare il tipo di attestazione utilizzata (al momento solo DCAP) come di seguito.

```
sgx.remote_attestation = "dcap"
```

Low-level interface L'enclave che utilizza la remote attestation può accedere ai file speciali in */dev/attestation* (riportati di seguito).

```

dev/attestation/
├── attestation_type
├── user_report_data
├── target_info
├── my_target_info
├── report
├── quote
├── keys
│   └── default
│       └── <key names>

```

Lo pseudo-file *report* contiene il report SGX generato a partire dal contenuto in *user_report_data* (contiene una stringa di questo inclusa nel report di 64B) e *target_info* (contiene 512B di dati contenenti al target info). Il

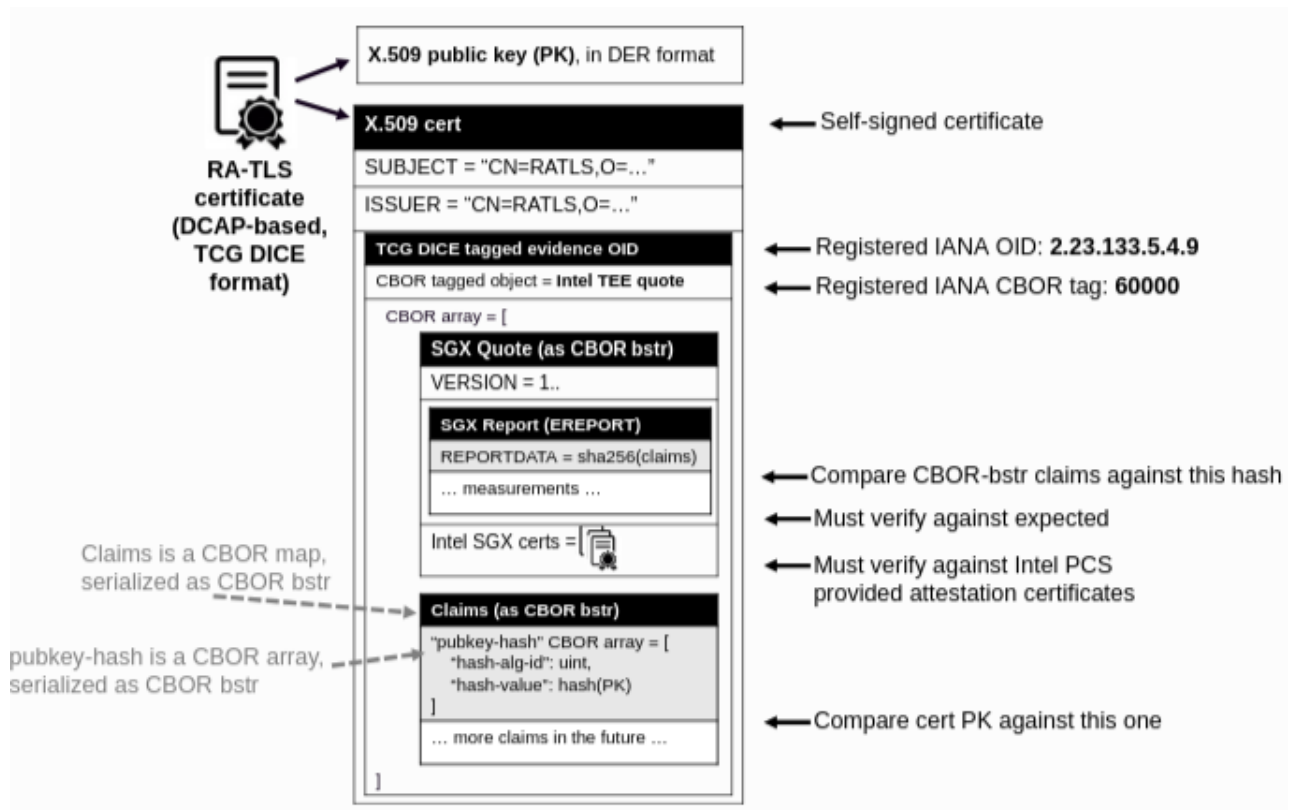


Figura 5: Certificato X.509 arricchito con i dati del report SGX

file *attestation.type* contiene il nome dello schema di attestazione in uso. Infine, l'SGX quote può essere letto da */dev/attestation/quote*. Quando si comunica con i dispositivi virtuali, SGX utilizza le istruzioni privilegiate come EREPORT e permette di comunicare con le *architectural enclave* (quali *Quoting Enclave*, *Provisioning Certificate Enclave*) attraverso il **Architectural Enclave Service Manager (AESM)**.

Infine, nella cartella *keys* sono presenti le chiavi di crittografia ottenute da un servizio remoto come file binari di 128 – bit.

RA-TLS Gramine utilizza le API di basso livello per fornire la possibilità di creare un tunnel TLS attestato con **RA-TLS**: un insieme di due librerie che integra il protocollo TLS aggiungendo informazioni sul *quote* all'interno del certificato **X.509** (Figura 5). Le funzioni sono documentate in *tools/sgx/ra-tls.h* in [10]

- **ra-tls-attest.so**: contiene la funzione *ra-tls-create-key-and-crt-der()* che serve ad aggiungere il quote in un certificato self-signed
- **ra-tls-verify-dcap.so**: contiene la funzione *ra-tls-verify-callback-der()* utilizzata dal client per verificare il certificato. Anche se è possibile specificare una callback per verificare la callback, la funzione confronta i dati ricevuti dal server con le variabili d'ambiente **RA-TLS_*** (MRENCLAVE, MRSIGNER, IS_PROV_ID);

Secret provisioning La secret provisioning è un'API di alto livello fornita attraverso due librerie e permette a due applicazioni che eseguono in TEE di negoziare segreti condivisi utilizzando la RA-TLS. Il caso d'uso più semplice è quello di un'applicazione client eseguita in un'enclave che riceve da un server delle chiavi per decifrare delle immagini o un modello.

Le librerie fornite sono:

- **secret-prov-attest.so**: libreria utilizzata dal client. Una volta terminato il provisioning, il segreto viene fornito nella variabile d'ambiente *SECRET_PROVISIONING_SECRET_STRING*. Inoltre, la chiave viene salvata in */dev/attestation/keys/key-name*;

- *secret_prov_verify_dcap.so*: libreria utilizzata dal server (che non esegue in un enclave) che richiama RA-TLS per verificare la firma del client.

2.3 OpenEnclave

Open enclave è un progetto che astrae il concetto di enclave proposto da Intel SGX in modo da renderlo *hardware-agnostic* con l'intenzione di portarlo su altre piattaforme come ARM Trustzone. Come riportato in Tabella 1, Open Enclave richiede di suddividere il codice sorgente in *enclave* (trusted) e *host* (non trusted) con delle funzioni esplicite per l'esecuzione dell'enclave. In particolare, l'idea principale consiste nel generalizzare l'SDK proposta di Intel per SGX[2].

2.3.1 Struttura di un'applicazione con Open Enclave

L'applicazione viene definita mediante un *file* scritto in **Enclave Definition Language (EDL)** definito da Intel nell'SDK[2]. Il linguaggio consente di importare altre definizioni e definisce una separazione netta tra parte sicura e parte non sicura. Come mostrato in listato 3, è riportato un esempio in cui si definiscono 2 funzioni:

- *enclave_helloworld()*: funzione da eseguire nell'enclave;
- *host_helloworld()*: funzione host non sicura;

L'enclave importa delle definizioni standard come le *syscall* in cui è implementato il meccanismo di comunicazione tra *secure world* e *non secure world*. Inoltre, può essere anche importata una libreria condivisa tra le due parti.

Listato 3: Esempio base di un file EDL

```

1 enclave {
2     from "openenclave/edl/syscall.edl" import *;
3     from "platform.edl" import *;
4
5     include "shared.h"
6
7     trusted {
8         public void enclave_helloworld();
9     };
10
11     untrusted {
12         void host_helloworld();
13     };
14 };

```

Analogamente a come avviene per IntelSDK, il file viene usato per generare tutte le funzioni e le definizioni su cui strutturare l'applicazione con l'utility *oedger8r*, come mostrato in §3.2.2.

2.3.2 Attestazione

In OpenEnclave, l'attestazione remota è supportata solo per le macchine in Azure. Pertanto, gli esempi non includono la parte di comunicazione con i servizi remoti. Sebbene in modo più complicato, OpenEnclave crea un'astrazione simile a RA-TLS che può essere utilizzata per generare negoziare un segreto tra due enclave.

3 Sviluppo di un'applicazione sicura

3.1 Setup di laboratorio

Gli esempi che seguono sono stati realizzati utilizzando il setup riportato in Tabella 3. Come indicato in , esistono 2 tipi di remote attestation: **Enhanced Privacy ID (EPID)** e **Data Center Attestation Primitives (DCAP)**. EPID incarica la QE di mettersi in contatto con l'**Intel Attestation Service (IAS)** per avviare un algoritmo che consente di verificare la TCB. Purtroppo, questo meccanismo (più semplice da configurare) è stato deprecato e l'IAS sarà chiuso definitivamente in data *2 aprile 2025*. L'unica modalità di fare attestazione remota è quella di configurare DCAP con uno sforzo infrastrutturale del cloud provider. Questa scelta è data anche dalla volontà di Intel di unificare i servizi di remote attestation tra SGX e TDX.

Componente	Descrizione
CPU	i7-9700 CPU @ 3.00GHz, SGX1 e FLC
RAM	24 GB, 2666 MT/s DDR4
OS	Ubuntu 24.04 LTS, kernel 6.8.0-41-generic
Platform	Dell Optiplex-7070
SGX SDK version	2.25
SGX DCAP version	1.22
Docker version	27.3.1
Gramine Version	v1.8
OpenEnclave Version	v0.19.8

Tabella 3: Caratteristiche tecniche macchina di laboratorio

Sia Gramine che OpenEnclave necessitano di componenti specifiche per la versione di Ubuntu utilizzata. Per assicurare la riproducibilità degli esperimenti, è stato utilizzato Docker. A partire dal kernel 5.11, SGX con l'attestazione DCAP è incluso come driver in upstream e, pertanto, non necessita di installazioni. In particolare, sono forniti due dispositivi Linux che devono essere utilizzati all'interno dei container per accedere alle istruzioni SGX: */dev/sgx.enclave* e */dev/sgx.provision*. Il codice sorgente dei driver e dell'SDK è riportato in [11].

3.1.1 Installazione Gramine

Per usare correttamente le primitive DCAP, è necessario compilare da codice sorgente Gramine con il flag *-Ddcap=enabled*. Per rendere riproducibile la procedura di compilazione, installazione e configurazione è stato sviluppato un Dockerfile (riportato in listato 4). È possibile creare un'immagine con:

```
docker build -t gramine-dcap .
```

Listato 4: Dockerfile per la creazione di un ambiente di sviluppo riproducibile con Gramine

```
1 FROM ubuntu:24.04
2
3 WORKDIR /tmp
4
5 # install dependencies
6 RUN apt-get update && DEBIAN_FRONTEND=noninteractive apt-get install -y \
7     vim \
8     curl \
9     gnupg2 \
10    cmake \
11    psmisc \
12    make \
13    build-essential \
14    autoconf \
15    bison \
16    gawk \
17    nasm \
18    ninja-build \
19    pkg-config \
20    meson \
21    libprotobuf-c-dev \
```

```

22  protobuf-c-compiler \
23  libprotobuf-c-dev \
24  protobuf-c-compiler \
25  python3 \
26  python3-click \
27  python3-jinja2 \
28  python3-pip \
29  python3-pyelftools \
30  python3-pyelftools \
31  python3-tomli \
32  python3-tomli-w \
33  python3-voluptuous \
34  python3-cryptography \
35  python3-protobuf
36
37 # configure intel sgx sdk
38 RUN curl -fsSLo /usr/share/keyrings/intel-sgx-deb.asc https://download.01.org/intel-sgx/sgx_repo/
    ubuntu/intel-sgx-deb.key && \
39  echo "deb [arch=amd64 signed-by=/usr/share/keyrings/intel-sgx-deb.asc] https://download.01.org/
    intel-sgx/sgx_repo/ubuntu noble main" \
40  | tee /etc/apt/sources.list.d/intel-sgx.list
41
42 RUN apt-get update && DEBIAN_FRONTEND=noninteractive apt-get install -y libsgx-epid libsgx-quote-ex
    libsgx-dcap-ql
43
44 RUN curl -O https://download.01.org/intel-sgx/sgx-dcap/1.22/linux/distro/ubuntu24.04-server/
    sgx_linux_x64_sdk_2.25.100.3.bin
45 RUN chmod +x sgx_linux_x64_sdk_2.25.100.3.bin
46 RUN ./sgx_linux_x64_sdk_2.25.100.3.bin --prefix /opt/intel/
47
48 RUN DEBIAN_FRONTEND=noninteractive apt-get install -y \
49  libsgx-enclave-common-dev \
50  libsgx-dcap-ql-dev \
51  libsgx-dcap-default-qpl-dev \
52  libsgx-dcap-quote-verify-dev
53
54 # download and build gramine
55 WORKDIR /workspace
56 RUN curl -O -L https://github.com/gramineproject/gramine/archive/refs/tags/v1.8.tar.gz
57 RUN tar xvf v1.8.tar.gz
58 RUN mv gramine-1.8 /workspace/gramine
59 RUN rm v1.8.tar.gz
60
61 WORKDIR /workspace/gramine
62 RUN meson setup build/ \
63  --buildtype=release \
64  -Ddirect=enabled \
65  -Dsgx=enabled \
66  -Ddcap=enabled
67
68 RUN ninja -C build/
69 RUN ninja -C build/ install
70
71 RUN echo "source /opt/intel/sgxsdk/environment" >> /root/.bashrc
72
73 # avoid "Signing key does not exist" error
74 RUN gramine-sgx-gen-private-key
75
76 # configure AESM - Architectural Enclaves Service Manager
77 RUN echo "#!/bin/sh \n \
78  set -e \n \
79  killall -q aesm_service || true \n \
80  AESM_PATH=/opt/intel/sgx-aesm-service/aesm LD_LIBRARY_PATH=/opt/intel/sgx-aesm-service/aesm exec
    /opt/intel/sgx-aesm-service/aesm/aesm_service --no-syslog \n\
81  " >> /restart_aesm.sh
82
83 RUN mkdir -p /var/run/aesmd
84 RUN chmod +x /restart_aesm.sh
85
86 # add user to sgx_prv to access remote attestation primitives
87 RUN groupadd sgx_prv
88 RUN usermod -aG sgx_prv root

```

```
89
90 ENTRYPOINT ["/bin/sh", "-c"]
91 CMD ["/restart_aesm.sh ; exec /bin/bash"]
```

```
docker run \
-it \
--network=host \
--device /dev/sgx_enclave \
--device /dev/sgx_provision \
gramin-dcap
```

3.1.2 Installazione Open Enclave

OpenEnclave è supportato solo con Ubuntu 20.04 a causa di alcune dipendenze con OpenSSL. Per questo motivo, l'SDK è stata installata utilizzando un container *Docker* utilizzando il Dockerfile (creato a partire dalla documentazione [3]) in listato 5. A partire da questo Dockerfile, può essere creata un'immagine con (supponendo di essere nella stessa cartella del file):

```
docker build -t openenclave-sdk .
```

Listato 5: Dockerfile per la creazione di un ambiente di sviluppo riproducibile con OpenEnclave SDK

```
1 FROM ubuntu:20.04
2
3 WORKDIR /workspace
4
5 # install dependencies
6 RUN apt-get update && DEBIAN_FRONTEND=noninteractive apt-get install -y \
7     gnupg2 \
8     curl \
9     make \
10    build-essential
11
12 RUN echo 'deb [arch=amd64] https://download.01.org/intel-sgx/sgx_repo/ubuntu focal main' | tee /etc
13     /apt/sources.list.d/intel-sgx.list
14 RUN curl https://download.01.org/intel-sgx/sgx_repo/ubuntu/intel-sgx-deb.key | apt-key add -
15
16 RUN echo "deb http://apt.llvm.org/focal/ llvm-toolchain-focal-11 main" | tee /etc/apt/sources.list.
17     d/llvm-toolchain-focal-11.list
18 RUN curl https://apt.llvm.org/llvm-snapshot.gpg.key | apt-key add -
19
20 RUN echo "deb [arch=amd64] https://packages.microsoft.com/ubuntu/20.04/prod focal main" | tee /etc/
21     apt/sources.list.d/msprod.list
22 RUN curl https://packages.microsoft.com/keys/microsoft.asc | apt-key add -
23
24 # install openenclave-sdk
25 RUN apt-get update && DEBIAN_FRONTEND=noninteractive apt-get -y install clang-11 \
26     libssl-dev \
27     gdb \
28     libsgx-enclave-common \
29     libsgx-quote-ex \
30     libprotobuf17 \
31     libsgx-dcap-ql \
32     libsgx-dcap-ql-dev \
33     az-dcap-client \
34     open-enclave
35
36 RUN echo "source /opt/openenclave/share/openenclave/openenclaverc" >> /root/.bashrc
37
38 # download open-enclave-sdk for samples
39 RUN curl -L -o /tmp/v0.19.8.tar.gz https://github.com/openenclave/openenclave/archive/refs/tags/v0
40     .19.8.tar.gz
41
42 RUN tar xvf /tmp/v0.19.8.tar.gz
43 RUN mv openenclave-0.19.8 openenclave
44
45 # configure AESM - Architectural Enclaves Service Manager
```

```

42 RUN echo "#!/bin/sh \n \
43     set -e \n \
44     killall -q aesm_service || true \n \
45     AESM_PATH=/opt/intel/sgx-aesm-service/aesm LD_LIBRARY_PATH=/opt/intel/sgx-aesm-service/aesm exec
      /opt/intel/sgx-aesm-service/aesm/aesm_service --no-syslog \n\
46     " >> /restart_aesm.sh
47
48 RUN mkdir -p /var/run/aesmd
49 RUN chmod +x /restart_aesm.sh
50
51 # add user to sgx_prv to access remote attestation primitives
52 RUN groupadd sgx_prv
53 RUN usermod -aG sgx_prv root
54
55 ENTRYPOINT ["/bin/sh", "-c"]
56 CMD ["/restart_aesm.sh ; exec /bin/bash"]

```

Creata l'immagine, è possibile avere un terminale interattivo all'interno di un container condividendo il codice sorgente dell'applicazione (supponendo di essere nella *top-level directory*) del proprio progetto.

```

docker run \
  -it \
  --network=host \
  --device /dev/sgx_enclave \
  --device /dev/sgx_provision \
  openenclave-sdk \

```

3.1.3 Configurazione infrastruttura DCAP

L'infrastruttura DCAP (Figura 6) mira a minimizzare le interazioni con i servizi remoti di Intel concentrando la gestione dei certificati su un componente *trusted* detto **Provisioning Certificate Caching Service (PCCS)**. Il PCCS è l'unico componente dell'infrastruttura che può interagire con i servizi di Intel (attraverso una API Key, Figura 7) e può essere contattato direttamente dai quoting enclave delle singole macchine come un servizio HTTP. A partire dalla versione DCAP 1.22, Intel non offre più un'implementazione di riferimento di questo servizio, ma rimane comunque accessibile dalle release precedenti. Tutte le informazioni riportate sono state ottenute dai documenti e dal codice sorgente rilasciati da Intel[8],[9].

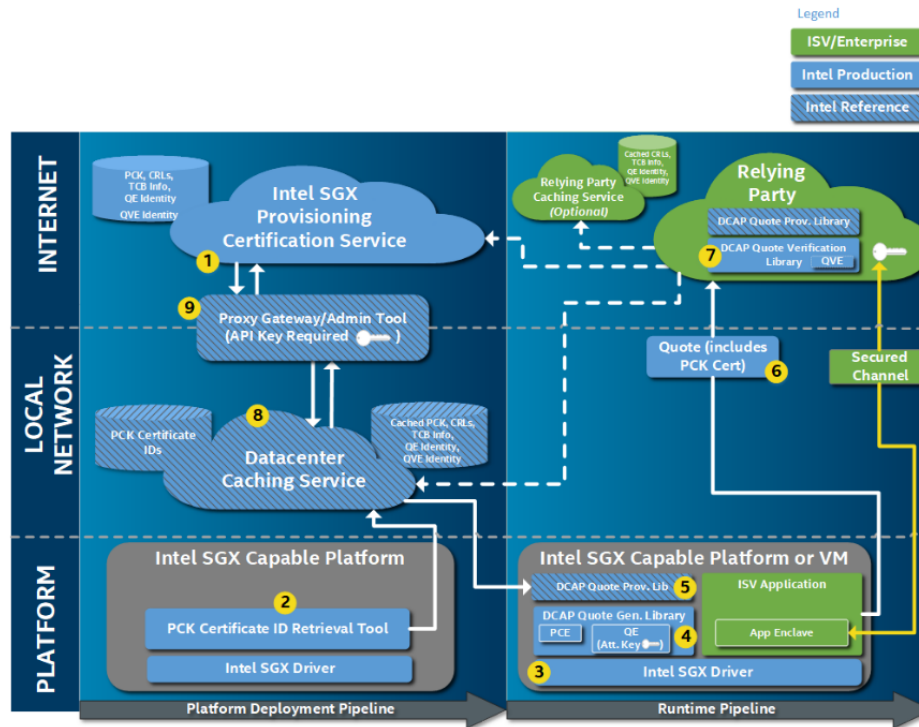


Figura 6: Diagramma di infrastruttura DCAP

Le API Key possono essere ottenute registrando un account su questo portale.

Services for Intel hardware-based Trusted Execution Environments (TEEs)

Intel provides both registration and provisioning services for selected Intel® Xeon® E processors and Intel® Xeon® Scalable processors (starting from 3rd generation). These services support both Intel® Software Guard Extensions (Intel® SGX) and Intel® Trust Domain Extension (Intel® TDX) services, providing the data and collateral to enable third party remote attestation using the Elliptic Curve Digital Signature Algorithm (ECDSA).

Intel® SGX and Intel® TDX Registration Service

Intel provides the Registration Service to create a package that will register platform root keys (PRKs) shared between all processors on a platform. Visit the [Intel® SGX and Intel® TDX Registration Service page](#) for more information.

Intel® SGX and Intel® TDX Provisioning Certification Service

Intel provides Provisioning Certification Services enabling retrieval of necessary collateral to attest the Intel® SGX-enabled module or an Intel® TDX-trusted virtual machine (VM). To learn more and subscribe to the service, visit our [Intel® SGX and Intel® TDX Provisioning Certification Service page](#).

Intel® SGX Attestation Service Utilizing Enhanced Privacy ID (EPID)

The Intel® SGX Attestation Service utilizing EPID is only available on select client systems, select Intel® Xeon® E3 processors, and select Intel® Xeon® E processors.

Intel plans to end of life (EOL) this service April 2, 2025. This would include all active API versions. Please factor this into your engagement plans (reference this [link](#) for additional details and Intel-offered attestation alternatives). As previously planned and communicated, Intel has limited access to the Intel Development (DEV) environment as of September 28, 2024.

Intel® SGX Attestation Service enables a relying party to attest an enclave without knowing the specific Intel® processor that the enclave is running on. To learn more and subscribe to the service, visit our [Intel® SGX Attestation Service page](#).

Attestation Service utilizing EPID is only available on select client systems, select Intel® Xeon® E3 processors, and select Intel® Xeon® E processors.

You can go to [here](#) to learn more about Intel® SGX.

Intel® Registration Service for Scalable Platforms

To support Intel® Xeon® Scalable processor-based server platforms, Intel is providing a registration service that creates a package that will register platform root keys (PRKs) that are shared between all of the processors on the platform.

The [Intel® SGX Services](#) and [Intel® TDX Services Terms of Use](#) govern your use of these Services except where we expressly state that separate terms (and not these) apply. By using our services, you are agreeing to these terms. Please make sure you read them carefully.

Intel, the Intel logo and Xeon are trademarks of Intel Corporation or its subsidiaries.

Register Platform

API Documentation

This API enables registration of a PRK that is shared between all processors on the platform.

Add Package

API Documentation

This API allows for managing processors on the platform.



(a) Navigazione sul sito

(b) Click sul pulsante subscribe

Intel® Software Guard Extensions Registration Service

Configuration for APIs that requires subscription in Registration Service

Search APIs

Name	Description
RS Add Package	
RS Register Platform	
Subscribe	

(c) Creazione API Key

Figura 7: Procedura di registrazione sul sito Intel

Configurazione host Anche se si utilizzano ambienti containerizzati, bisogna configurare l'host con i permessi per accedere ai device Linux che si occupano di generare report. In particolare, come mostrato di seguito, bisogna

configurare i permessi sui driver di *sgx_enclave* *sgx_provision* (implementati già nel kernel Linux) conferendo l'accesso in lettura e scrittura per gli utenti appartenenti al gruppo *sgx_prv*. Per fare questo, aggiungere nel file */etc/udev/rules.d/90-sgx-v40.rules* le seguenti espressioni:

```
SUBSYSTEM=="misc",KERNEL=="sgx_enclave",MODE="0666",SYMLINK+="sgx/enclave"
SUBSYSTEM=="misc",KERNEL=="sgx_provision",GROUP="sgx_prv",MODE="0660",SYMLINK+="sgx/provision"
```

e ricaricare le regole con:

```
udevadm trigger
```

Dopo queste modifiche, bisogna aggiungere qualsiasi utente (incluso root in un container Docker) al gruppo *sgx_prv*. Se si utilizzano i Dockerfile mostrati in precedenza, l'utente root è già stato configurato correttamente.

Installazione PCCS Intel specifica l'architettura del PCCS in e fino alla versione 1.21 DCAP fornisce un'implementazione di riferimento. Di seguito, si riporta la procedura di installazione e configurazione per quella implementazione. Il PCCS è un'applicazione web scritta in Javascript e può essere trovata nel path *QuoteGeneration/pccs* della repository nella versione 1.21. Prima di installare, bisogna ottenere delle API Key dal servizio Intel creando un account seguendo questa guida [7]. Per rendere riproducibile la compilazione di questo componente, è stato creato il Dockerfile in listato 6 con cui creare un'immagine:

```
docker build -t pccs .
```

Listato 6: Dockerfile per la compilazione del PCCS

```
1 FROM ubuntu:24.04 AS build
2 WORKDIR /tmp
3
4 # dependencies
5 RUN apt-get update && DEBIAN_FRONTEND=noninteractive apt-get install -y \
6     git \
7     curl \
8     python-is-python3 \
9     python3 \
10    python3-pip \
11    python3-setuptools \
12    openssl \
13    libssl-dev \
14    build-essential
15
16 # configure intel sgx sdk
17 RUN curl -fsSL /usr/share/keyrings/intel-sgx-deb.asc https://download.01.org/intel-sgx/sgx_repo/
18     ubuntu/intel-sgx-deb.key && \
19     echo "deb [arch=amd64 signed-by=/usr/share/keyrings/intel-sgx-deb.asc] https://download.01.org/
20         intel-sgx/sgx_repo/ubuntu noble main" \
21     | tee /etc/apt/sources.list.d/intel-sgx.list
22
23 # install sgx sdk
24 RUN curl -O https://download.01.org/intel-sgx/sgx-dcap/1.22/linux/distro/ubuntu24.04-server/
25     sgx_linux_x64_sdk_2.25.100.3.bin
26 RUN chmod +x sgx_linux_x64_sdk_2.25.100.3.bin
27 RUN ./sgx_linux_x64_sdk_2.25.100.3.bin --prefix /opt/intel/
28
29 # get source code
30 RUN git clone --recurse-submodules --depth=1 --branch DCAP_1.21 https://github.com/intel/
31     SGXDataCenterAttestationPrimitives dcap
32
33 # build PCKCertSelection
34 WORKDIR /tmp/dcap/tools/PCKCertSelection
35
36 RUN . /opt/intel/sgxsdk/environment && \
37     make && \
38     mkdir /tmp/dcap/QuoteGeneration/pccs/lib && \
39     cp /tmp/dcap/tools/PCKCertSelection/out/libPCKCertSelection.so /tmp/dcap/QuoteGeneration/pccs/lib
40
41 RUN curl -sL https://deb.nodesource.com/setup_21.5 | bash -
42 RUN DEBIAN_FRONTEND=noninteractive apt-get -y install nodejs npm
```

```

39
40 WORKDIR /tmp/dcap/QuoteGeneration/pccs
41
42 RUN npm install
43
44 FROM node:21.5
45 WORKDIR /app
46
47 COPY --from=build /tmp/dcap/QuoteGeneration/pccs .
48
49 ENTRYPOINT [ "node", "pccs_server.js" ]

```

Il PCCS non è pensato per essere eseguito in un enclave, ma è una normale applicazione che espone delle API attraverso un tunnel TLS. Per eseguire un container con PCCS, bisogna creare una coppia di chiavi pubblica e privata da cui si genera un certificato:

```

openssl genrsa -out private.pem 2048
openssl req -new -key private.pem -out csr.pem
openssl x509 -req -days 365 -in csr.pem -signkey private.pem -out file.crt

```

Inoltre, bisogna creare un file di configurazione in formato json simile a quello mostrato in listato 7. Nell'esempio, viene utilizzato un database SQLite.

Listato 7: Esempio file di configurazione per PCCS

```

1 {
2   "HTTPS_PORT": 8081,
3   "hosts": "0.0.0.0",
4   "uri": "https://api.trustedservices.intel.com/sgx/certification/v4/",
5   "ApiKey": "",
6   "proxy": "",
7   "RefreshSchedule": "0 0 1 * * *",
8   "UserTokenHash": "",
9   "AdminTokenHash": "",
10  "CachingFillMode": "LAZY",
11  "OPENSSL_FIPS_MODE": false,
12  "LogLevel": "info",
13  "DB_CONFIG": "sqlite",
14  "sqlite": {
15    "database": "database",
16    "username": "",
17    "password": "",
18    "options": {
19      "host": "localhost",
20      "dialect": "sqlite",
21      "pool": {
22        "max": 5,
23        "min": 0,
24        "acquire": 30000,
25        "idle": 10000
26      },
27      "define": {
28        "freezeTableName": true
29      },
30      "logging": false,
31      "storage": "pckcache.db"
32    }
33  }
34 }

```

Bisogna inserire in *ApiKey* una delle due chiavi ricavate dalla piattaforma intel. Per *UserTokenHash* e *AdminTokenHash*, bisogna salvare l'hash con *sha512* sostituendo le effettive password per nel seguente snippet:

```

echo -n "user_password" | sha512sum | tr -d '[:space:]' -'
echo -n "admin_password" | sha512sum | tr -d '[:space:]' -'

```

Infine, è possibile eseguire l'applicazione (per semplificare le attività di deploy tutti i container eseguiti usano la rete dell'host):

```

docker run \
  -d \
  --network=host \
  -v $(pwd)/<path-to-config-file>:/app/config/default.json \
  -v $(pwd)/<path-to-certs-dir>:/app/ssl_key/ \
  pccs

```

Configurazione AESM service Per usare correttamente, la remote attestation bisogna configurare ed utilizzare le *Architectural Enclave (AE)* (come la *Quoting Enclave (QE)*, *Quoting Verification Enclave (QVE)*). Le *architectural enclave* sono enclave privilegiate fornite e firmate da Intel che devono essere gestite da attraverso l'*Architectural Enclave Service Manager (AESM)*. AESM è un'applicazione che espone i servizi delle architectural enclave attraverso una *unix socket*. Queste enclave possono comunicare con il PCCS e vengono configurate attraverso il file `/etc/sgx.default_qcnl.conf` (riportato di seguito). Se si vuole comunicare con un PCCS che usa certificati *self-signed*, bisogna impostare il flag `use_secure_cert` a *false*. AESM è installato sia nelle immagini di openenclave che gramine.

Listato 8: Configurazione remote attestation per le Architectural Enclaves

```

1 {
2   // *** ATTENTION : This file is in JSON format so the keys are case sensitive. Don't change them.
3
4   //PCCS server address
5   "pccs_url": "https://localhost:8081/sgx/certification/v4/"
6
7   // To accept insecure HTTPS certificate, set this option to false
8   ,"use_secure_cert": true
9
10  // You can use the Intel PCS or another PCCS to get quote verification collateral. Retrieval of
11    PCK
12  // Certificates will always use the PCCS described in pccs_url. When collateral_service is not
13    defined, both
14  // PCK Certs and verification collateral will be retrieved using pccs_url
15  //,"collateral_service": "https://api.trustedservices.intel.com/sgx/certification/v4/"
16
17  // Type of update to TCB Info. Possible value: early, standard. Default is standard.
18  // early indicates an early access to updated TCB Info provided as part of a TCB recovery event
19  // (commonly the day of public disclosure of the items in scope)
20  // standard indicates standard access to updated TCB Info provided as part of a TCB recovery
21    event
22  // (commonly approximately 6 weeks after public disclosure of the items in scope)
23  //,"tcb_update_type" : "standard"
24
25  // If you use a PCCS service to get the quote verification collateral, you can specify which PCCS
26    API version is to be used.
27  // The legacy 3.0 API will return CRLs in HEX encoded DER format and the sgx_ql_qve_collateral_t.
28    version will be set to 3.0, while
29  // the new 3.1 API will return raw DER format and the sgx_ql_qve_collateral_t.version will be set
30    to 3.1. The pccs_api_version
31  // setting is ignored if collateral_service is set to the Intel PCS. In this case, the
32    pccs_api_version is forced to be 3.1
33  // internally. Currently, only values of 3.0 and 3.1 are valid. Note, if you set this to 3.1,
34    the PCCS use to retrieve
35  // verification collateral must support the new 3.1 APIs.
36  //,"pccs_api_version": "3.1"
37
38  // Maximum retry times for QCNL. If RETRY is not defined or set to 0, no retry will be performed.
39  // It will first wait one second and then for all forthcoming retries it will double the waiting
40    time.
41  // By using retry_delay you disable this exponential backoff algorithm
42  //,"retry_times": 6
43
44  // Sleep this amount of seconds before each retry when a transfer has failed with a transient
45    error
46  //,"retry_delay": 10
47
48  // If local_pck_url is defined, the QCNL will try to retrieve PCK cert chain from local_pck_url
49    first,
50  // and failover to pccs_url as in legacy mode.

```

```

40 //,"local_pck_url": "http://localhost:8081/sgx/certification/v4/"
41
42 // If local_pck_url is not defined, set pck_cache_expire_hours to a none-zero value will enable
    local cache.
43 // The PCK certificates will be cached in memory and then to the disk drive.
44 // The local cache files will be sequentially searched in the following directories until located
    in one of them:
45 // Linux : $AZDCAP_CACHE, $XDG_CACHE_HOME, $HOME, $TMPDIR, /tmp/
46 // Windows : $AZDCAP_CACHE, $LOCALAPPDATA\..\..\LocalLow
47 // Please be aware that the environment variable pertains to the account executing the process
    that loads QPL,
48 // not the account used to log in. For instance, if QPL is loaded by QGS, then those environment
    variables relate to
49 // the "qgsd" account, which is the account that runs the QGS daemon.
50 // You can remove the local cache files either manually or by using the QPL API,
    sgx_qpl_clear_cache. If you opt to
51 // delete them manually, navigate to the aforementioned caching directories, find the folder
    named .dcap-qcml, and delete it.
52 // Restart the service after all cache folders were deleted. The same method applies to "
    verify_collateral_cache_expire_hours"
53 ,"pck_cache_expire_hours": 168
54
55 // To set cache expire time for quote verification collateral in hours
56 // See the above comment for pck_cache_expire_hours for more information on the local cache.
57 ,"verify_collateral_cache_expire_hours": 168
58
59 // When the "local_cache_only" parameter is set to true, the QPL/QCNL will exclusively use PCK
    certificates
60 // from local cache files and will not request any PCK certificates from service providers,
    whether local or remote.
61 // To ensure that the PCK cache is available for use, an administrator must pre-populate the
    cache folders with
62 // the appropriate cache files. To generate these cache files for specific platforms, the
    administrator can use
63 // the PCCS admin tool. Once the cache files are generated, the administrator must distribute
    them to each platform
64 // that requires provisioning.
65 ,"local_cache_only": false
66
67 // You can add custom request headers and parameters to the get certificate API.
68 // But the default PCCS implementation just ignores them.
69 //,"custom_request_options" : {
70 //   "get_cert" : {
71 //     "headers": {
72 //       "head1": "value1"
73 //     },
74 //     "params": {
75 //       "param1": "value1",
76 //       "param2": "value2"
77 //     }
78 //   }
79 // }
80 }

```

3.2 Applicazione base: helloworld

In questo esempio, viene illustrato un programma minimale sia con Gramine che con Open Enclave con lo scopo di evidenziarne le differenze. Lo scopo del programma è quello di stampare a video un messaggio di benvenuto.

3.2.1 Gramine

Come detto in §2.2, il codice sorgente dell'applicazione rimane invariato, ma bisogna focalizzarsi in sul manifest file, riportato di seguito.

Listato 9: Manifest per un semplice programma helloworld

```
1 # Copyright (C) 2023 Gramine contributors
2 # SPDX-License-Identifier: BSD-3-Clause
3
4 # Hello World manifest file example
5
6 libos.entrypoint = "/helloworld"
7 loader.log_level = "{{ log_level }}"
8
9 loader.env.LD_LIBRARY_PATH = "/lib"
10
11 fs.mounts = [
12   { path = "/lib", uri = "file:{{ gramine.runtimedir() }}" },
13   { path = "/helloworld", uri = "file:helloworld" },
14 ]
15
16 sgx.debug = true
17 sgx.edmm_enable = {{ 'true' if env.get('EDMM', '0') == '1' else 'false' }}
18
19 sgx.trusted_files = [
20   "file:helloworld",
21   "file:{{ gramine.runtimedir() }}/",
22 ]
```

La struttura della directory è riportata di seguito. La *toolchain* Gramine riconosce l'applicazione dal nome del file manifest dato in ingresso.

```
app/
├─ helloworld.c
├─ Makefile
└─ helloworld.manifest
```

Il file in listato 9 deve essere preprocessato e firmato con *sgx*. Questa procedura crea un file chiamato *helloworld.manifest* e la sua firma in *helloworld.sig*. In listato 10, è riportato il *Makefile* utilizzato per la compilazione dell'applicazione ed è possibile osservare che la generazione dell'eseguibile non dipende dalla parte. La regola per generare *helloworld.manifest* utilizza *gramine-manifest* che effettua un *preprocessing* di *helloworld.manifest.template*. Questo comando può definire variabili all'interno del manifest che poi saranno utilizzate con la sintassi di *jinja*. Ad esempio, *-Dlog_level* definisce la variabile *{log_level}*.

Listato 10: Makefile per applicazione helloworld

```
1 CFLAGS=-Wall -Wextra -O0
2 GRAMINE_LOG_LEVEL?=debug
3
4 run: build
5   gramine-sgx helloworld
6
7 .PHONY: build
8 build: helloworld.manifest.sgx helloworld.sig
9
10 helloworld.sig helloworld.manifest.sgx &: helloworld.manifest
11   gramine-sgx-sign \
12     --manifest $< \
13     --output $<.sgx
14   gramine-manifest-check $<.sgx
```

```

Gramine is starting. Parsing TOML manifest file, this may take some time...
(host_main.c:964:load_enclave) debug: Gramine parsed TOML manifest file successfully
(host_framework.c:237:create_enclave) debug: Enclave created:
(host_framework.c:238:create_enclave) debug:     base:           0x0000000000000000
(host_framework.c:239:create_enclave) debug:     size:           0x0000000010000000
(host_framework.c:240:create_enclave) debug:     misc_select:    0x00000000
(host_framework.c:241:create_enclave) debug:     attr.flags:     0x0000000000000007
(host_framework.c:242:create_enclave) debug:     attr.xfrm:      0x0000000000000007
(host_framework.c:243:create_enclave) debug:     ssa_frame_size: 4

```

Figura 8: Creazione dell'enclave con Gramine

```

15
16 helloworld.manifest: helloworld.manifest.template helloworld
17   gramine-manifest \
18     -Dlog_level=$(GRAMINE_LOG_LEVEL) \
19     $< $@
20   gramine-manifest-check $@
21
22 helloworld: helloworld.o
23
24 helloworld.o: helloworld.c
25
26 .PHONY: check
27 check: all
28   $(GRAMINE) helloworld > OUTPUT
29   echo "Hello, world" | diff OUTPUT -
30   @echo "[ Success ]"
31
32 .PHONY: clean
33 clean:
34   $(RM) *.sig *.manifest.sgx *.manifest helloworld.o helloworld OUTPUT
35
36 .PHONY: distclean
37 distclean: clean

```

Con *gramine-sgx-sign* viene calcolata una firma dell'enclave cifrata con una chiave ottenuta eseguendo il comando *gramine-sgx-gen-private-key* (che genera una chiave rsa a 3072 bit). In questa fase, il codice dell'enclave viene misurato insieme ad altri metadati per generare la *SIGSTRUCT*. La genuinità di questa firma può essere verificata attraverso la *remote attestation*. Successivamente, l'applicazione può essere eseguita con privilegi di amministratore usando il comando:

```
$ gramine-sgx helloworld
```

Dove *helloworld* è il nome del manifest file privato delle estensioni. Analizzando il log della compilazione, Gramine mostra tutte le misure che effettua per ogni pagina di memoria. All'esecuzione, l'enclave viene creata (Figura 8, i campi *attr* sono relativi alla struttura *SECS*, mentre con *size* viene specificata quella riportata nel *manifest*) e la sua memoria popolata pagina per pagina configurando i permessi di lettura/scrittura/esecuzione (RWX)(Figura 9).

```

(host_main.c:494:initialize_enclave) debug: Adding pages to SGX enclave, this may take some time...[]
(host_framework.c:323:add_pages_to_enclave) debug: Adding pages to enclave: 0xffffd000-0x10000000 [REG:R--] (manifest) measured
(host_framework.c:323:add_pages_to_enclave) debug: Adding pages to enclave: 0xffffd000-0xffffd000 [REG:RW-] (ssa) measured
(host_framework.c:323:add_pages_to_enclave) debug: Adding pages to enclave: 0xffff9000-0xffffd000 [TCS:---] (tcs) measured
(host_framework.c:323:add_pages_to_enclave) debug: Adding pages to enclave: 0xfffd5000-0xfffd9000 [REG:RW-] (tls) measured
(host_framework.c:323:add_pages_to_enclave) debug: Adding pages to enclave: 0xff95000-0xfffd5000 [REG:RW-] (stack) measured
(host_framework.c:323:add_pages_to_enclave) debug: Adding pages to enclave: 0xff55000-0xff95000 [REG:RW-] (stack) measured
(host_framework.c:323:add_pages_to_enclave) debug: Adding pages to enclave: 0xff15000-0xff55000 [REG:RW-] (stack) measured
(host_framework.c:323:add_pages_to_enclave) debug: Adding pages to enclave: 0xfed5000-0xff15000 [REG:RW-] (stack) measured
(host_framework.c:323:add_pages_to_enclave) debug: Adding pages to enclave: 0xfec5000-0xfed5000 [REG:RW-] (sig_stack) measured
(host_framework.c:323:add_pages_to_enclave) debug: Adding pages to enclave: 0xfec5000-0xfec5000 [REG:RW-] (sig_stack) measured
(host_framework.c:323:add_pages_to_enclave) debug: Adding pages to enclave: 0xfea5000-0xfec5000 [REG:RW-] (sig_stack) measured
(host_framework.c:323:add_pages_to_enclave) debug: Adding pages to enclave: 0xfe95000-0xfea5000 [REG:RW-] (sig_stack) measured
(host_framework.c:323:add_pages_to_enclave) debug: Adding pages to enclave: 0xfe3d000-0xfe8c000 [REG:R-X] (code) measured
(host_framework.c:323:add_pages_to_enclave) debug: Adding pages to enclave: 0xfe8c000-0xfe8f000 [REG:RW-] (data) measured
(host_framework.c:323:add_pages_to_enclave) debug: Adding pages to enclave: 0xfe8f000-0xfe95000 [REG:RW-] (bss) measured
(host_framework.c:323:add_pages_to_enclave) debug: Adding pages to enclave: 0x10000-0xfe3d000 [REG:RWX] (free)
(host_main.c:576:initialize_enclave) debug: Added all pages to SGX enclave
(host_framework.c:556:init_enclave) debug: Enclave initializing:
(host_framework.c:557:init_enclave) debug:     enclave id:  0x00000000ffff000
(host_framework.c:558:init_enclave) debug:     mr_enclave: 2a87821904b3a9f79d489582104f639bf5e357c994d25155a8e7ddc36cfa9b06
(host_framework.c:561:init_enclave) debug:     isv_prod_id: 0
(host_framework.c:562:init_enclave) debug:     isv_svn:   0

```

Figura 9: Mapping delle pagine

Dopo aver caricato le pagine, viene inizializzato il *libOs* che monta i file system ed esegue l'entrypoint.

3.2.2 OpenEnclave

L'applicazione base con Open Enclave risulta più complicata. Innanzitutto, bisogna specificare un file EDL come quello in listato 11. In particolare, questo definisce due funzioni (*host_helloworld* e *enclave_helloworld*) che l'enclave può eseguire in due differenti modalità e che devono essere implementate nell'applicazione finale.

Listato 11: EDL per applicazione helloworld

```

1 // Copyright (c) Open Enclave SDK contributors.
2 // Licensed under the MIT License.
3
4 enclave {
5     from "openenclave/edl/syscall.edl" import *;
6     from "platform.edl" import *;
7
8     trusted {
9         public void enclave_helloworld();
10    };
11
12    untrusted {
13        void host_helloworld();
14    };
15 };

```

Con OpenEnclave, c'è una separazione netta tra parte trusted e untrusted rispecchiata dalla struttura del progetto.

```

app/
├── host
│   ├── host.c
│   └── Makefile
├── enclave
│   ├── enc.c
│   └── Makefile
├── Makefile
└── helloworld.edl

```

Prima di sviluppare l'applicazione, è necessario creare le definizioni di tipi e importare le funzioni principali di OpenEnclave. Principalmente, queste definizioni servono a facilitare la gestione del ciclo di vita dell'enclave (Figura 2), gestendo le ECALL e OCALL con opportune EENTER e EEXIT. Questo può essere fatto con *oedger8r* che prende in ingresso il file *edl* e si occupa di definire i tipi giusti in base al *backend* specificato. Ad esempio, per generare il codice relativo all'enclave bisogna eseguire (stando nella cartella *enclave*):

```

oedger8r --search-path /opt/openenclave/include
--search-path /opt/openenclave/include/openenclave/edl/sgx
../helloworld.edl --trusted

```

Per generare la parte untrusted, bisogna eseguire (stando nella cartella *untrusted*):

```

oedger8r --search-path /opt/openenclave/include
--search-path /opt/openenclave/include/openenclave/edl/sgx
../helloworld.edl --untrusted

```

Il Makefile principale si limita a richiamare gli altri file nelle rispettive sezioni dell'applicazione.

Listato 12: Makefile helloworld con openenclave

```

1 .PHONY: all build clean run
2
3 OE_CRYPTOLIB := mbedtls
4 export OE_CRYPTOLIB
5
6 all: build
7
8 build:
9     $(MAKE) -C enclave
10    $(MAKE) -C host
11
12 clean:
13     $(MAKE) -C enclave clean
14     $(MAKE) -C host clean
15
16 run:
17     host/helloworld_host

```

Listato 12: Makefile helloworld con openenclave

Applicazione host L'applicazione host gestisce il ciclo di vita dell'enclave dalla sua creazione alla distruzione. Il codice (in cui sono stati omessi i controlli degli errori per brevità) è riportato in listato 13. Questo non cambia molto rispetto ad un'applicazione classica in *C*. La funzione *oe_create_helloworld_enclave* crea l'enclave e prende come primo parametro il path per l'enclave firmata definito nella macro *ENCLAVE_SIGNED_PATH*. Alla creazione dell'enclave viene verificata la firma e viene popolata la EPC. La funzione *enclave_helloworld* è una funzione nella parte trusted che esegue nel TEE (una ECALL) alla quale corrisponde un cambio di contesto del processore. Infine, con *oe_terminate_enclave*, si elimina l'enclave.

```

1 // Copyright (c) Open Enclave SDK contributors.
2 // Licensed under the MIT License.
3
4 #include <openenclave/host.h>
5 #include <stdio.h>
6
7 // Include the untrusted helloworld header that is generated
8 // during the build. This file is generated by calling the
9 // sdk tool oeedger8r against the helloworld.edl file.
10 #include "helloworld_u.h"
11
12 #define ENCLAVE_SIGNED_PATH "./enclave/enclave.signed"
13
14 // This is the function that the enclave will call back into to
15 // print a message.
16 void host_helloworld()
17 {
18     fprintf(stdout, "Enclave called into host to print: Hello World!\n");
19 }
20
21 int main()
22 {
23     oe_result_t result;
24     int ret = 1;
25     oe_enclave_t* enclave = NULL;
26
27     uint32_t flags = OE_ENCLAVE_FLAG_DEBUG;
28
29     // Create the enclave
30     result = oe_create_helloworld_enclave(
31         ENCLAVE_SIGNED_PATH, OE_ENCLAVE_TYPE_AUTO, flags, NULL, 0, &enclave);
32     if (result != OE_OK)
33     {
34         fprintf(
35             stderr,
36             "oe_create_helloworld_enclave(): result=%u (%s)\n",
37             result,
38             oe_result_str(result));
39         goto exit;
40     }
41
42     // Call into the enclave
43     result = enclave_helloworld(enclave);
44     if (result != OE_OK)
45     {
46         fprintf(
47             stderr,
48             "calling into enclave_helloworld failed: result=%u (%s)\n",
49             result,
50             oe_result_str(result));
51         goto exit;
52     }
53
54     ret = 0;
55
56 exit:
57     // Clean up the enclave if we created one
58     if (enclave)
59         oe_terminate_enclave(enclave);
60
61     return ret;
62 }

```

Questo componente è dotato di un Makefile, riportato in listato 14, in cui si compila normalmente l'applicazione dopo aver generato i flag del compilatore a partire da *open-enclave*. Il compilatore utilizzato è *clang*, ma sono supportati anche *cc* e *gcc*. Per cambiare compilatore, bisogna modificare il valore della variabile *CC* per ogni Makefile.

```

1 .PHONY: all build clean run
2
3 OE_CRYPTOLIB := mbedtls
4 export OE_CRYPTOLIB
5
6 all: build
7
8 build:
9     $(MAKE) -C enclave
10    $(MAKE) -C host
11
12 clean:
13     $(MAKE) -C enclave clean
14     $(MAKE) -C host clean
15
16 run:
17     host/helloworld_host

```

Applicazione enclave Per compilare l'enclave, bisogna seguire i seguenti passi racchiusi nel Makefile in listato 16.

- *compilare l'applicazione*: procedimento standard che avviene specificato nella regola *build*. Avviene analogamente alla parte host;
- *generare una coppia di chiavi*: con la regola *keys* si genera una chiave privata a 3072-bit da cui se ne deriva una pubblica;
- *firmare l'applicazione*: con *oesign* nella regola *sign*, l'applicazione viene firmata con la chiave privata generata in precedenza e un file di configurazione riportato in listato 15;

Il file di configurazione definisce alcuni parametri di configurazione per l'enclave ed è un semplice file di testo:

- *debug*: saranno stampate informazioni aggiuntive;
- *NumHeapPages*: massimo numero di pagine nell'heap per ogni thread nell'enclave;
- *NumStackPages*: massimo numero di pagine nello stack per ogni thread nell'enclave;
- *NumTCS*: massimo numero di *Thread Control Structure* consentito nell'enclave;
- *ProductID*: definisce un identificativo univoco per l'applicazione in modo tale da poter distinguere due programmi firmati dallo stesso *MRSIGNER*;
- *SecurityVersion*: protegge contro gli attacchi di tipo *rollback*;

Listato 15: Configurazione per la generazione dell'enclave

```

1 Debug=1
2 NumHeapPages=1024
3 NumStackPages=1024
4 NumTCS=1
5 ProductID=1
6 SecurityVersion=1

```

L'applicazione utilizza una versione minimale tls chiamata *mbedtls*, ma potrebbe essere usata *openssl* assegnando alla variabile *CFLAGS* il valore ottenuto dal comando:

```
pkg-config oeenclave-clang --variable=openssl_3flags
```

Listato 16: Makefile per la parte enclave

```

1 # Copyright (c) Open Enclave SDK contributors.
2 # Licensed under the MIT License.
3
4 COMPILER=clang

```

```

5 CC=$(COMPILER)-11
6
7 CRYPTO_LDFLAGS := $(shell pkg-config oeenclave-$(COMPILER) --variable=${OE_CRYPTO_LIB}libs)
8
9 CFLAGS=$(shell pkg-config oeenclave-$(COMPILER) --cflags)
10 LDFLAGS=$(shell pkg-config oeenclave-$(COMPILER) --libs)
11 INCDIR=$(shell pkg-config oeenclave-$(COMPILER) --variable=includedir)
12
13 all:
14     $(MAKE) build
15     $(MAKE) keys
16     $(MAKE) sign
17
18 build:
19     @ echo "Compilers used: $(CC), $(CXX)"
20     oeedger8r ../helloworld.edl --trusted \
21         --search-path $(INCDIR) \
22         --search-path $(INCDIR)/openenclave/edl/sgx
23     $(CC) -g -c $(CFLAGS) -DOE_API_VERSION=2 enc.c -o enc.o
24     $(CC) -g -c $(CFLAGS) -DOE_API_VERSION=2 helloworld_t.c -o helloworld_t.o
25     $(CC) -o enclave helloworld_t.o enc.o $(LDFLAGS) $(CRYPTO_LDFLAGS)
26
27 sign:
28     oesign sign -e enclave -c helloworld.conf -k private.pem
29
30 clean:
31     rm -f enc.o enclave enclave.signed private.pem public.pem helloworld_t.o helloworld_t.h
32         helloworld_t.c helloworld_args.h
33
34 keys:
35     openssl genrsa -out private.pem -3 3072
36     openssl rsa -in private.pem -pubout -out public.pem

```

Il codice *enc.c*, riportato listato 17, non contiene un *main*, ma solo la funzione da eseguire nel TEE. La funzione *host_helloworld* è una OCALL e comporta un'uscita dall'enclave per entrare nella parte untrusted ed è definita nella parte *host*.

Listato 17: Codice principale applicazione enclave

```

1 // Copyright (c) Open Enclave SDK contributors.
2 // Licensed under the MIT License.
3
4 #include <stdio.h>
5
6 // Include the trusted helloworld header that is generated
7 // during the build. This file is generated by calling the
8 // sdk tool oeedger8r against the helloworld.edl file.
9 #include "helloworld_t.h"
10
11 // This is the function that the host calls. It prints
12 // a message in the enclave before calling back out to
13 // the host to print a message from there too.
14 void enclave_helloworld()
15 {
16     // Print a message from the enclave. Note that this
17     // does not directly call fprintf, but calls into the
18     // host and calls fprintf from there. This is because
19     // the fprintf function is not part of the enclave
20     // as it requires support from the kernel.
21     fprintf(stdout, "Hello world from the enclave\n");
22
23     // Call back into the host
24     oe_result_t result = host_helloworld();
25     if (result != OE_OK)
26     {
27         fprintf(
28             stderr,
29             "Call to host_helloworld failed: result=%u (%s)\n",
30             result,
31             oe_result_str(result));
32     }
33 }

```

3.3 Remote attestation

Gli esempi che interagiscono con la parte di attestazione remota. Con Gramine, vengono esplorate le API di *secret provisioning* oltre a quelle di generazione di *report* e *quote*.

3.3.1 Gramine

Low-level interface In listato 18, è riportato un'applicazione demo in Python che interagisce con i file descritti in precedenza generando e stampando a video report e quote (la decodifica è stata fatta in base ai documenti in [2]).

Listato 18: Interazione con i file per l'attestazione remota

```

1  #!/usr/bin/env python3
2
3  import os
4  import sys
5
6  if not os.path.exists("/dev/attestation/quote"):
7      print(
8          "Cannot find '/dev/attestation/quote'; "
9          "are you running under SGX, with remote attestation enabled?"
10     )
11     sys.exit(1)
12
13  with open("/dev/attestation/attestation_type") as f:
14      print(f"Detected attestation type: {f.read()}")
15
16  with open("/dev/attestation/user_report_data", "wb") as f:
17      f.write(b"\0" * 64)
18
19  with open("/dev/attestation/quote", "rb") as f:
20      quote = f.read()
21
22  print(f"Extracted SGX quote with size = {len(quote)} and the following fields:")
23  print(f"  ATTRIBUTES.FLAGS: {quote[96:104].hex()} [ Debug bit: {quote[96] & 2 > 0} ]")
24  print(f"  ATTRIBUTES.XFRM: {quote[104:112].hex()}")
25  print(f"  MRENCLAVE: {quote[112:144].hex()}")
26  print(f"  MRSIGNER: {quote[176:208].hex()}")
27  print(f"  ISVPRODID: {quote[304:306].hex()}")
28  print(f"  ISVSVN: {quote[306:308].hex()}")
29  print(f"  REPORTDATA: {quote[368:400].hex()}")
30  print(f"  {quote[400:432].hex()}")
31
32  if not os.path.exists("/dev/attestation/report"):
33      print("Cannot find '/dev/attestation/report'; are you running under SGX?")
34      sys.exit(1)
35
36  with open("/dev/attestation/my_target_info", "rb") as f:
37      my_target_info = f.read()
38
39  with open("/dev/attestation/target_info", "wb") as f:
40      f.write(my_target_info)
41
42  with open("/dev/attestation/user_report_data", "wb") as f:
43      f.write(b"\0" * 64)
44
45  with open("/dev/attestation/report", "rb") as f:
46      report = f.read()
47
48  print(f"Generated SGX report with size = {len(report)} and the following fields:")
49  print(f"  ATTRIBUTES.FLAGS: {report[48:56].hex()} [ Debug bit: {report[48] & 2 > 0} ]")
50  print(f"  ATTRIBUTES.XFRM: {report[56:64].hex()}")
51  print(f"  MRENCLAVE: {report[64:96].hex()}")
52  print(f"  MRSIGNER: {report[128:160].hex()}")
53  print(f"  ISVPRODID: {report[256:258].hex()}")
54  print(f"  ISVSVN: {report[258:260].hex()}")
55  print(f"  REPORTDATA: {report[320:352].hex()}")

```

```
56 print(f"                                {report[352:384].hex()}")
```

Inoltre, bisogna modificare il manifest file aggiungendo tutti i file necessari ad eseguire un'istanza dell'interprete python e specificare il tipo di attestazione richiesta.

Listato 19: Manifest file per l'esecuzione di un interprete Python

```
1 libos.entrypoint = "{{ entrypoint }}"
2
3 loader.log_level = "{{ log_level }}"
4
5 loader.env.LD_LIBRARY_PATH = "/lib:/lib:{{ arch_libdir }}:/usr/{{ arch_libdir }}"
6
7 loader.insecure__use_cmdline_argv = true
8
9 sys.enable_sigterm_injection = true
10
11 fs.mounts = [
12   { path = "/lib", uri = "file:{{ gramine.runtimedir() }}" },
13   { path = "{{ arch_libdir }}", uri = "file:{{ arch_libdir }}" },
14   { path = "/usr/{{ arch_libdir }}", uri = "file:/usr/{{ arch_libdir }}" },
15   {% for path in python.get_sys_path(entrypoint) %}
16   { path = "{{ path }}", uri = "file:{{ path }}" },
17   {% endfor %}
18   { path = "{{ entrypoint }}", uri = "file:{{ entrypoint }}" },
19
20   { type = "tmpfs", path = "/tmp" },
21 ]
22
23 sys.stack.size = "2M"
24 sys.enable_extra_runtime_domain_names_conf = true
25
26 sgx.enclave_size = "1G"
27
28 sgx.remote_attestation = "dcap"
29
30 sgx.trusted_files = [
31   "file:{{ entrypoint }}",
32   "file:{{ gramine.runtimedir() }}/",
33   "file:{{ arch_libdir }}/",
34   "file:/usr/{{ arch_libdir }}/",
35   {% for path in python.get_sys_path(entrypoint) %}
36   "file:{{ path }}{{ '/' if path.is_dir() else '' }}" ,
37   {% endfor %}
38   "file:main.py",
39 ]
```

Il Makefile è analogo al caso precedente. Il risultato è riportato in Figura 10.

RA-TLS Gramine non effettua nessuna scelta sull'implementazione TLS da utilizzare (è possibile utilizzare sia OpenSSL che mbedtls). Il codice mostrato è tratto dall'esempio "ra-tls-mbedtls". Il server deve essere eseguito in un enclave; mentre il client può essere un'applicazione normale (può essere eseguito in un enclave). Una volta compilata l'applicazione con

```
make clean app dcap
```

bisogna ottenere le informazioni da passare al client per verificare la firma del server. Questo può essere fatto con il tool *gramine-sgx-sigstruct-view*.

Per esempio può essere eseguita una demo con

```
gramine-sgx ./server &
```

```
RA_TLS_MRENCLAVE=<MRENCLAVE of the server enclave> \
RA_TLS_MRSIGNER=<MRSIGNER of the server enclave> \
RA_TLS_ISV_PROD_ID=<ISV_PROD_ID of the server enclave> \
RA_TLS_ISV_SVN=<ISV_SVN of the server enclave> \
./client dcap
```

[illegible]

Figura 10: Lettura del Quote e del Report attraverso i file virtuali in Linux

Inoltre, bisogna ricordare di impostare le seguenti variabili d'ambiente se si eseguono le enclave in modalità debug o non si sono effettuati aggiornamenti al bios:

```
export RA_TLS_ALLOW_DEBUG_ENCLAVE_INSECURE=1
export RA_TLS_ALLOW_OUTDATED_TCB_INSECURE=1
export RA_TLS_ALLOW_HW_CONFIG_NEEDED=1
export RA_TLS_ALLOW_SW_HARDENING_NEEDED=1
```

Secret provisioning Il secret provisioning deve essere configurato nel manifest con le seguenti variabili:

- ***SECRET_PROVISION_CONSTRUCTOR***: se impostato a 1 effettua il provisioning prima di eseguire l'entrypoint in maniera automatica;
- ***SECRET_PROVISION_SET_KEY***: nome della chiave con cui verrà salvata in `/dev/attestation/keys` (default a `default`);
- ***SECRET_PROVISION_SERVERS***: lista dei server da contattare per avere il segreto (default a `localhost:4433`) o può essere fornita come argomento alla funzione `secret_provision_start()`

Il segreto può essere ricavato o in maniera automatica o utilizzando la funzione `secret_provision_get()`.

Esempio minimale Per avere un segreto prima che l'applicazione parta, bisogna aggiungere i seguenti campi al manifest file.

Listato 20: Caricamento di un segreto prima che l'applicazione parta

```
1 loader.env.LD_PRELOAD = "libsecret_prov_attest.so"
2 loader.env.SECRET_PROVISION_CONSTRUCTOR = "1"
3 loader.env.SECRET_PROVISION_SET_KEY = "default"
4 loader.env.SECRET_PROVISION_CA_CHAIN_PATH = "/ca.crt"
5 loader.env.SECRET_PROVISION_SERVERS = "localhost:4433"
6 sgx.remote_attestation = "dcap"
```

Il server è mostrato in listato 21.

Listato 21: Server che fornisce un segreto statico

```
1 #include <stdio.h>
2
3 #include "secret_prov.h"
4
5 #define PORT "4433"
6 #define SRV_CERT_PATH "./ssl/server.crt"
7 #define SRV_KEY_PATH "./ssl/server.key"
8
9 int main(void) {
10     uint8_t secret[] = "A_SIMPLE_SECRET";
11     puts("--- Starting the Secret Provisioning server on port " PORT " ---");
12     int ret = secret_provision_start_server(
13         secret, sizeof(secret), PORT, SRV_CERT_PATH, SRV_KEY_PATH, NULL, NULL);
14     if (ret < 0) {
15         fprintf(stderr, "[error] secret_provision_start_server() returned %d\n",
16             ret);
17         return 1;
18     }
19     return 0;
20 }
```

Il client riceve il segreto come variabile d'ambiente.

Listato 22: Client che riceve un segreto automaticamente all'avvio

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5     char *secret = getenv("SECRET_PROVISION_SECRET_STRING");
```

```

6  if (secret)
7      printf("--- Received secret = '%s' ---\n", secret);
8  else
9      printf("--- Did not receive any secret! ---\n");
10
11  return 0;
12 }

```

Per eseguire l'esempio, bisogna impostare a *false* il flag *use_secure_cert* nel file */etc/sgx.default-qcnl.conf* e rieseguire AESM con */restart-aesm.sh*. Compilare client e server con:

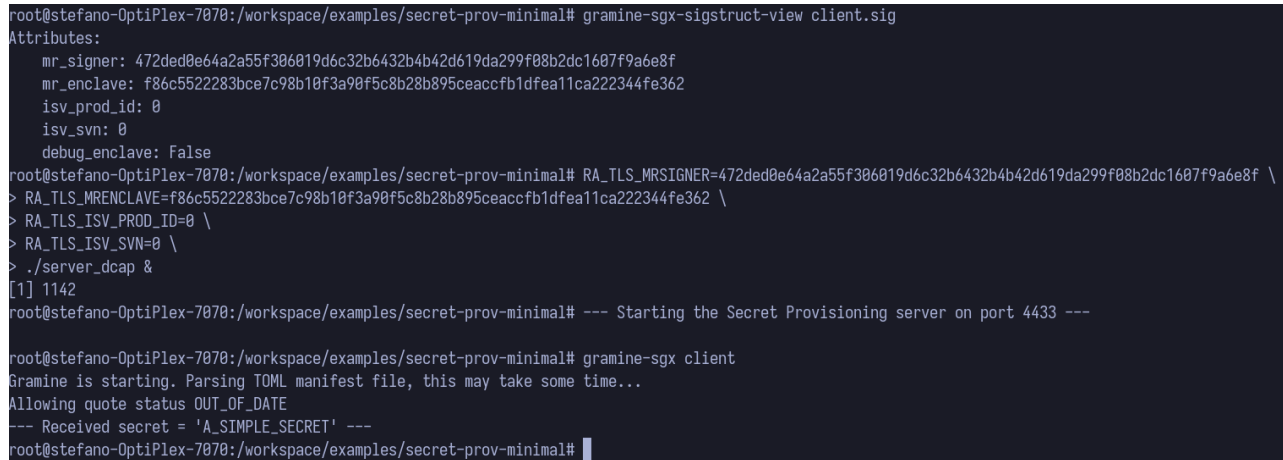
```
make clean app dcap
```

Successivamente, bisogna ricavare i valori di firma dell'enclave dal file *client.sig* con *gramine-sgx-sigstruct-view*, eseguire il server e l'enclave con *gramine-sgx* (Figura 11). Se si verificano errori quali ***TCB Outdated***, bisogna esportare le variabili d'ambiente:

```

export RA_TLS_ALLOW_DEBUG_ENCLAVE_INSECURE=1
export RA_TLS_ALLOW_OUTDATED_TCB_INSECURE=1
export RA_TLS_ALLOW_HW_CONFIG_NEEDED=1
export RA_TLS_ALLOW_SW_HARDENING_NEEDED=1

```



```

root@stefano-OptiPlex-7070:/workspace/examples/secret-prov-minimal# gramine-sgx-sigstruct-view client.sig
Attributes:
  mr_signer: 472ded0e64a2a55f306019d6c32b6432b4b42d619da299f08b2dc1607f9a6e8f
  mr_enclave: f86c5522283bce7c98b10f3a90f5c8b28b895ceaccfb1dfea11ca222344fe362
  isv_prod_id: 0
  isv_svn: 0
  debug_enclave: False
root@stefano-OptiPlex-7070:/workspace/examples/secret-prov-minimal# RA_TLS_MRSIGNER=472ded0e64a2a55f306019d6c32b6432b4b42d619da299f08b2dc1607f9a6e8f \
> RA_TLS_MRENCLAVE=f86c5522283bce7c98b10f3a90f5c8b28b895ceaccfb1dfea11ca222344fe362 \
> RA_TLS_ISV_PROD_ID=0 \
> RA_TLS_ISV_SVN=0 \
> ./server_dcap &
> ./server_dcap &
[1] 1142
root@stefano-OptiPlex-7070:/workspace/examples/secret-prov-minimal# --- Starting the Secret Provisioning server on port 4433 ---

root@stefano-OptiPlex-7070:/workspace/examples/secret-prov-minimal# gramine-sgx client
Gramine is starting. Parsing TOML manifest file, this may take some time...
Allowing quote status OUT_OF_DATE
--- Received secret = 'A_SIMPLE_SECRET' ---
root@stefano-OptiPlex-7070:/workspace/examples/secret-prov-minimal#

```

Figura 11: Stampa dei parametri di firma dell'enclave, avvio del server e condivisione di un segreto statico

Il server può specificare una *callback* per comunicare con il client per creare un protocollo più complicato. Come mostrato in listato 23, il server può interagire con il client attraverso *secret_provision_write* e *secret_provision_read*.

Listato 23: Interazione continua con il client

```

1  static int communicate_with_client_callback(struct ra_tls_ctx* ctx) {
2      int ret;
3      ret = secret_provision_read(ctx, buf, sizeof(buf));
4
5      /* if we reached this callback, the first secret was sent successfully */
6      printf("--- Sent secret1 ---\n");
7      ret = secret_provision_write(ctx, (uint8_t*)SECOND_SECRET, sizeof(SECOND_SECRET));
8      if (ret < 0) {
9          fprintf(stderr, "[error] secret_provision_write() returned %d\n", ret);
10         return -EINVAL;
11     }
12
13     printf("--- Sent secret2 = '%s' ---\n", SECOND_SECRET);
14     return 0;
15 }
16 int main(void) {
17     puts("--- Starting the Secret Provisioning server on port " PORT " ---");

```



```
18     ret = secret_provision_start_server((uint8_t*)FIRST_SECRET, sizeof(FIRST_SECRET),
19                                       PORT, SRV_CERT_PATH, SRV_KEY_PATH,
20                                       NULL,
21                                       communicate_with_client_callback);
22     if (ret < 0) {
23         fprintf(stderr, "[error] secret_provision_start_server() returned %d\n", ret);
24         return 1;
25     }
26     return 0;
27 }
```

3.4 Applicazione avanzata: sqlite3

In questo esempio, si vuole mostrare come creare un'applicazione che utilizza Sqlite e salvare in maniera cifrata i dati sul disco.

4 Sviluppi futuri: Intel Trusted Domains Extensions (TDX)

Lo sviluppo di SGX ha subito un rallentamento in favore di tecnologie che sfruttano la virtualizzazione. Intel nel 2021 ha proposto *Trusted Domain Extensions* (TDX) che introduce un'altra modalità all'interno della CPU: *Secure Arbitration Mode (SEAM)* (Figura 12). TDX è un'implementazione di TEE basata su macchine virtuali come *CoVE* e AMD-SEV.

Intel TDX mette insieme diverse tecnologie consolidate per offrire un'implementazione TEE con supporto per la virtualizzazione[5] quali:

- Intel VT-x: tecnologia che supporta la virtualizzazione in hardware unitamente a EPT;
- Intel SGX: implementazione di TEE in *user space*;
- Intel MKTME/TME: tecnologia per cifrare la memoria con diverse chiavi il cui identificativo viene codificato nei bit più alti dell'indirizzo;

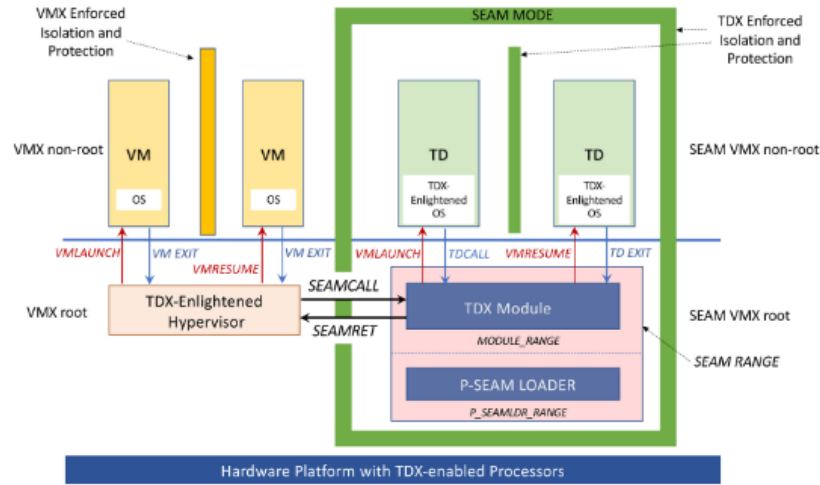


Figura 12: Architettura di TDX

4.1 Virtualizzazione

Il concetto principale di TDX è il *Trusted Domain (TD)* di cui viene garantita la confidenzialità e l'integrità della memoria e dello stato delle vCPU. TDX utilizza VT per garantire la separazione i *trusted domains*. In particolare, il modulo TDX e i TD eseguono in un nuovo stato all'interno degli stati *VMX Root* e *Non root*. Nelle prime versioni di TDX non è disponibile la virtualizzazione innestata e c'è un limitato supporto alla condivisione di memoria. Per ogni TD, sono create due EPT: una privata e cifrata e un'altra in chiaro.

4.2 Memory protection

La protezione della memoria viene fatta utilizzando la *Multiple Keys Total Memory Encryption (MKTME)*. La cifratura della memoria viene affidata completamente al modulo TDX che gestisce le chiavi di ogni singolo TD. MKTME prevede la cifratura della memoria con una chiave identificata da un *Host Key Identifier (HKID)* codificata all'interno dell'indirizzo fisico e memorizzata in una tabella detta *Key Encryption Table (KET)*. Lo spazio delle chiavi è diviso in:

- HKID private: chiavi assegnate a singoli TD;
- HKID shared: chiavi utilizzate per la memoria condivisa;

4.3 Attestazione

In TDX, l'attestazione è fortemente collegata alle *architectural enclaves* presentate in precedenza. TDX effettua un'attestazione locale con la Quoting Enclave e utilizza i servizi spiegati in precedenza per effettuare l'attestazione DCAP come in SGX. Il Quote di TDX include un certificato rilasciato da Intel.

Riferimenti

- [1] Gramine documentation. <https://gramine.readthedocs.io/en/v1.8/>.
- [2] Intel SGX, Documentation for Linux SDK . <https://download.01.org/intel-sgx/sgx-linux/2.25/docs/>.
- [3] Open Enclave repository . <https://github.com/openenclave/openenclave>.
- [4] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, Santa Clara, CA, jul 2017. USENIX Association.
- [5] Pau-Chen Cheng, Wojciech Ozga, Enriqueillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel tdx demystified: A top-down approach. *ACM Comput. Surv.*, 56(9), apr 2024.
- [6] Victor Costan and Srinivas Devadas. Intel sgx explained. Cryptology ePrint Archive, Paper 2016/086, 2016.
- [7] Intel. Tutorial DCAP setup. <https://www.intel.com/content/www/us/en/developer/articles/guide/intel-software-guard-extensions-data-center-attestation-primitives-quick-install-guide.html>, 2018.
- [8] Intel. DCAP Documentation References. <https://download.01.org/intel-sgx/sgx-dcap/1.22/linux/docs/>, 2024.
- [9] Intel. DCAP Source Code. <https://github.com/intel/SGXDataCenterAttestationPrimitives>, 2024.
- [10] Intel. Gramine Source Code. <https://github.com/gramineproject/gramine/>, 2024.
- [11] Intel. Linux-sgx Source Code. <https://github.com/intel/linux-sgx/>, 2024.
- [12] Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Trusted execution environments: Properties, applications, and challenges. *IEEE Security & Privacy*, 18(2):56–60, 2020.
- [13] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, HASP ’16, New York, NY, USA, 2016. Association for Computing Machinery.