POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea

Protezione automatica del codice tramite enclavi SGX

Relatori
Prof. Antonio Lioy
Ing. Leonardo Regano
Ing. Daniele Canavese

Candidato
Matteo Costa

Anno accademico 2019-2020

Alla mia famiglia
♡ *Al mio amore*

Indice

Capitolo 1

Introduzione

La crescente diffusione dei dispositivi informatici e la continua connessione ad Internet facilitano gli attaccanti aprendo nuove vie d'accesso ai dispositivi. L'attaccante che ha accesso al dispositivo agisce con lo scopo di creare un danno economico, ad esempio, rubando informazioni riservate e modellando i dati a piacimento.

Per questo motivo, le applicazioni che manipolano informazioni sensibili devono proteggere i propri dati, per esempio cifrando i file prima di memorizzarli su disco e utilizzando dei protocolli di comunicazione cifrati quando si transitano i dati sulla rete. Tuttavia, queste barriere non sono sufficienti quando il sistema operativo ed i processi eseguiti con privilegi elevati non sono fidati. Infatti, i processi con alti privilegi possono accedere illimitatamente alle risorse del sistema, influendo quindi anche sulle risorse già utilizzate da un'altra applicazione. Questo potere può essere utilizzato per leggere e scrivere la memoria principale del sistema, dando così la possibilità di gestire completamente i dati delle applicazioni in memoria. A questo scopo si sono progettate diverse soluzioni per la protezione del codice e dei dati in memoria. Una prima soluzione è quella di rendere il codice ed i dati quasi incomprensibili, modificandone la struttura ed utilizzando delle tecniche di offuscamento. In questo modo, un attaccante che utilizza malware o debugger per accedere alle risorse dell'applicazione, non riesce direttamente a comprendere il significato del codice e dei dati, poiché questo non è memorizzato esplicitamente in memoria. Un'alternativa alle tecniche di protezione citate è la tecnologia Intel SGX[1]. Questa è basata sull'utilizzo di servizi forniti dal processore che consentono una protezione del codice e dei dati in regioni di memoria cifrate, chiamate enclavi. Queste regioni protette sono adatte a contenere sezioni di codice e dati sensibili che non possono essere letti e modificati dai processi eseguiti con privilegi elevati. Le enclavi SGX sono memorizzate nella memoria principale del sistema e forniscono le proprietà di integrità e di riservatezza alle informazioni contenute.

Di conseguenza, un processo con privilegio elevato può leggere la regione di memoria ma non potrà comprenderla poiché sprovvisto della chiave per decifrarla. Allo stesso modo, un processo dell'attaccante può modificare parti della regione protetta, provocando però la terminazione del programma. Infatti, la tecnologia

[1] https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html

Intel SGX si accorge della variazione non autorizzata ed impedisce il proseguimento dell'applicazione. Intel Software Guard Extensions (SGX) è in grado di proteggere il codice scritto in linguaggio C e C++. Tuttavia, per l'utilizzo della tecnologia sono necessari l'esecuzione su un processore compatibile[2] ed un'organizzazione specifica del codice. Quest'ultimo deve essere diviso in due parti nominate trusted ed untrusted. La prima è l'insieme di codice e di dati da proteggere mentre la seconda è la parte non sensibile dell'applicazione. Il codice da proteggere deve essere inserito nel file sorgente dell'enclave e quest'ultima deve essere definita attraverso un linguaggio proprietario di Intel SGX che utilizza il formato nominato `EDL`. Inoltre, la parte untrusted dell'applicazione deve generare in memoria l'enclave, coordinare la comunicazione tra il codice trusted ed untrusted, e gestire i possibili errori di allocazione e di comunicazione.

Questi adempimenti sono modifiche da effettuare nel codice sorgente dell'applicazione prima della sua compilazione. Il programmatore che intende utilizzate la tecnologia Intel SGX per proteggere l'applicazione, deve inizialmente farsi carico della completa comprensione di questa tecnologia. Poi, questo è responsabile della decisione della linea di divisione tra codice trusted ed untrusted. Inoltre, il programmatore si fa carico di modificare i file sorgenti C/C++ in modo tale da rispettare la struttura richiesta da Intel SGX. Infine, questo deve generare il codice di gestione, il codice di esecuzione e il codice di definizione dell'enclave.

Tutte queste operazioni risultano complesse ed onerose da eseguire manualmente, soprattutto se si applicano su progetti di grandi dimensioni. Quindi, è desiderabile uno strumento in grado di effettuare la conversione automatica di un'applicazione preesistente per l'utilizzo della tecnologia Intel SGX.

La modifica del codice necessaria per applicare le tecniche di offuscamento, può essere svolta sia manualmente che automaticamente (e.g. attraverso l'utilizzo di strumenti appositi come ESP [1], Diablo[3] e Tigress[4]). Tuttavia, la prima opzione è sconsigliata, poiché anche piccole modifiche effettuate manualmente al codice sorgente possono introdurre nuovi bug o persino nuove vulnerabilità. Allo stesso modo, le modifiche necessarie all'utilizzo della tecnologia SGX, come la divisione del codice trusted ed untrusted, la generazione del codice aggiuntivo per la gestione, l'esecuzione e la definizione dell'enclave possono essere automatizzate.

A questo scopo si è sviluppato un framework di conversione abile nel modificare il codice sorgente per renderlo protetto con SGX. L'utente che vuole utilizzare il framework per rendere la propria applicazione protetta, annota il sorgente con delle parole chiave che non alterano in nessun modo il funzionamento ed il flusso del codice. Queste annotazioni sono utilizzate per segnalare al framework le funzioni e i dati che sono strettamente connessi alle informazioni sensibili. Poi, l'utente configura lo strumento selezionando i file del codice da convertire ed avvia la conversione.

La prima operazione indispensabile eseguita dal framework è analizzare il codice sorgente e comprenderlo interamente. In generale, l'attività di analisi può essere

[2] https://www.intel.it/content/www/it/it/support/articles/000028173/processors.html

[3] https://diablo.elis.ugent.be/node/1

[4] https://tigress.wtf/introduction.html

svolta sia staticamente, senza eseguire il codice, sia dinamicamente, analizzando l'esecuzione del codice a runtime. Tuttavia, l'utilizzo dell'analisi dinamica non è compatibile con lo scopo del framework, poiché si necessita di modificare il sorgente a livello di linguaggio C/C++ e non di codice macchina. Quindi, l'analisi statica del codice può essere eseguita sia sintatticamente, valutando i costrutti del codice senza la loro comprensione, sia semanticamente, analizzando il significato dei predicati e delle operazioni svolte. Il framework utilizza entrambi i metodi di analisi per comprendere le componenti del software e le relazioni tra le entità, e generare un modello di dati rappresentativo dell'applicazione.

Poi, il framework controlla la compatibilità del pacchetto di codice con le limitazioni di Intel SGX, come ad esempio l'insieme di funzioni utilizzate nell'ambiente sicuro, il tipo dei loro parametri, il metodo di comunicazione degli argomenti alle funzioni sicure e la modalità di invocazione delle funzioni protette. Controllata la compatibilità, il framework calcola la divisione tra la parte trusted ed untrusted dell'applicazione. Questa analisi utilizza sia il modello che rappresenta le entità e le dipendenze software generate dall'analisi statica, sia le informazioni delle annotazioni inserite dall'utente, per dedurre quali funzioni, strutture dati, enumerazioni, unioni e definizioni appartengono all'insieme di codice trusted. Poi, le parti sensibili di codice vengono spostate nel sorgente dell'enclave che, per definizione, non può essere direttamente invocato dall'esterno. Quindi, il framework genera delle nuove funzioni di comunicazione e gestione degli errori che permettono di connettere il codice trusted dell'enclave con il codice untrusted. In questo modo, le funzioni ed i dati spostati nell'enclave ereditano le proprietà offerte da Intel SGX, senza che il funzionamento ad alto livello del codice venga alterato. Infine, il framework genera sia il codice di definizione dell'enclave necessario alla compilazione con Intel SGX sia il codice di gestione dell'enclave che si occupa di gestirne l'allocazione e gli errori.

Nei capitoli seguenti si descrivono la tecnologia utilizzata, le analisi effettuate, la struttura ed il funzionamento del framework. In particolare, il Capitolo 2 introduce la tecnologia Intel SGX, in grado di proteggere parti di codice attraverso delle strutture protette in memoria, chiamate enclavi. Inoltre, si approfondiscono gli aspetti interni, le proprietà offerte, le modalità di utilizzo, le limitazioni, i possibili attacchi ed alternative alla tecnologia SGX. Il Capitolo 3 descrive lo stato dell'arte dell'analisi del codice. Il framework esegue un'analisi del codice che gli permette di individuare le entità fondamentali (come funzioni, strutture dati ed enumerazioni) ed il flusso del codice. Si introduce quindi il concetto di analisi statica e dinamica del codice, se ne presentano gli utilizzi ed i limiti, e si introducono strumenti utili a tali scopi. Il Capitolo 4 descrive nel dettaglio la struttura ed i componenti del framework, analizzando il funzionamento e la collaborazione tra i singoli moduli interni. Il funzionamento del framework ed il suo flusso di lavoro, a partire dalle annotazioni inserite dall'utente alla compilazione del binario protetto, vengono descritte nel Capitolo 5. Successivamente, nel Capitolo 6 si descrivono entrambi i tipi di test eseguiti sul framework e sulle applicazioni convertite. In particolare, si analizzano le prestazioni di conversione del framework al variare della complessità e della dimensione del codice. Quindi, si analizzano i rendimenti delle applicazioni convertite e di conseguenza l'impatto della tecnologia e del framework sull'esecuzione del programma. Infine, il Capitolo 7 descrive altre tecnologie di protezione del

codice, applicabili con gli strumenti di protezione automatica, come Diablo, Tigress ed ESP. In ultimo, si analizzano gli utilizzi pratici della tecnologia SGX, come la protezione dei contenitori Docker e delle architetture basate sui microservizi.

Capitolo 2

Tecnologia SGX

Uno dei compiti del sistema operativo è applicare una politica di sicurezza (come `page faults`[1] e `W^X`[2]) in modo tale da disciplinare l'accesso ai dati secondo applicazioni e utenti. Infatti, questo garantisce che un utente possa accedere solo ai propri file e un'applicazione possa leggere solo i propri dati in memoria. Inoltre, questo gestisce i privilegi di amministratore che permettono ad un'applicazione o ad un utente di accedere direttamente alle risorse del sistema operativo. Per garantire ulteriori proprietà di sicurezza, spesso le applicazioni adottano garanzie aggiuntive, come la crittografia dei dati, per assicurare che i dati memorizzati in archivio o inviati tramite rete non siano accessibili a terzi anche se il sistema operativo e l'hardware siano stati compromessi. Nonostante le protezioni offerte dal sistema operativo ed aggiunte dalle applicazioni, esiste ancora una significativa vulnerabilità: un'applicazione non ha alcuna protezione dai processi in esecuzione con privilegi più elevati, incluso il sistema operativo stesso. Quindi un processo, con privilegi elevati può superare le barriere del sistema operativo e leggere eventualmente dati non cifrati di un'applicazione in memoria. Quindi, un malware sofisticato che ottiene i privilegi di amministratore, quindi con accesso illimitato a tutte le risorse del sistema, può colpire gli schemi di protezione di un'applicazione per estrarre le chiavi crittografiche e persino i dati segreti stessi direttamente dalla memoria[3]. Intel SGX è un modello di progettazione ed esecuzione del codice che permette di proteggere il software dalla vulnerabilità citata e quindi stabilisce una barriera di protezione tra i software protetti e i programmi eseguiti con privilegi elevati.

In questo capitolo si analizzano quindi, gli aspetti principali dell'architettura Intel SGX, le funzionalità di sicurezza offerte, le modalità di impiego ed i suoi limiti. Inoltre si definiscono le proprietà del codice protetto con questo strumento e si descrivono alcuni possibili attacchi. Inoltre, si definiscono le fasi di sviluppo, il design del software ed il flusso di esecuzione dei programmi progettati per l'esecuzione

[1] http://www.inf.ed.ac.uk/teaching/courses/os/slides/11-virtualmem.pdf

[2] http://www.openbsd.org/33.html

[3] https://software.intel.com/en-us/articles/intel-software-guard-extensions-tutorial-part-1-foundation

protetta. Infine si pone attenzione sulle alternative attualmente disponibili per proteggere l'esecuzione del codice e si confrontano i principali aspetti, similitudini e differenze con Intel SGX.

2.1 Analisi della tecnologia SGX

Intel Software Guard Extensions (SGX)[4] è una soluzione di protezione del codice, alternativa all'architettura Intel Trusted Execution Technology (TXT) spiegata in Sezione 2.3.3. Questa nuova tecnica nasce a partire dal 2015 con la sesta generazione di Intel Core Skylake con lo scopo di proteggere il codice in esecuzione dall'ambiente in cui viene eseguito. Questa architettura si basa sull'introduzione nel set di istruzioni Intel x86-64[5] di un insieme di istruzioni speciali le quali consentono al codice di allocare regioni private di memoria, chiamate *enclavi*. Questi spazi sono protetti da accesso anche dai processi in esecuzione a livelli di privilegio più elevati[6]. Il codice ed i dati inseriti dentro all'enclave in esecuzione sono protetti per preservarne l'integrità e la confidenzialità, e di conseguenza non sono comprensibili nè modificabili dal sistema operativo, dai driver, dal BIOS e di conseguenza da ogni attacco remoto[7]. Con integrità si intende la protezione da modifiche improprie o distruzione delle informazioni e include la garanzia di non ripudio e autenticità delle informazioni. Invece, con confidenzialità si intende preservare le restrizioni autorizzate sull'accesso alle informazioni e divulgazione, compresi i mezzi per la protezione della privacy e informazioni proprietarie [2].

L'utilizzo delle regioni di memoria protetta trova impiego solo per quelle sezioni di codice che devono lavorare con i segreti di un'applicazione come informazioni sensibili, credenziali degli account (password e informazioni personali), informazioni bancarie, chiavi di crittografia e cartelle cliniche. Non è consigliato eseguire in memoria protetta codice che non necessita delle proprietà fornite dall'enclave dato il peggioramento delle prestazioni, come spiegato in Sezione 2.1.6.

Il nuovo modello SGX non protegge completamente l'applicazione dagli attacchi ma si limita a ridurre la superficie vulnerabile di un'applicazione. Infatti, l'uso di regioni protette limita la superficie di attacco tra hardware e il sistema operativo (o tra hardware e Virtual Machine Manager (VMM)[8] in caso di sistemi virtualizzati), e all'ingresso ed uscita dell'enclavi. Finché l'esecuzione avviene in enclave, i dati memorizzati ed il codice eseguito non sono comprensibili né modificabili dagli altri processi. Quando i dati vengono esportati fuori dall'enclave non saranno più

[4] https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html

[5] https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html#nine-volume

[6] https://software.intel.com/en-us/blogs/2013/09/26/protecting-application-secrets-with-intel-sgx

[7] https://software.intel.com/sites/default/files/332680-002.pdf

[8] Il Virtual Machine Manager è lo strato software che si interfaccia tra hardware e il sistema operativo. Il VMM è in grado di avviare, controllare ed arrestare le macchine virtuali che vengono eseguite sul sistema.

protetti con le proprietà di integrità e riservatezza e di conseguenza saranno visibili nuovamente dai processi con esecuzione più privilegiata, come il sistema operativo. La Figura 2.1 mostra la differenza tra le superfici di attacco con e senza l'aiuto delle enclavi Intel SGX.

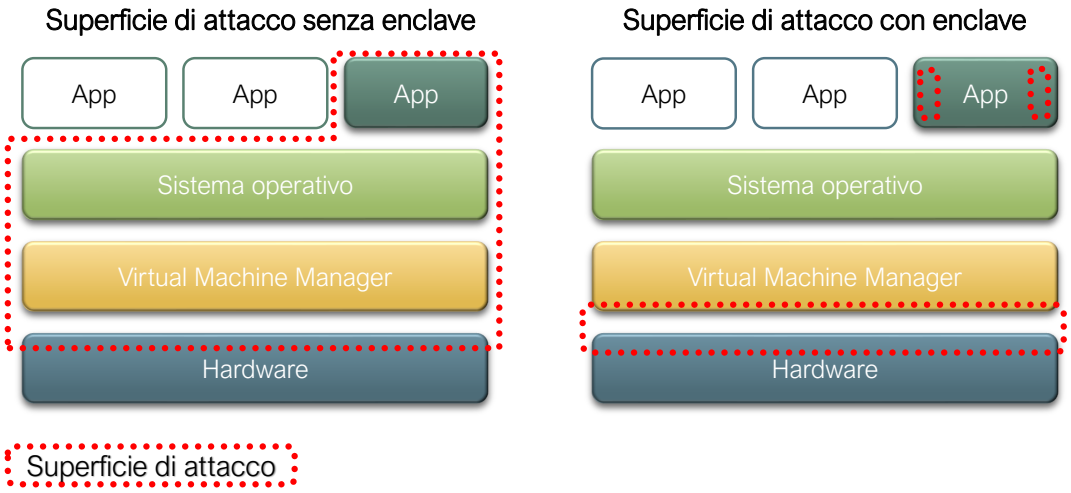


Figura 2.1: Superficie di attacco con e senza l'esecuzione protetta in enclave.

Senza l'esecuzione in enclave, la superficie di attacco del codice risulta molto ampia. Infatti, i processi con privilegi elevati possono leggere e modificare la memoria dell'applicazione. Inoltre, un'altra superficie di attacco è l'hardware non protetto. Infatti, è possibile virtualizzare l'hardware ed eseguire l'applicazione in una macchina virtuale in modo da leggere comunque il contenuto della memoria. In casi più avanzati, si potrebbero utilizzare gli attacchi `RAMBleed Attack` e `DMA Attack` per accedere ai dati memorizzati in memoria RAM. Il primo sfrutta l'effetto Rowhammer[9], l'errata inversione dei bit delle celle della memoria DRAM quando si eseguono accessi ripetuti alle celle vicine, per leggere le informazioni di altri processi[10]. Il secondo attacco[11] utilizza l'accesso diretto alla macchina per connettere un interfaccia PCI al DMA Controller ed accedere di conseguenza alla memoria RAM senza controllo del sistema operativo. Con l'uso di Intel SGX vengono offerte queste protezioni da noti attacchi hardware e software[12]:

1. la memoria dell'enclave è protetta da integrità e riservatezza fornite dalla CPU e non è comprensibile nè modificabile dall'esterno dell'enclave indipendentemente dal livello di privilegio corrente e dalla modalità CPU;

[9] https://arstechnica.com/information-technology/2019/06/researchers-use-rowhammer-bitflips-to-steal-2048-bit-crypto-key/

[10] https://rambleed.com/

[11] https://www.synacktiv.com/posts/pentest/practical-dma-attack-on-windows-10.html

[12] https://software.intel.com/en-us/articles/intel-software-guard-extensions-tutorial-part-1-foundation

2. le enclavi compilate in modalità produzione (descritte in Sezione 2.2.5) non possono essere sottoposte a debug software o hardware;

3. l'accesso all'enclave è gestito da nuove istruzioni del processore che eseguono controlli ed impediscono l'accesso all'esecuzione protetta tramite chiamate di funzioni classiche, istruzioni di jump, manipolazioni dei registri o manipolazioni dello stack;

4. la memoria dell'enclave è cifrata, quindi modificando i dati in memoria o collegando i moduli DRAM a un altro sistema si otterranno solo dati incomprensibili;

5. la chiave di cifratura della memoria viene generata ad ogni ciclo di accensione (e.g. all'avvio del sistema, alla ripresa dagli stati di sospensione e ibernazione) e non è accessibile in alcun modo via software[13], poiché memorizzata nella CPU;

6. i dati memorizzati all'interno di un'enclave sono accessibili solo da codice appartenente alla stessa enclave.

2.1.1 Memoria protetta dall'enclave

L'architettura Intel SGX permette di allocare più enclavi per la singola applicazione. Questa caratteristica permette uno sviluppo più semplice in caso di applicazioni molto grandi e di utilizzo codici standard già scritti. Infatti, è possibile sfruttare questa caratteristica per dividere le parti sensibili di un'applicazione in diverse enclavi indipendenti e rendere il codice modulare. In questo modo, è possibile riutilizzare lo stesso codice per più applicazioni e velocizzare lo sviluppo riducendo il costo di scrittura e debug di nuovo codice. Inoltre, enclavi più piccole consentono sia un'allocazione mirata per la singola funzionalità richiesta al momento sia una minor superficie di attacco dall'esterno. Infatti, limitando il numero di accessi in ingresso ed uscita dall'enclave si avrà una porzione di superficie di attacco ridotta e di conseguenza un codice più affidabile. In ultimo, un approccio di progettazione modulare del codice come la suddivisione delle responsabilità e la divisione dei dati, rendono il codice più robusto e maggiormente comprensibile dallo sviluppatore.

I dati delle parti di codice protetti dall'enclave sono scritti nella memoria principale che non fornisce alcuna protezione dal codice potenzialmente dannoso. È quindi utilizzato il processo di cifratura per memorizzare i dati su memoria in modo da garantire la loro riservatezza. I dati, per poter essere letti, devono prima essere decifrati e questa operazione si svolge interamente all'interno della CPU dove è anche memorizzata la chiave di cifratura. L'architettura fornisce due modi per cifrare i dati nell'enclavi. Nel primo caso si genera una chiave univoca per ogni enclave dell'applicazione. La chiave è valida solo per la singola enclave allo stato attuale ed è sensibile alla modifica del codice sorgente dell'enclave. In questo

[13]Con la rimozione del package della CPU, l'utilizzo di strumentazione avanzata per separare singoli strati del processore e l'uso di microscopi a fasci ionici è possibile recuperare la chiave di cifratura memorizzata nella CPU [3].

modo, i dati di un'enclave non possono essere letti da altre enclavi, nemmeno della stessa applicazione con una nuova versione. Nel secondo caso si genera una chiave unica per più enclavi. In questo modo è possibile che le enclavi di un'applicazione condividano la stessa chiave per le operazioni di cifratura. Questo approccio è utile quando diverse enclavi interagiscono per uno scopo comune. Infatti, condividendo la chiave sarà possibile leggere e scrivere dati condivisi tra più enclavi garantendo comunque la protezione dall'ambiente esterno.

2.1.2 Protezione dei dati con Intel SGX

In questa sezione si descrivono le proprietà di sicurezza offerte da Intel SGX. Inoltre, si descrive l'architettura interna del Memory Encryption Engine, componente essenziale per la gestione delle regioni di memoria protette. Infine, si descrivono gli algoritmi utilizzati e le strategie per gestire alcuni possibili attacchi.

Intel SGX è supportato dal *Memory Encryption Engine* (MEE), un circuito aggiuntivo che risiede nel Memory Controller (MC). Questo circuito gestisce delle regioni di memoria principale chiamate MEE Region mentre le altre regioni di memoria non protette sono lasciate al MC. Le proprietà offerte dalla gestione effettuata da MEE sono la confidenzialità dei dati e l'integrità. Inoltre, il Memory Encryption Engine gestisce gli attacchi di tipo replay[14] (utilizzando un MAC[15] stateful), l'attacco di traffic analysis[16] (rendendo casuali gli indirizzi ed il momento della scrittura dei dati in memoria) e l'attacco cold boot[17] (memorizzando le chiavi internamente ad MEE)[18].

MEE utilizza per le regioni di memoria protette la cifratura con l'algoritmo AES 128 [4] CTR [5] per fornire la proprietà di riservatezza mentre utilizza un algoritmo tipo Carter-Wegman MAC[19] troncato a 56 bit per la proprietà di integrità. La chiave utilizzata per la cifratura e la chiave utilizzata per il calcolo del MAC sono memorizzate nella memoria interna del MEE [6].

In Figura 2.2 sono rappresentate le parti hardware coinvolte nella lettura di una pagina protetta. Alla sinistra è rappresentato il core della CPU che fa richiesta della pagina all'interfaccia della cache adiacente. Al centro è mostrata la struttura interna del Memory Controller che contiene il componente MEE. Infine, al lato destro della Figura 2.2 è rappresentata la memoria principale RAM.

[14] L'attaccante che svolge un attacco di tipo replay intercetta e scambia i nuovi messaggi con vecchi messaggi inviati precedentemente.

[15] Il Message Authentication Code (MAC) è un blocco di dati utilizzato per garantire l'integrità e l'autenticazione di un messaggio. Questo blocco, anche chiamato tag, viene generato da un algoritmo MAC utilizzando una chiave segreta ed i dati del messaggio.

[16] Il traffic analysis è il processo di accumulo di dati proveniente da messaggi scambiati tra due entità esterne, al fine di analizzare ed estrarre informazioni sensibili.

[17] L'attacco cold boot è eseguito da un attaccante che ha accesso diretto alla macchina in funzione. Questa viene arrestata per poter poter leggere il contenuto della memoria RAM e rilevare informazioni segrete.

[18] https://software.intel.com/sites/default/files/332680-002.pdf#page=166

[19] http://www.nuee.nagoya-u.ac.jp/labs/tiwata/ask2016/slides/ask2016_08_

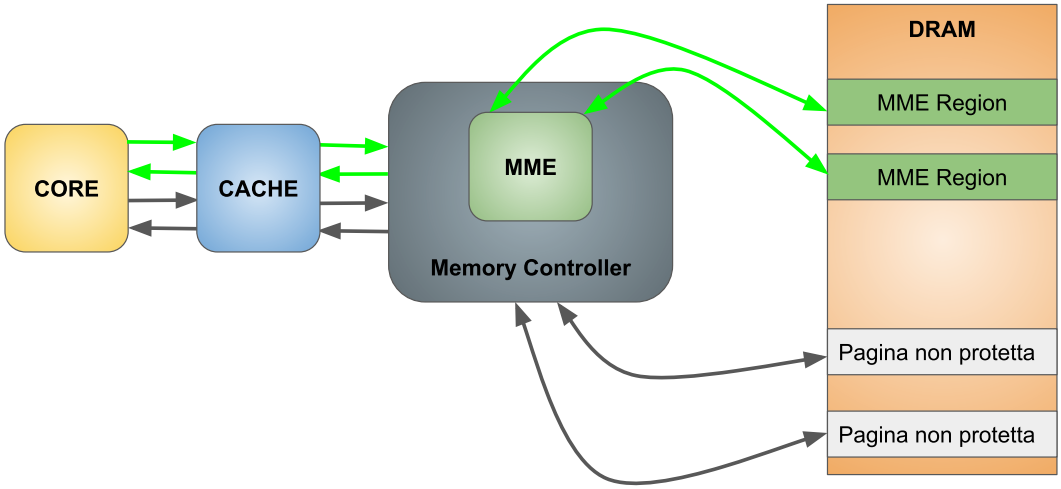


Figura 2.2: Schema hardware ad alto livello per l'accesso alle regioni protette.

Quando si ha un accesso in memoria a seguito di un cache miss[20], la richiesta della pagina di memoria arriva al controllore della memoria. Questo, individua se l'indirizzo della pagina richiesta appartiene ad una regione del MEE. In caso affermativo, la richiesta viene gestita dal MEE che si fa carico di gestire le operazioni richieste per la lettura o la scrittura come la decifratura e la cifratura della pagina.

2.1.3 Flusso delle applicazioni compilate con Intel SGX

Le applicazioni compatibili con Intel SGX hanno due tipi di componenti: il codice sorgente esposto (di seguito untrusted) ed il codice protetto dall'enclave (di seguito trusted).

Come mostrato in Figura 2.3 queste due parti fanno parte del codice binario dell'applicazione e collaborano per un obbiettivo comune. Quando l'applicazione viene eseguita, essa inizia eseguendo la parte di codice non sicuro che deve istanziare l'enclave per poter eseguire le chiamate protette[21]. Le chiamate protette sono funzioni che vengono invocate dalla parte esterna dell'applicazione e vengono eseguite all'interno dell'enclave, fornendo così le proprietà di sicurezza di SGX. Per la creazione dell'enclave viene ricercato e caricato in memoria il file di libreria dinamica associato, generato in fase di compilazione e contenente il codice dell'enclave. A questo punto l'enclave è memorizzata in memoria protetta e si può procedere con le chiamate al codice trusted. Il codice eseguito all'interno dell'enclave è protetto dal software esterno anche se questo è eseguito con un livello di privilegio più

`Jooyoung.pdf`

[20]Un cache miss è un evento che si verifica quando il dato richiesto dalla CPU non è presente nella memoria cache. Il dato viene quindi caricato dalla memoria principale RAM per poterlo servire dalla cache alle prossime richieste del processore.

[21]https://software.intel.com/sites/default/files/managed/c3/8b/intel-sgx-product-brief-2019.pdf

elevato. Si eseguono le chiamate protette finché l'applicazione lo necessita e quando non più necessario, vengono liberate le risorse occupate dall'enclave. Quindi, il codice untrusted continua la sua esecuzione. Come descritto in Sezione 2.2.1, nell'architettura SGX vengono definite ECall le funzioni protette di ingresso al codice dell'enclave, mentre si definiscono OCall le funzioni non protette invocate dal codice interno all'enclave.

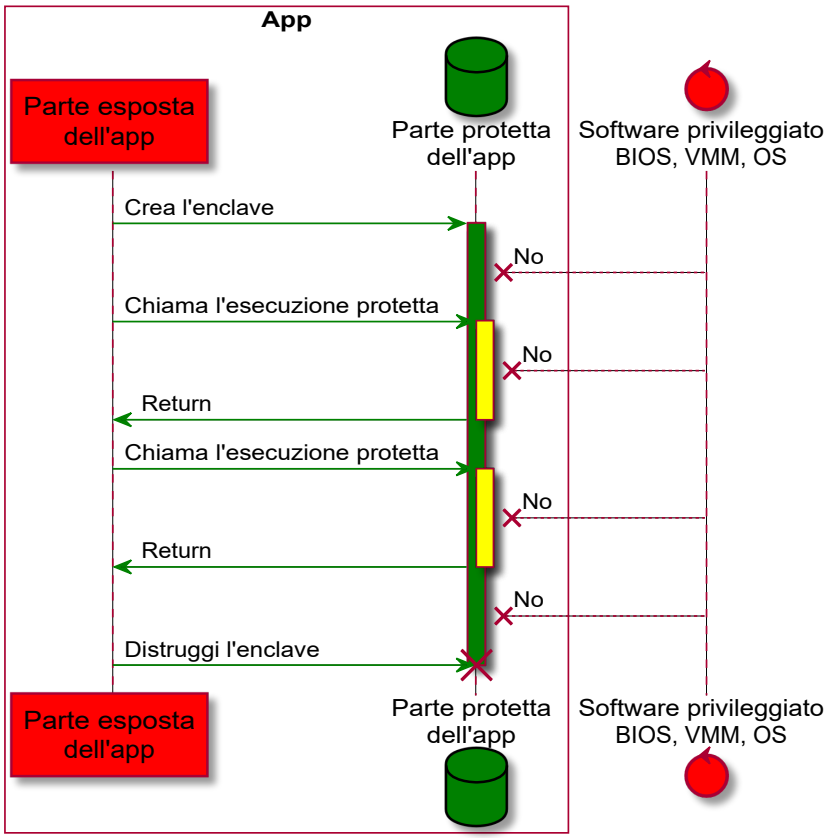


Figura 2.3: Flusso di esecuzione delle applicazioni Intel Software Guard Extensions.

2.1.4 Attestazione offerta da SGX

Con attestazione si intende il processo di verifica dell'affidabilità di un software eseguito su una piattaforma. Nel progetto di questa tesi non si utilizzeranno i processi di attestazione ma, per completezza, si descrivono le principali caratteristiche e proprietà offerte dalle attestazioni SGX. L'architettura fornisce un meccanismo mediante il quale è possibile stabilire se un'entità software sia in esecuzione all'interno di un'enclave su una macchina abilitata a Intel SGX. Questo meccanismo ha lo scopo di verificare che l'entità software sia protetta e affidabile, prima di iniziare uno scambio di dati protetti con essa. L'attestazione di SGX si basa sulla capacità della piattaforma di generare una credenziale che rifletta accuratamente un'enclave e lo stato della piattaforma su cui è in esecuzione. Esistono due tipi di attestazione supportate dall'architettura SGX: l'attestazione locale e l'attestazione remota. La prima viene utilizzata quando due o più enclave eseguite sulla stessa macchina

hanno necessità di scambiarsi dati in modalità protetta. La seconda invece, viene utilizzata per eseguire un'enclave su una macchina remota e verificare che l'enclave sia in esecuzione con supporto alle proprietà offerte da SGX.

Attestazione locale

L'*attestazione locale* è un processo che si verifica prima che due o più enclavi sulla stessa macchina condividano localmente dei dati[22]. Questo processo consente ad un'enclave di dimostrare, ad un'altra parte all'interno della stessa piattaforma locale, la propria identità e autenticità. Un'enclave può chiedere all'hardware Intel SGX di generare una credenziale, nota come report, che include la prova crittografica dell'esistenza dell'enclave sulla piattaforma. Poi, il report è consegnato ad un'altra enclave, che lo verifica assicurandosi che sia stato generato sulla stessa piattaforma. Per fare ciò, il meccanismo utilizza una chiave simmetrica incorporata nella piattaforma hardware e non disponibile esternamente.

La Figura 2.4 mostra il flusso dell'attestazione locale tra due enclavi appartenenti a due diverse applicazioni ma il processo non varia se le enclavi appartengono alla stessa applicazione. L'attestazione può essere reciproca (come mostrato in Figura 2.4) o singola escludendo quindi la fase numero 3. Le fasi del flusso risultano:

1. dopo che le applicazioni hanno stabilito un percorso di comunicazione tra le due enclavi, l'enclave B invia la sua identità (chiamata MRENCLAVE) all'enclave A;

2. l'enclave A utilizza l'hardware per generare una struttura di report destinata all'enclave B utilizzando il valore MRENCLAVE appena ricevuto e trasmette il suo report all'enclave B; l'enclave B esegue la verifica del report utilizzando l'hardware SGX che afferma se l'enclave A sia eseguito sulla stessa piattaforma dell'enclave B;

3. il processo continua all'inverso, l'enclave B genera e trasmette il proprio report all'enclave A; il report viene verificato allo stesso modo dall'enclave A.

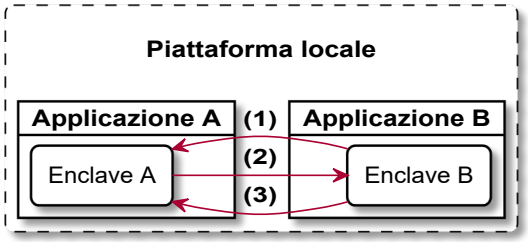


Figura 2.4: Flusso dell'attestazione locale con SGX.

[22] https://software.intel.com/en-us/node/702983

Dopo la verifica dell'autenticità delle due parti è possibile proseguire con lo scambio di dati. È importante ricordare che un'enclave non ha accesso ai dati di un'altra, anche se entrambe appartengono alla stessa applicazione. È quindi necessario un protocollo di scambio chiavi (come ECDH [7]), per condividere una chiave di sessione al fine di proteggere con cifratura i dati scambiati tra due enclavi. Inoltre, un'enclave non può accedere allo spazio di memoria protetta di un'altra enclave, di conseguenza tutti i puntatori scambiati come argomenti devono essere sottoposti a marshalling[23] per poter condividere i valori tra le enclavi.

Attestazione remota

L'*attestazione remota* viene utilizzata per verificare l'esecuzione e la generazione dell'enclave eseguita su una macchina remota[24]. L'applicazione che gestisce l'enclave può richiederle di generare un report che rifletta lo stato dell'enclave. Il report è processato dalla Quoting Enclave (QE), un'altra enclave della piattaforma che produce una credenziale, nota come citazione, che riflette la principale enclave e lo stato della piattaforma. La citazione è passata all'entità al di fuori della piattaforma e viene verificata utilizzando le tecniche di verifica Intel EPID[25] (Enhanced Privacy ID).

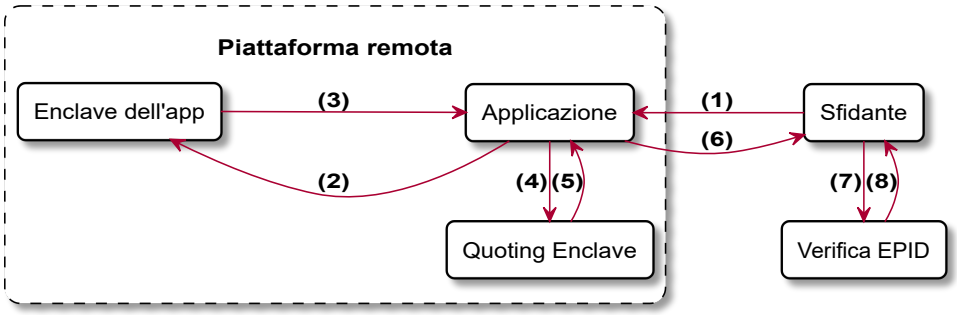


Figura 2.5: Flusso dell'attestazione remota con SGX.

In Figura 2.5 sono riportati i passaggi per l'esecuzione di un'attestazione remota che risultano essere:

1. lo sfidante si collega alla piattaforma remota e comunica una sfida contenente un nonce[26] all'applicazione;

2. l'applicazione comunica il nonce all'enclave e le richiede il report;

[23] Il marshalling è il processo di conversione dei dati da rappresentazione interna a rappresentazione utile per trasmissione e memorizzazione. I dati convertiti devono essere comprensibili da altre applicazioni e non devono dipendere da impostazioni locali della macchina.

[24] https://software.intel.com/en-us/node/702984

[25] https://software.intel.com/en-us/node/702985

[26] Un nonce è un numero casuale utilizzato una sola volta.

3. l'enclave genera e restituisce il report;
4. il report viene consegnato all'enclave QE per la firma;
5. l'enclave QE autentica il report, lo firma con la chiave Intel EPID e lo restituisce all'applicazione;
6. l'applicazione comunica report firmato allo sfidante;
7. lo sfidante utilizza il preventivo e Intel EPID per per verificare la validità;
8. vengono confrontate le informazioni della configurazione attendibile con quelle presenti nel preventivo e si considera il servizio affidabile solo se le informazioni delle due parti coincidono.

2.1.5 Attacchi

In questa sezione sono riassunti alcuni attacchi effettuati contro l'architettura Intel SGX. Di seguito, l'attacco Prime+Probe utilizza gli accessi alla memoria cache ed alcune istruzioni del processore per calcolare le chiavi di cifratura dell'enclave vittima. Un altro attacco fatto a Intel SGX sfrutta la vulnerabilità Spectre[27] dei processori con esecuzione speculativa per alterare l'esecuzione delle istruzioni ed inserire in memoria contenuti altrimenti inaccessibili. Inoltre, in Sezione 2.1.5 si descrivono i rischi legati ai software malevoli eseguiti all'interno di regioni protette e si descrivono i risultati di progettazione di malware eseguiti in enclavi SGX.

Attacco Prime+Probe

L'attacco *Prime+Probe* è stato sviluppato nel 2019 da alcuni ricercatori di Graz University of Technology, in Austria. Questo di basa sull'utilizzo della tecnologia Intel SGX per nascondere un malware in grado di recuperare chiavi RSA generate ed utilizzate dal codice eseguito all'interno di altre enclavi SGX. Il team di ricercatori ha creato un attacco chiamato "Prime + Probe" [8] dimostrandone la corretta esecuzione sia in un ambiente Intel SGX nativo sia attraverso l'uso di container Docker[28].

Il malware è progettato per recuperare le chiavi RSA di altre enclavi in un processo diviso in tre fasi che localizzano la parte di memoria cache della vittima, leggono la parte di memoria durante la generazione della firma ed estraggono la chiave.

La tecnica utilizzata usa una misurazione di tempi ad alta risoluzione sfruttando le istruzioni del processore `inc` e `add`, e le specifiche dei processori Intel, come il numero di colpi di clock necessari e la frequenza del processore. Queste istruzioni vengono utilizzate rispettivamente per incrementare un registro e sommare due valori. Poi, la strategia monitora gli accessi alla memoria DRAM e cache, per recuperare

[27] https://meltdownattack.com/

[28] https://www.docker.com/

i bit degli indirizzi fisici della regione dell'enclave di interesse. Poiché la memoria dell'enclave SGX viene allocata in modo contiguo è possibile eseguire la ricerca dell'enclave anche ad indirizzi virtuali. Poi, il malware attacca l'implementazione RSA di mbedTLS[29] utilizzata ad esempio in OpenVPN[30].

L'attacco avviene monitorando le parti di cache vulnerabili, ricercando i dati del calcolo della chiave RSA contenuti nell'immagine protetta dell'enclave. Vengono cosi acquisiti i dati di più esecuzioni della creazione dell'enclave. Al termine delle acquisizioni i dati vengono processati per rimuovere il rumore acquisito causato da errori di temporizzazione e dai cambi di contesto dovuti allo scheduler[31]. Per ogni esecuzione viene estratta una chiave parziale utilizzata insieme alle altre per ripristinare la chiave privata originale.

Questo attacco ha successo nonostante la protezione offerte da SGX, descritte in Sezione 2.1, e riesce ad estrarre il 96% di una chiave privata RSA a 4096 bit con sole 11 esecuzioni. Secondo lo studio è possibile avere il ripristino completo della chiave in soli 5 minuti circa [8]. L'attacco funziona anche su diversi contenitori Docker, poiché il motore Docker chiama lo stesso driver SGX per entrambi i contenitori.

Attacco Spectre

Il team LSDS all'Imperial College di Londra[32] ha effettuato delle ricerche riguardo l'esecuzione di attacchi di tipo Spectre in ambienti protetti con Intel SGX. Questo tipo di attacco può essere adattato per l'esecuzione in enclave a seguito di un'ispezione del codice ed un'esecuzione speculativa. Infatti, il gruppo ha rilasciato il codice per eseguire localmente questo tipo di attacco rinominato SGX Spectre per i processori SGX[33]. Lo studio suggerisce che i programmi eseguiti in enclave sono spesso vulnerabili agli attacchi SGX Spectre poiché le vulnerabilità necessarie sono disponibili nella maggior parte delle librerie SGX (e.g. Intel SGX SDK) [9]. Inoltre, queste vulnerabilità sono introdotte anche con il codice utente eseguito in enclave. Intel SGX ha quindi rilasciato delle linee guida per gli sviluppatori al fine di ridurre le vulnerabilità del codice e ridurre i rischi dovuti agli attacchi di tipo Spectre [10].

L'attacco di tipo Spectre sfrutta la branch-prediction (la previsione del ramo condizionale) per alterare l'esecuzione delle istruzioni al fine di inserire nella memoria contenuti altrimenti inaccessibili. La branch-prediction è una funzionalità delle CPU Intel, utilizzata per velocizzare l'esecuzione del codice, in cui il processore cerca di predire i rami di codice da eseguire in base alle istruzioni eseguite precedentemente. Ad esempio, in caso di un costrutto `if`, la CPU suppone quale ramo debba essere eseguito prima che l'esecuzione della condizione sia verificata.

[29] https://tls.mbed.org/api/

[30] https://openvpn.net/private-tunnel/

[31] Lo scheduler è un processo del sistema operativo che gestisce le esecuzioni dei programmi. Questo componente stabilisce l'ordine e la durata delle esecuzioni dei programmi ottimizzando l'accesso alle risorse.

[32] https://lsds.doc.ic.ac.uk/projects/sereca

[33] https://github.com/lsds/spectre-attack-sgx

```
void victim_function(size_t x) {
       if (x < array1_size) {
              temp &= array2[array1[x] * 512];
       }
}
```

Figura 2.6: Esempio di vulnerabilità del codice ad attacco Spectre.

Eseguendo il blocco in Figura 2.6 più volte, l'attaccante può addestrare il branch-predictor per forzare l'esecuzione del codice indipendentemente dal valore in ingresso. Una volta eseguito l'addestramento, l'attacco consiste nell'esecuzione del blocco con valori di `x` al fine di accedere agli indirizzi di memoria che non rientrano nei limiti del vettore. Poi, combinando questo attacco con il monitoraggio della memoria cache, l'attaccante è in grado di condizionare il contenuto visto dal processo della vittima e osservare i cambiamenti della memoria al fine di conoscere i segreti dell'applicazione. Tuttavia, l'attacco Spectre può funzionare solo se l'indirizzo di destinazione è accessibile dal contesto del processo vittima ed inoltre per eseguire questo attacco il codice vittima deve presentare delle vulnerabilità di questo tipo. Questo implica che l'attaccante intenzionato ad eseguire un attacco Spectre debba prima analizzare il codice sorgente per identificare le funzioni che sono vulnerabili.

Enclave malware

L'architettura Intel SGX rappresenta una protezione ai fini di rendere il codice ed i dati protetti nell'enclavi inaccessibili dall'esterno, indipendentemente dai privilegi degli altri processi. Tuttavia, questa protezione non impedisce che il codice all'interno dell'enclavi sia un codice malware. Infatti, il codice inserito all'interno dell'enclavi potrebbe essere un malware o un codice appositamente creato per rubare informazioni personali o danneggiare l'utente e la macchina in cui è in esecuzione. Questa opportunità potrebbe essere un modo per rendere i malware più forti, poiché con l'utilizzo di Intel SGX, né il sistema operativo né l'antivirus può accedere ai dati ed al codice dell'enclave vanificando le analisi effettuate per riconoscere i codici pericolosi.

Alcuni ricercatori della Graz University of Technology hanno eseguito del codice maligno su sistemi con CPU Intel in modo tale da impedire agli antivirus di analizzare il malware [11]. Il loro studio ricerca i limiti ed i modi in cui un malware possa danneggiare il sistema e dimostra che il codice malevolo possa implementare un ramsonware per rubare e cifrare documenti ed estorcere denaro all'utente. Inoltre, il codice può agire per conto dell'utente, come inviare email o strutturare un nodo per gli attacchi di denial-of-service.

Inoltre, i ricercatori hanno sviluppano un nuovo tipo di attacco nominato SGX-ROP che utilizza il nuovo set di istruzioni di memoria Transactional Synchronization eXtensions (TSX) per eliminare le protezioni fornite dall'Address Space Layout Randomization (ASLR), stack canaries e address sanitizer, ed avere quindi pieno

accesso alla memoria non cifrata del sistema operativo [11]. Le istruzioni TSX sono istruzioni del processore che mirano ad accelerare l'esecuzione di software multi-thread aggiungendo il supporto alla memoria transazionale. Questo tipo memoria è in grado di eseguire eseguire le operazioni di `load` e `store`[34] in modo atomico. La protezione ASLR consiste nel caricare a indirizzi casuali le librerie e i programmi, in modo da nascondere le funzioni di libreria. In questo modo l'attaccante è costretto ad indovinare l'indirizzo delle funzioni da attaccare, generando ad ogni tentativo errato un'eccezione che termina con il crash del programma[35]. Gli stack canaries sono dati inseriti ai limiti della regione di memoria stack del programma. Questi fungono da limite per lo stack overflow, la condizione in cui il programma scrive oltre la propria regione di memoria stack. L'address sanitizer[36] è uno strumento dei compilatori che rileva i tentativi di indirizzi di memoria invalidi, come puntatori invalidi (Dangling pointer) e accesso oltre i limiti dei vettori.

2.1.6 Limiti di Intel SGX

Il primo limite della tecnologia è il requisito di eseguire le applicazioni sicure con processori compatibili. Intel rende disponibile uno strumento[37] di ricerca per trovare i processori che supportano la tecnologia Intel SGX. In via generale, è necessario usare una macchina con un processore Intel Core di sesta generazione o successiva e abilitare il supporto all'opzione di SGX dal boot del sistema operativo. Inoltre, a seconda del sistema operativo in uso, sono richiesti strumenti per compilare, eseguire il debug e rilasciare l'applicazione[38]. Per i sistemi operativi Microsoft Windows e Linux, per eseguire le applicazioni protette è necessario installare il pacchetto Intel SGX SDK e seguire la guida di installazione disponibile in Appendice A. In particolare, sui sistemi Windows 10 è necessario installare una versione di Visual Studio 2015 professional o successiva e compilare l'applicazione dal programma. Per Linux invece, sono disponibili e consigliati due diversi approcci: utilizzare la GNU toolchain per compilare il codice attraverso lo strumento Make, oppure utilizzare il programma Eclipse con il relativo plugin per compilare applicazioni Intel SGX, come descritto nella guida di SGX per Linux[39].

Di seguito si analizzano i principali limiti della tecnologia.

---

[34] Le istruzioni di `load` e `store` servono rispettivamente a leggere e memorizzare i dati nella memoria principale.

[35] https://pax.grsecurity.net/docs/aslr.txt

[36] https://github.com/google/sanitizers

[37] https://www.intel.it/content/www/it/it/support/articles/000028173/processors.html

[38] https://software.intel.com/sites/default/files/managed/c3/8b/intel-sgx-product-brief-2019.pdf

[39] https://01.org/sites/default/files/documentation/intel_sgx_sdk_installation_guide_for_linux_os.pdf

Dimensione della memoria protetta

La dimensione della memoria protetta è impostata dal BIOS del sistema ed ha come valori tipici 64 o 128 MB. Alcune interfacce di boot permettono la configurazione di questo valore[40]. Questa dimensione di memoria è tuttavia una misura globale di tutto il sistema e le enclavi attualmente allocate sul sistema condividono questo spazio. A seconda della dimensione delle enclavi si stima un valore tra 5 e 20 differenti enclavi allocate simultaneamente[41].

Eventi di alimentazione

Alcuni eventi di alimentazione, come la sospensione e l'ibernazione del sistema, causano la perdita dei dati delle enclavi quali la cache in cui sono memorizzati i dati delle strutture protette, le chiavi di cifratura e le regioni di memoria protetta salvate in RAM[42]. Questo è uno dei principali limiti della tecnologia. Infatti, le transizioni di alimentazione devono essere gestite in modo differente in base al momento in cui avvengono. Se la transizione avviene durante l'esecuzione di codice non protetto, questo alla ripresa continuerà a funzionare e dovrà gestire la perdita dell'enclave. A questo scopo è possibile utilizzare primitive del sistema operativo per iscriversi agli eventi di cambio stato e allocare nuovamente l'enclave. In questo caso, la perdita di dati sarà completamente da ricercarsi nella memoria protetta e non sarà possibile recuperare tali informazioni. Invece, se la transizione di stato avviene durante l'esecuzione di una funzione ECall o nel caso peggiore di una chiamata OCall, alla ripresa del sistema il codice untrusted riceverà come valore di ritorno alla funzione ECall lo stato di errore `SGX_ERROR_ENCLAVE_LOST`. A questo punto, oltre ad essere stato perso il contenuto dell'enclave, il codice non protetto non avrà modo di verificare il punto del crash e nel caso di esecuzione di OCall, un'eventuale scrittura di dati in memoria o file potrebbe risultare corrotta e incompleta.

Diminuzione delle prestazioni

L'uso delle funzioni protette introduce un rallentamento delle prestazioni dovuto principalmente a due fattori. Il primo risiede nell'insieme di chiamate a funzione ed istruzioni SGX che devono essere aggiunte per raggiungere l'esecuzione del codice protetto. Queste chiamate automaticamente generate durante la compilazione sono responsabili della produzione del valore di ritorno dello stato di esecuzione della chiamata. La seconda causa di perdita di prestazioni è il processo di cifratura e decifratura che serve all'architettura per mantenere le proprietà di integrità e

[40] https://www.dell.com/support/manuals/it/it/itbsdt1/vostro-14-3481-laptop/vos_3481_setup_specifications_guide/intel-software-guard-extensions?guid=guid-873f3274-7d4c-4b63-bbd2-9de7771aea95&lang=en-us

[41] https://software.intel.com/en-us/articles/intel-software-guard-extensions-tutorial-part-1-foundation

[42] https://software.intel.com/en-us/articles/intel-sgx-tutorial-part-9-power-events-and-data-sealing

riservatezza della memoria protetta. In generale le prestazioni delle funzioni protette possono essere paragonate a quelle insicure. Questo significa che una volta decifrata la memoria contenente il codice invocato, questo viene eseguito in modo veloce. Sono invece le procedure di cambio di contesto (entrata e uscita dall'enclave) a rallentare notevolmente l'esecuzione, come descritto in Sezione 6.3.2. Si ha quindi una perdita di prestazioni molto pronunciata soprattutto per le transizioni frequenti tra codice dentro e fuori l'enclave[43].

Librerie ridefinite da SGX per l'esecuzione protetta

All'interno dell'enclave non è possibile eseguire codice esterno non sicuro. In generale, le librerie standard non sono disponibili per essere importate nel codice protetto. Intel SGX SDK[44] ridefinisce queste librerie con file nuovi implementati appositamente per l'architettura. Questo nuovo codice espone alcune funzioni delle librerie standard C e C++, e rimuove altre funzioni considerate non sicure. Per esempio non è consentito utilizzare le interfacce standard per input e output per tastiera e schermo, e alcune chiamate al sistema operativo non sono disponibili come per esempio la funzione "`exit`". Questa limitazione può essere risolta utilizzando le chiamate OCall (descritte in Sezione 2.2.1) per definire delle funzioni utente che facciano da ponte tra l'ambiente protetto e le funzioni di libreria standard non disponibile all'enclave.

2.2 Progettazione con Intel SGX

In questa sezione si descrivono i principali aspetti del design del software per l'architettura Intel SGX. Inoltre, si introducono tutte le nozioni dell'ambiente protetto come il file di descrizione dell'enclave ed i tipi di chiamate ECall ed OCall. Poi, si approfondisce la sintassi EDL proprietaria di Intel SGX e si descrive l'utilizzo del file EDL in fase di compilazione. Infine, si elencano gli errori ed i problemi comuni associati all'ambiente protetto e si introducono le modalità di compilazione delle applicazioni SGX.

2.2.1 Design del software

Sia il codice da proteggere con Intel SGX che il codice di interfaccia all'enclave devono essere obbligatoriamente scritti in linguaggio C o C++. Il file generato dall'enclave sarà una libreria dinamica (formato `.dll` in Windows, formato `.so` in Linux). È quindi possibile interfacciarsi con tale libreria per chiamare la parte protetta anche da codice esposto scritto in altro linguaggio. Infatti, in caso di necessità si può sviluppare una nuova libreria dinamica scritta in linguaggio C/C++

[43] https://medium.com/@danny_harnik/impressions-of-intel-sgx-performance-22442093595a

[44] https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions/sdk.html

per interfacciare il codice dell'enclave con il codice scritto in linguaggi diversi a C e C++[45]. In generale, l'applicazione compatibile con Intel SGX deve essere divisa in due parti distinte, come rappresentato in Figura 2.7:

1. l'insieme dei componenti non attendibili, esposti all'ambiente non sicuro;
2. l'insieme di componenti da includere nell'enclave, in modo da garantirne confidenzialità ed integrità.

Il primo componente è rappresentato dal codice esposto dell'applicazione che si occupa della creazione dell'ambiente protetto e della parte di software che non necessita di protezione. Questo software viene eseguito senza protezioni di SGX e non ha modo di accedere direttamente alla parte da proteggere. Il secondo componente invece è la parte di codice e di dati sensibili che necessitano della protezione offerta da Intel SGX. Questi vengono inseriti nell'enclave in modo tale da eseguire il codice e memorizzare i dati in memoria protetta al fine di renderli inaccessibili dall'esterno.

I due componenti collaborano durante l'esecuzione dell'applicazione ed interagiscono scambiandosi dati dove necessario. A questo scopo si utilizzano i punti di ingresso ed uscita dall'enclave chiamati rispettivamente *ECall* ed *OCall*. Queste vie sono le uniche modalità di comunicazione tra i due ambienti come spiegato in Sezione 2.2.1.

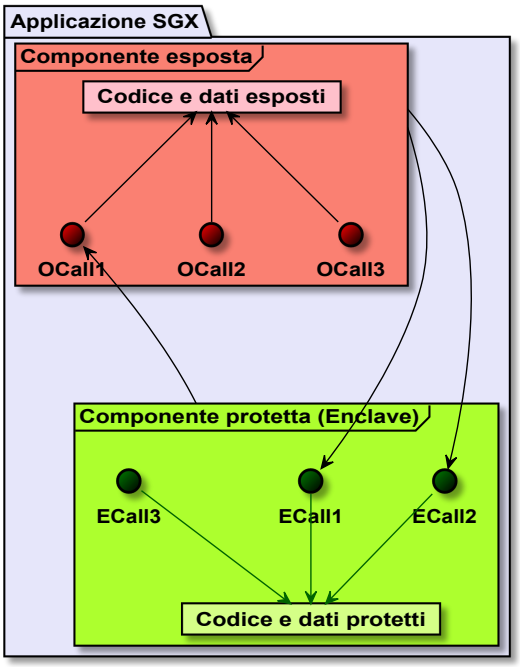


Figura 2.7: Componenti di un'applicazione Intel SGX.

[45] https://software.intel.com/en-us/articles/intel-software-guard-extensions-tutorial-part-2-app-design

Una buona condotta durante lo sviluppo di applicazioni SGX è mantenere il componente attendibile il più piccolo possibile, sia in termini di codice inserito al suo interno sia in termini di dati transitati in memoria protetta, in modo da preservare le prestazioni dell'applicazione, come spiegato in Sezione 2.1.6. Inoltre, è consigliato limitare i punti di entrata e di uscita dall'enclave per poter ridurre il più possibile la superficie di attacco della parte protetta. Infine, risulta buona norma limitare le variabili statiche grandi e le grandi allocazioni in memoria protetta poiché lo spazio a disposizione è limitato e condiviso tra tutte le enclavi del sistema, come spiegato in Sezione 2.1.6.

Chiamate ECall e chiamate OCall

In questa sezione si descrivono le funzionalità delle chiamate ECall ed OCall analizzando le modalità di utilizzo ed i loro limiti.

Come descritto in Sezione 2.1, l'architettura Intel SGX alloca uno spazio di memoria dove i dati ed il codice contenuto sono protetti dai processi con privilegi di esecuzione elevati. Infatti, con le proprietà di integrità e riservatezza offerte da SGX, il contenuto dell'enclave non è direttamente comprensibile nè modificabile dall'esterno, nemmeno dal sistema operativo. Esistono quindi delle regole per poter comunicare con il codice all'interno dell'enclave al fine di scambiare dati ed eseguire del calcolo in modalità protetta.

L'architettura SGX definisce due tipi di chiamate a funzione nominate Enclave call (ECall) ed Outside call (OCall) [12]. La chiamata ECall viene utilizzata per eseguire una funzione inserita dentro all'enclave. Quando viene eseguita una chiamata ECall, le librerie dell'enclave (descritte in Sezione 2.2.3) eseguono delle istruzioni hardware dei processori Intel che commutano l'esecuzione in ambiente protetto. A questo punto, l'esecuzione continua all'interno dell'enclave e termina al ritorno della funzione destinazione della chiamata ECall. La funzione di ingresso dell'enclave ha visibilità dei dati inseriti al suo interno e conosce il resto del codice dell'ambiente. Questa funzione può quindi richiamare altre funzioni interne all'enclave nello stesso modo del codice standard in linguaggio C. Per semplicità, di seguito si chiameranno funzioni ECall le chiamate a funzioni d'ingresso all'enclave e funzioni OCall le chiamate a funzioni di uscita dall'esecuzione protetta.

Tuttavia il codice interno all'enclave non conosce l'ambiente esterno e non ha visibilità sul resto del codice dell'applicazione. Per colmare questa mancanza, SGX definisce le chiamate OCall che permettono al codice protetto di eseguire codice esterno all'enclave. Quando viene eseguita una chiamata OCall, le librerie dell'enclave (descritte in Sezione 2.2.3) fermano l'esecuzione dell'ambiente protetto ed eseguono un funzione dell'applicazione esterna all'enclave. Al suo ritorno, l'esecuzione continua nell'ambiente protetto come se si fosse eseguita una chiamata a funzione semplice.

Le chiamate ECall ed OCall hanno come destinazione una funzione standard in linguaggio C o C++ definita dall'utente. Tuttavia, al fine di eseguire le istruzioni interne di SGX per commutare l'esecuzione tra i due ambienti, il codice invocato all'esecuzione di una chiamata ECall o OCall è una funzione di gestione generata automaticamente dal compilatore, come approfondito in Sezione 2.2.3. Questa

ha lo stesso nome della funzione destinazione dell'ambiente opposto ma aggiunge parametri ai suoi argomenti e differisce per il valore di ritorno. Per entrambi i tipi di chiamata il valore di ritorno diventa un codice di stato sull'esecuzione della chiamata. Questo codice indica il successo di esecuzione o il codice di errore che ha impedito l'esecuzione della chiamata. Il valore di ritorno della funzione destinazione è incapsulato e ritornato tramite un puntatore passato alla funzione di gestione come parametro aggiuntivo alle chiamate ECall ed OCall.

Infine, le funzioni di gestione delle chiamate ECall aggiungono nei propri argomenti un ulteriore parametro utilizzato come identificatore dell'enclave destinazione. Senza questo identificatore conosciuto alla creazione dell'enclave, non sarebbe possibile individuare l'enclave contenente la funzione destinazione richiesta.

La definizione delle funzioni destinazione delle chiamate ECall ed OCall ha luogo all'interno del file EDL (Enclave Definition Language). Questo file di sintassi proprietaria di Intel SGX, viene scritto manualmente dal programmatore e viene processato e compilato insieme al codice sorgente dell'applicazione. Il file EDL e la sua sintassi vengono approfonditi in Sezione 2.2.2.

2.2.2 File di descrizione dell'enclave

In questa sezione si descrive l'utilizzo e la sintassi del file EDL (Enclave Definition Language) utilizzato nella definizione dell'enclave.

Il file in formato *EDL* viene utilizzato dal programmatore per definire completamente ed unicamente un'enclave. Questo file viene processato dallo strumento `sgx_edger8r` [12] per generare le librerie in linguaggio C dell'enclave, come definito in Sezione 2.2.3. Il file di descrizione dell'enclave è diviso in tre differenti parti per definire strutture dati, ECall e OCall. Queste definizioni vengono descritte con il formato EDL che risulta molto simile alla sintassi dei file header in C. In ogni definizione ECall ed OCall si ritrovano i dati sul nome della funzione destinazione, i nomi degli argomenti, i tipi degli argomenti ed il valore di ritorno. Questi dati vengono utilizzati per generare le funzioni di gestione delle chiamate ECall ed OCall per commutare l'esecuzione tra i due ambienti, come descritto in Sezione 2.2.1. Inoltre, l'architettura SGX impone la definizione di un ulteriore parametro per ogni argomento puntatore passato alle chiamate ECall ed OCall. Infatti, i dati puntati dalla memoria cifrata, non possono essere letti dall'esterno dell'enclave poiché in caso di lettura si otterrebbero solo informazioni incomprensibili. A questo scopo, si introducono delle modalità di copia per i puntatori, in modo da copiare le loro strutture dati o i loro buffer allocati e comunicare i valori decifrati alle chiamate OCall.

Nella Figura 2.8 si nota un file in formato EDL diviso in tre parti: la definizione di strutture dati, la definizione delle funzioni trusted e la definizione delle funzioni untrusted. Queste ultime due sono rispettivamente le funzioni destinazione delle chiamate ECall ed OCall. Inoltre, è possibile notare vari esempi per la definizione del metodo di copia `in, out, in/out` o per la definizione `user_check`. Le chiamate ECall possono essere definite pubbliche o private utilizzando l'attributo `public`. Le chiamate ECall private possono essere invocate solo dal codice OCall a cui è

stato concesso il permesso tramite l'attributo "`allow(nome_funzione)`". Le definizione delle strutture dati inserite all'interno del file EDL vengono riportate nelle librerie dell'enclave in modo tale da renderle disponibili al codice sorgente protetto. La generazione delle librerie dell'enclave a partire dal file EDL è disponibile in Sezione 2.2.3.

```
enclave {
    struct struct_foo_t {
        int a;
        char v[10]
    };

    trusted {
        public void funzione_ecall(float val, struct struct_foo_t
            str);
        public void codifica ([in, out, size = num_byte] void *
            ptr, size_t num_byte);
        public int calcola_media ([in, out, count = cnt] int *
            arr, size_t cnt);
        void codifica_danger ([user_check] void * val, size_t sz);
    };

    untrusted {
        void ocall_foo([user_check] int *char);
        void ocall_foo_danger(void) allow(ecall_function_private);
    };

};
```

Figura 2.8: Esempio di file EDL in sintassi proprietaria di Intel SGX.

Le copie dei buffer vengono utilizzate per rendere il codice dell'enclave più robusto. Infatti, il codice interno all'enclave non ha modo di verificare la dimensione e la validità di un puntatore passato negli argomenti. Quindi, in assenza di copia, un codice malevolo potrebbe effettuare una chiamata ECall passando come parametro un puntatore invalido. Il codice della chiamata ECall proverebbe a scrivere sull'indirizzo passato agli argomenti con risultato di un errore di segmentazione che porterebbe al crash del programma. In casi più sofisticati, il codice malevolo potrebbe passare un puntatore alla memoria dell'enclave stesso al fine di far modificare il codice o i dati direttamente da una chiamata ECall.

Per questi motivi i puntatori passati alle chiamate ECall ed OCall devono essere annotati con ulteriori parametri come la modalità di copia, la grandezza della regione di memoria puntata e la direzione di copia. Le modalità di copia dei puntatori

sono[46]:

1. `in`: copia nella regione destinazione del valore iniziale indirizzato dal puntatore;

2. `out`: copia nella regione di partenza del valore finale indirizzato dal puntatore;

3. `in/out`: copia nella regione destinazione del valore iniziale indirizzato dal puntatore ed il valore finale nella regione di partenza.

In particolare con la modalità di copia `in` viene allocato, nella regione destinazione, un buffer di dimensione specificata e viene riempito con i dati dell'argomento. I dati sono copiati prima della commutazione d'ingresso (tra enclave e codice untrusted e viceversa) e le modifiche effettuate nella regione destinazione non alterano i dati iniziali, presenti nella regione di partenza.

Invece, con la modalità di copia `out` viene allocato, nella regione destinazione, un buffer di dimensione specificata. La regione destinazione modifica il buffer inizialmente vuoto ed al ritorno della chiamata, il valore modificato viene copiato nel buffer del chiamante, presente nella regione di partenza. Quindi, i dati sono copiati al ritorno della chiamata.

Infine, con modalità di copia `in/out` viene allocato un buffer di dimensione specificata nella regione di destinazione. Questo buffer viene riempito con i dati iniziali dell'argomento della chiamata, in modo che la regione di destinazione possa leggeri e modificarli. Il valore finale del buffer viene copiato nella regione di partenza, in modo che i dati del chiamante siano aggiornati. Quindi, la copia è effettuata alla partenza e al ritorno della chiamata.

Al fine di effettuare le copie, la libreria dell'enclave ha bisogno delle dimensioni delle strutture dati o dei buffer indirizzati dai puntatori. Queste dimensioni possono essere definite con dei valori numerici attraverso numeri interi o definizioni di costanti (con direttiva di preprocessore `#define`), oppure possono essere associate ad un nome di un parametro numerico passato negli argomenti della funzione. Queste dimensioni vengono utilizzate con gli attributi `size` e `count` per definire la dimensione rispettivamente in byte o come contatore di elementi consecutivi disponibili. In entrambi i casi queste definizioni vanno associate al nome ed al metodo di copia dell'argomento.

In ultimo, è possibile richiedere al compilatore di non effettuare la copia dei valori puntati attraverso l'utilizzo della direttiva `user_check`. Tuttavia, questa direttiva è sconsigliata dalla comunità di SGX [12] poiché il codice interno all'enclave non ha modo di verificare la dimensione e la validità di un puntatore passato negli argomenti.

[46] https://github.com/intel/linux-sgx/blob/master/SampleCode/SampleEnclave/Enclave/Edger8rSyntax/Pointers.edl

2.2.3 Librerie generate dal file EDL

Il file EDL contiene le dichiarazioni delle funzioni per entrare ed uscire dall'enclave. Questo file non è direttamente utilizzabile dal codice C ma deve essere opportunamente processato con lo strumento `sgx_edger8r` [12] per generare quattro file compatibili con il codice sorgente in linguaggio C. Vengono quindi generati due file per la parte untrusted (`Enclave_u.h` e `Enclave_u.c`) e due file per la parte trusted (`Enclave_t.h` e `Enclave_t.c`).

I quattro file sono divisi in codice trusted e untrusted che rappresentano rispettivamente codice dell'enclave e codice esterno all'enclave. Il codice esterno all'enclave importa i file con desinenza `'_u'` mentre il codice sicuro quelli con desinenza `'_t'`. I file di header definiscono il prototipo delle funzioni di confine dell'enclave in base al contesto in cui vengono chiamate, mentre i file di codice C definiscono delle funzioni wrapper che predispongono le vere chiamate alle funzioni definite ECall e OCall.

Si prenda in esempio il lato sicuro del codice. Il codice interno all'enclave, non conosce la dichiarazione, nè tantomeno l'implementazione delle funzioni OCall originali. Per questa ragione, il codice protetto importa i prototipi delle funzioni OCall tramite il file `Enclave_t.h`. Questi prototipi hanno lo stesso nome delle funzioni OCall ma differiscono leggermente dalla dichiarazione originale della funzione. Infatti, la funzione esposta dalla libreria trusted ridefinisce tutte le funzioni OCall con un nuovo codice implementativo che nasconde al suo interno la vera chiamata alla funzione OCall. Questo codice di gestione si occupa di eseguire le istruzioni hardware di SGX, che permettono la commutazione tra esecuzione in ambiente protetto ed esposto.

Allo stesso modo, il codice esterno all'enclave non conosce i veri prototipi delle chiamate ECall, e deve procedere all'importazione delle loro dichiarazioni attraverso il file `Enclave_u.h`. Questo file ridefinisce i prototipi delle chiamate ECall aggiungendo alcuni parametri e cambiando il loro tipo di ritorno come descritto in seguito. Queste definizioni hanno lo stesso nome delle chiamate ECall, tuttavia sono funzioni wrapper che nascondono al loro interno la vera chiamata alla funzione protetta nell'enclave.

In estrema sintesi, quando si ha una commutazione tra gli ambienti, la definizione della funzione da chiamare è il wrapper generato dallo strumento `sgx_edger8r` compilando il file EDL; quindi, alla chiamata si devono aggiungere ulteriori parametri e si deve controllare il valore di ritorno. In tutti gli altri casi si deve procedere con una chiamata semplice dello standard C. Infatti, il codice interno all'enclave può chiamare le funzioni destinatarie di chiamate ECall così come le conosce internamente. Allo stesso modo il codice esterno tratta le chiamate funzioni destinazione delle chiamate OCall come funzioni standard C.

2.2.4 Errori e soluzioni comuni associati alle applicazioni SGX

In questa sezione si descrivono gli errori e le soluzioni più comuni associati alle applicazioni Intel SGX.

Un primo problema che si verifica quando si vuole eseguire del codice compilato per Intel SGX è la verifica della compatibilità hardware e software del sistema in uso. Infatti, per poter eseguire i programmi compilati per SGX è necessario un hardware compatibile con l'architettura, il pacchetto di librerie Intel SGX SDK e l'esecuzione consentita a livello BIOS della modalità protetta. Per questo motivo, l'applicazione deve farsi carico di gestire i problemi dovuti all'incompatibilità del software con il sistema. Per fare ciò sono possibili due diversi approcci:

- l'applicazione funziona solo con i sistemi compatibili (basati su CPU Intel con supporto in hardware per SGX);
- l'applicazione gestisce i problemi di compatibilità e può essere eseguita su sistemi non compatibili con SGX.

La prima opzione è da preferire quando il contesto dell'applicazione e la sensibilità dei dati necessita in ogni caso della protezione offerta da SGX. Tuttavia, non è da escludere la necessità di un'applicazione che possa essere eseguita anche su sistemi legacy dove non è possibile acquisire la compatibilità con SGX. In entrambi i casi, le librerie di Intel SGX SDK offrono funzioni per la verifica della compatibilità hardware e software. Queste funzioni interrogano il sistema per conoscere l'abilitazione del BIOS a SGX e la compatibilità del processore all'architettura. Le informazioni fornite possono essere utilizzate per eseguire il programma senza la creazione dell'enclave ridirigendo le chiamate ECall ed OCall a funzioni alternative in modo tale da essere compatibili a sistemi non SGX.

Un secondo problema ricorrente è la gestione della perdita dell'enclave. La perdita dell'enclave si verifica a seguito di un evento di alimentazione come la sospensione o l'ibernazione del sistema. La perdita può avvenire in tre differenti momenti: durante l'esecuzione di codice non protetto, durante l'esecuzione del codice dell'enclave, durante l'esecuzione delle chiamate OCall. Come spiegato in Sezione 2.1.6, il momento della perdita dell'enclave influisce sulla quantità di dati persi. In ogni caso, la perdita dell'enclave può essere riconosciuta dal codice untrusted in due diversi modi: il codice esposto si iscrive ai segnali degli eventi di alimentazione del sistema operativo; il codice esposto controlla il valore di ritorno delle chiamate ECall. In entrambe le modalità, a seguito dalla perdita dell'enclave il codice deve gestire l'evento limitando il più possibile la perdita di dati e la generazione di errori. La gestione della perdita dipende dal caso concreto ed in linea teorica dalla quantità di dati memorizzati in enclave. Nel caso ideale in cui non ci sono dati memorizzati in enclave e l'esecuzione delle funzioni dipende solo dai dati inseriti, l'enclave potrebbe essere rigenerata nuovamente ed eventuali chiamate ECall fallite potrebbero essere rieseguite con gli stessi dati.

Un terzo problema da gestire nella parte di codice non protetto dall'enclave è la memorizzazione dei segreti. Sebbene, i segreti di un'applicazione Intel SGX dovrebbero essere inseriti tutti nelle enclavi, questi potrebbero comunque transitare nella parte di codice esposta per essere poi memorizzati in enclave. Infatti, i segreti potrebbero essere letti da file cifrati o inseriti in ingresso manualmente dall'utente e siccome le funzioni dell'enclave non possono direttamente eseguire operazioni di input ed output, come descritto in Sezione 2.1.6, i segreti dovrebbero necessariamente

essere memorizzati in variabili del codice esposto per poi essere passati all'enclave. Una buona pratica per ridurre la superficie di attacco di un'applicazione SGX, è l'azzeramento delle variabili utilizzate per contenere i segreti, una volta memorizzati in enclave. Si prenda quindi in esempio un ipotetico gestore di password che utilizza Intel SGX per rendere i dati sensibili incomprensibili ai malware. Il gestore delle password memorizza tutti i dati degli account su un file cifrato su disco e la chiave di cifratura è generata a partire dalla password dell'utente. L'utente quindi inserisce la sua password sulla console in lettura dalla parte esposta dell'applicazione. Questa parte legge i byte cifrati dal file e li invia all'enclave insieme alla password dell'utente. A questo punto, il codice esposto dovrebbe azzerare immediatamente i dati sensibili memorizzati, come la password in chiaro dell'utente.

2.2.5 Modalità di compilazione di SGX

Intel SGX definisce quattro diverse modalità per compilare ed eseguire le applicazioni, Queste vengono utilizzate principalmente per eseguire l'applicazione in ambiente di test ed in ambiente di rilascio:

- modalità simulata: l'applicazione è costruita utilizzando delle librerie di simulazione che non generano un'enclave cifrata in memoria;
- modalità debug: le ottimizzazioni del compilatore sono disabilitate in modo da poter eseguire il codice dell'enclave in modalità di test;
- modalità release: l'applicazione è compilata con le ottimizzazioni attive ed il supporto al debug disabilitato;
- modalità pre-release: è una modalità simile a release, dove rimane il supporto al debug e le ottimizzazioni del compilatore sono disabilitate.

Attualmente l'SDK di valutazione scaricabile dal sito di SGX consente allo sviluppatore di creare ed eseguire enclavi utilizzando i profili di debug e pre-release. Le enclavi compilate sotto il profilo Release non possono funzionare finché lo sviluppatore non avrà ottenuto da Intel una licenza di produzione richiesto da SGX[47].

2.3 Tecnologie correlate a SGX

In questa sezione si introducono le principali alternative a Intel SGX. Altre compagnie importanti concorrenti di Intel come AMD e ARM hanno implementato modelli e soluzioni hardware per aggiungere strati di protezione per le applicazioni. In Sezione 2.3.1 si descrive la soluzione di ARM nominata TrustZone che utilizza una nuova funzionalità del processore per dividere l'esecuzioni dei programmi in

[47] https://software.intel.com/content/www/us/en/develop/blogs/intel-sgx-debug-production-prelease-whats-the-difference.html

due ambienti hardware separati: sicuro e non sicuro. In Sezione 2.3.2 si descrive la soluzione AMD Secure Encrypted Virtualization che consente la cifratura della memoria per le singole macchine virtuali ospitate dal cloud computing. Poi, in Sezione 2.3.3 si descrive Intel Trusted Execution Technology (TXT), un'altra tecnica sviluppata precedentemente da Intel per la protezione del software. Infine, si descrive il Trusted Platform Module (TPM), un microchip hardware responsabile delle funzioni di sicurezza della macchina, come definito in Sezione 2.3.4.

2.3.1 ARM TrustZone

ARM TrustZone[48] è una tecnologia che utilizza l'hardware per creare un isolamento tra due esecuzioni di codice distinte al fine di aumentare la sicurezza delle applicazioni sui sistemi ARM. Questa inserisce un'opzione di sicurezza a livello di sistema per i processori basati su ARM Cortex di architetture con versioni ARMv6KZ, ARMv8-M e successivi. L'idea di base è quella di utilizzare due hardware diversi per eseguire il codice sicuro e non sicuro separatamente. In questo modo i due contesti sono separati e non condividono risorse. Tuttavia, TrustZone non utilizza due hardware separati per le due esecuzioni ma definisce una nuova modalità del processore che utilizza un "Secure bit" per definire la modalità operativa del processo. Questo bit, non si limita alla CPU, ma si propaga sul bus di sistema a periferiche e controller di memoria. L'architettura Arm TrustZone crea quindi due ambienti separati a livello hardware che possono essere eseguiti contemporaneamente su un core: un ambiente sicuro e un ambiente non sicuro[49]. In tecnologia TrustZone i due ambiti sono nominati Rich Execution Environment (REE) e Trusted Execution Environment (TEE). Nell'ambiente REE vengono eseguiti i processi non sicuri di grandi dimensioni che non sono considerati affidabili. Invece, nell'ambiente TEE vengono eseguiti solo i processi che hanno bisogno di protezione come processi di gestione del sensore di impronti digitali, processi di pagamento e processi con accesso ad informazioni sensibili. Quando la modalità sicura è attiva, il software in esecuzione sulla CPU ha una visibilità diversa sull'intero sistema rispetto al software in esecuzione in modalità non sicura. In questo modo, le funzionalità del sistema che riguardano funzioni sensibili e segreti come le credenziali crittografiche, possono essere nascoste dall'ambiente non sicuro.

Come è visibile in Figura 2.9, la distinzione tra le due modalità è ortogonale alla normale protezione offerta dai diversi livelli di privilegio[50]. In questo modo la piattaforma basata su ARM TrustZone implementa due diversi sistemi hardware condividendo però le risorse fisiche. In questo modo è possibile eseguire due sistemi differenti sullo stesso hardware, utilizzando due diversi sistemi operativi per le due parti. In questo modo è possibile porre maggiore attenzione ai programmi del lato fidato riducendo al minimo le loro dimensioni e focalizzandosi sulle strategie software per ridurre le superfici di attacco ed aumentare l'affidabilità del sistema.

[48] https://developer.arm.com/ip-products/security-ip/trustzone

[49] https://genode.org/documentation/articles/trustzone

[50] https://static.ptsecurity.com/phdays/presentations/phdays-9-arm-trustzone-for-dummies.pdf

Allo stesso tempo è comunque possibile eseguire programmi non fidati nello stato non sicuro del sistema in modo che questi non abbiano accesso alle funzionalità ed ai dati sensibili dello stato TEE.

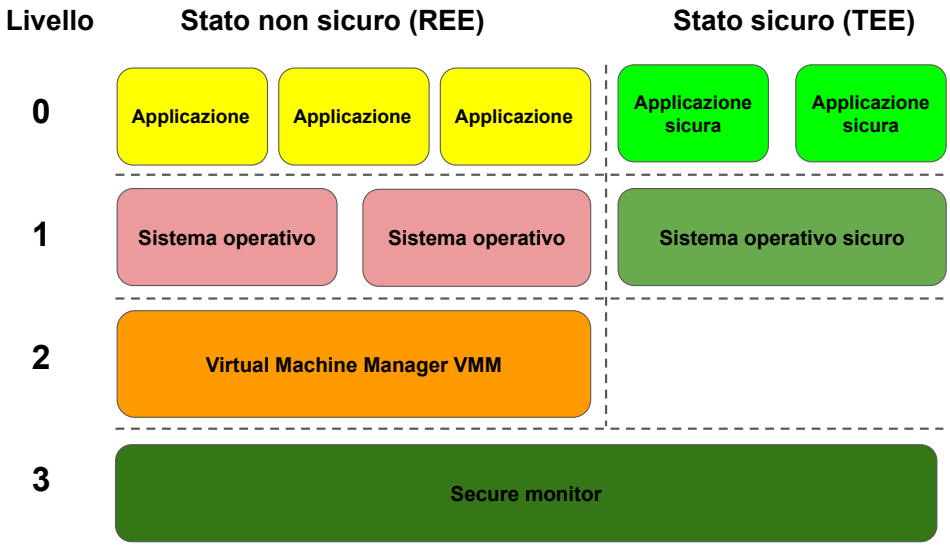


Figura 2.9: Gli stati REE e TEE ortogonali ai privilegi di processo.

2.3.2 AMD Secure Encrypted Virtualization (SEV)

I dati nella memoria RAM sono vulnerabili agli attacchi di software privilegiati e agli attacchi hardware effettuati dal gestore del sistema poiché le informazioni vengono memorizzate in chiaro. La tecnologia delle memorie RAM non volatili peggiora questo aspetto di sicurezza, poiché tali memorie possono essere fisicamente rimosse dal sistema senza alterare i dati contenuti al suo interno. Di conseguenza, le memorie possono essere lette da altri sistemi che possono intercettare le informazioni non cifrate memorizzate in memoria come dati sensibili, password e chiavi private.

Secure Encrypted Virtualization è funzione aggiunta all'architettura AMD progettata per affrontare meglio le esigenze di complessità e isolamento dei moderni sistemi[51]. Questa funzionalità presente sui processori AMD è un'estensione dell'architettura AMD-V che supporta l'esecuzione di macchine virtuali (VM) sotto il controllo di un ipervisore. SEV nasce con l'obbiettivo di proteggere le macchine virtuali cifrando la memoria principale di ogni macchina con una chiave univoca[52], come mostrato in Figura 2.10. Le chiavi di cifratura vengono direttamente memorizzate all'interno del processore AMD.

[51] https://developer.amd.com/sev/

[52] https://www.kernel.org/doc/html/latest/virt/kvm/amd-memory-encryption.html

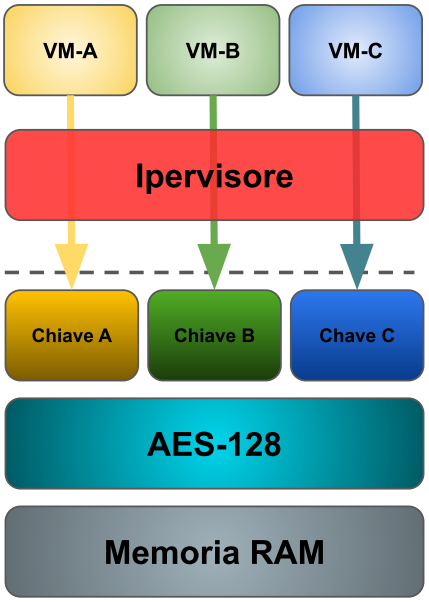


Figura 2.10: Architettura delle macchine virtualizzate con SEV.

SEV migliora quindi l'isolamento attraverso l'uso della cifratura di codice e dati e consente un modello di sicurezza completamente nuovo in cui il codice può essere protetto da livelli superiori come il codice privilegiato e l'hypervisor.

La nuova architettura rappresenta quindi un nuovo paradigma di sicurezza della virtualizzazione particolarmente adatto per il cloud computing poiché le macchine sono ospitate su server remoti che non possono essere direttamente controllati dai proprietari delle VM. AMD Secure Encrypted Virtualization inserisce quindi una nuova protezione tra la macchina virtuale ed i processi con alti privilegi di sistema e fornisce degli strumenti al proprietario della macchina per verificare la corretta abilitazione della protezione [13]. Infatti SEV permette di calcolare una firma del contenuto della memoria, che può essere inviata al proprietario della VM come attestazione riguardante la corretta cifratura della memoria.

Per utilizzare questa tecnologia non sono necessarie modifiche al software applicativo, infatti per l'abilitazione di SEV si utilizzano le configurazioni della macchina virtuale per selezionare quali parti di memoria debbano essere cifrate[53].

2.3.3 Intel Trusted Execution Technology (TXT)

Intel Trusted execution technology [14] (TXT), è una tecnologia dei processori Intel Xeon 5600 e successivi che ha come obbiettivo la protezione del software, attestando l'autenticità della piattaforma in uso, verificando che il sistema operativo venga eseguito in un ambiente sicuro e assicurando che il sistema protetto sia affidabile.

[53] https://documentation.suse.com/sles/15-SP1/html/SLES-amd-sev/index.html#sec-amd-sev-intro

Intel TXT si affida a una serie di componenti hardware (come il TPM [15]), software e firmware avanzati progettati per proteggere le informazioni sensibili dagli attacchi basati su software. Infatti, questo garantisce una protezione da software eseguiti con privilegi elevati e fornisce le proprietà di integrità, confidenzialità, affidabilità e disponibilità del sistema.

A differenza di Intel SGX, il quale fornisce una protezione solo al codice eseguito in regioni protette chiamate enclavi, come descritto in Sezione 2.1, Intel TXT provvede una garanzia di protezione fin dal boot del sistema. Infatti, i sistemi compatibili eseguono il proprio boot attraverso il Measured Launch Environment (MLE) che consente di controllare i singoli elementi responsabili dell'inizializzazione della macchina. Questi elementi vengono controllati attraverso un identificatore crittografico univoco, calcolato in precedenza, in modo tale da poter bloccare l'esecuzione di codice non conforme al codice approvato[54].

Gli identificatori sono memorizzati nella memoria non volatile del componente hardware di sicurezza TPM. Quando si esegue il boot del sistema, i componenti da eseguire in sequenza, come il BIOS e il sistema operativo, sono misurati al fine di confrontare i singoli risultati con i valori memorizzati nel TPM. In caso di modifica di un componente, il valore calcolato al boot del sistema risulterà differente a quello memorizzato in precedenza e di conseguenza lo stato dell'ambiente verrà considerato non sicuro. In caso contrario, una corrispondenza dei valori garantirà l'integrità e la sicurezza del componente. Questa analisi viene effettuata per ogni componente in esecuzione sul sistema[55].

2.3.4 Trusted Platform Module (TPM)

Il TPM [15] è un microchip hardware che fornisce le funzioni della macchina responsabili della sicurezza informatica. Questo componente nasce dalle specifiche del Trusted Computing Group (TCG)[56] nell'anno 2003, con lo scopo di migliorare la sicurezza della piattaforma.

Il TCG ha definito l'insieme minimo di algoritmi, operazioni e proprietà di sicurezza che devono essere offerte dal TPM, tra cui la compatibilità con [16]:

1. RSA: cifratura asimmetrica con chiavi di lunghezza 512, 1024 e 2048 bit;
2. SHA-1: algoritmo di hash a 160 bit;
3. HMAC: con chiave a 160 bit e dimensione blocco 64 byte, riferito allo standard RFC 2104 [17].

[54] https://software.intel.com/content/www/us/en/develop/articles/intel-trusted-execution-technology-intel-txt-enabling-guide.html#_Toc383534384

[55] https://www.intel.la/content/www/xl/es/architecture-and-technology/trusted-execution-technology/trusted-execution-technology-security-paper.html

[56] https://trustedcomputinggroup.org/resource/tpm-library-specification/

Per fornire queste funzionalità il modulo TPM deve essere fornito di diverse componenti, tra cui un coprocessore crittografico capace di eseguire le operazioni di cifratura e decifratura asimmetrica (RSA), la generazione di numeri casuali, la generazione di chiavi RSA e l'esecuzione delle operazioni di hash con SHA-1. Altro componente fondamentale di TPM è il gestore di input/output (I/O), il quale si occupa di gestire, codificare e dirigere i flussi di dati tra le comunicazioni sui bus interni ed esterni. Infine, gli altri componenti importanti di TPM sono:

1. gestore di alimentazione in grado di informare il modulo sullo stato di alimentazione dei dispositivi della macchina;
2. Opt-in che fornisce le funzionalità per accendere, spegnere, attivare e disattivare l'intero modulo TPM;
3. memoria non volatile deputata alla conservazione sicura delle informazioni sensibili.

L'utilizzo del TPM è associata ad un insieme di componenti software affidabili denominate Trusted Software Stack (TSS). L'utilizzo di entrambe le tecnologie TPM e TSS forniscono le operazioni di attestazione dell'identità, come l'attestazione remota, il data sealing[57] e il data binding. Quest'ultima, è una tecnica che consiste nel cifrare i dati conservati in memoria con una chiave che dipende dallo stato hardware e software del sistema. In questo modo la decifratura potrà essere effettuata solo utilizzando la chiave proveniente dalla stessa configurazione del sistema al momento del salvataggio dei dati.

[57]Il data sealing è l'operazione di cifratura delle informazioni inviate sui bus di sistema.

Capitolo 3

Stato dell'arte: analisi del codice

In questo capitolo si descrive lo stato dell'arte dell'analisi del codice. In particolare si definiscono i due approcci di analisi del codice, statica e dinamica, e se ne definiscono gli utilizzi. Inoltre, in Sezione 3.1 vengono definiti alcuni strumenti di analisi statica del sorgente, evidenziandone le capacità. Infine, In Sezione 3.2 si approfondiscono alcuni strumenti di analisi dinamica del codice che raccolgono informazioni durante l'esecuzione del programma.

3.1 Analisi statica

Con l'analisi statica del codice si intende l'insieme di procedure e calcoli che si eseguono automaticamente per analizzare il testo di un codice sorgente senza far eseguire il programma. Questa analisi utilizza proprietà sintattiche e regole del linguaggio per trovare eventuali errori locali. Alcuni analizzatori di codice utilizzano regole di semantica del linguaggio per comprendere in parte il flusso del codice. In genere questi costruiscono un modello che rappresenta il codice e poi analizzano il modello per ricavarne informazioni.

I risultati saranno poi trasformati in avvisi se errori o `code smell` verranno rilevati. Il code smell nel codice non indica necessariamente un errore ma rivela una cattiva condotta di programmazione o una vulnerabilità che può ridurre l'indice di qualità del codice. Tipici esempi di code smell sono: il codice duplicato; le costanti magiche[1]; il dead code[2]; le funzioni o i costrutti troppo complessi.

Nelle successive sezioni si introducono alcuni strumenti di analisi statica del codice e se ne descrivono le principali funzionalità. Infatti, in Sezione 3.1.1 si descrive lo strumento di analisi del codice Ctags, in grado di riassumere le definizioni presenti (e.g. funzioni, strutture dati, metodi) nei file sorgente in 118 diversi linguaggi di

[1]Con costanti magiche si intende l'insieme di valori costanti memorizzati nel codice sorgente con funzionalità implicita (o non documentata). Questi valori, se modificati, cambiano la funzionalità del codice o lo rendono non funzionante. È consigliabile memorizzare questi valori in un file di configurazione documentato.

[2]Il dead code è l'insieme di parti di codice presenti nel codice del programma ma mai eseguite.

programmazione. Poi, la Sezione 3.1.2 descrive Frama-C, uno strumento più avanzato in grado di analizzare i codici scritti in linguaggi C e C++. In Sezione 3.1.3 invece, si approfondisce SonarQube, un analizzatore capace di individuare bug e vulnerabilità presenti nel codice in diversi linguaggi di programmazione. Infine, la Sezione 3.1.4 descrive lo strumento di analisi CppCheck, in grado di evidenziare errori e falle nei programmi in linguaggio C e C++.

3.1.1 Ctags

Ctags[3] è un programma portatile eseguibile su linea di comando che genera un file testuale il quale riassume un codice sorgente in input. In questo file di `tag` è possibile trovare una rappresentazione a tabella che esprime informazioni su funzioni, variabili, metodi, classi, macro e definizioni varie a seconda del linguaggio di programmazione in uso. Questo tipo di file può essere letto manualmente come file di testo o utilizzato con diversi editor compatibili[4], come Atom, CodeLite, Emacs, Vim, Gedit e Notepad++, che ne traggono vantaggio e aiutano l'utente consapevole a navigare tra i molteplici file sorgenti.

In alternativa è anche possibile specificare opzioni dello strumento in modo che l'output sia più comprensibile dall'utente rispetto ad un linguaggio macchina. In questo caso non sarà neanche necessario generare un file, ma l'utente potrà leggere il risultato direttamente sulla console dei comandi.

Nel caso della modalità standard invece, il file di output è nominato `tags` per impostazione predefinita ed è possibile ridirigere l'output sia su un file con diverso nome che su standard output. Nell'impostazione base di Ctags, questo formato consiste in una lista di linee, su cui ogni elemento è descritto.

```
{tagName}<Tab>{tagFile}<Tab>{tagAddress}
```

Figura 3.1: Formato del file di tag.

Come visibile in Figura 3.1, ogni linea contiene: `tagName`, il nome identificativo dell'elemento, come il nome di una funzione o di una definizione; `Tab`, un carattere di tabulazione generalmente identificato con `\t`; `tagFile`, il percorso del file dove è stato definito l'identificativo dell'elemento, relativamente alla cartella corrente; `tagAddress`, la linea in cui si trova la definizione.

Le linee di tag sono ordinate alfabeticamente in modo predefinito, per rendere più veloce un'eventuale ricerca ma è anche possibile ordinarle per ordine di apparizione nel file. Il programma consente[5] di filtrare i dati per tipo (come strutture

[3] https://github.com/universal-ctags

[4] http://ctags.sourceforge.net/tools.html

[5] https://docs.ctags.io/en/latest/man-pages.html

dati, funzioni e metodi) e di personalizzare il livello di dettaglio fornito, come abilitare informazioni sui parametri delle funzioni, sulle proprietà di variabili (`static`, `volatile`) e informazioni implementative.

In Figura 3.2 sono mostrati il contenuto di un file in linguaggio C e il risultato in forma tabulare dell'esecuzione di Ctags. Come è possibile notare, lo strumento riconosce il nome delle entità e fornisce informazioni utili alla ricerca, come il nome del file di riferimento e la riga della definizione. In ultimo, è mostrato il tipo di entità, in particolare `d`, `f` e `S`, rispettivamente per definizione `#define`, funzione e struttura dati.

In principio questo programma è stato introdotto in BSD Unix ma è ad oggi disponibile per entrambi i sistemi Linux e Windows con la versione `Universal Ctags`[6]. Questa garantisce la compatibilità[7] con diversi tipi di file e linguaggi di programmazione sia basso livello che ad oggetti come: C, C++, Java, Javascript, HTML, Tcl e Python.

```
1  #include <stdio.h>
2  #define MAX 100
3
4  struct myT {
5          ...
6          int a;
7  };
8
9  int foo(char *buf){
10         ...
11 }
12
13 int main() {
14         ...
15         return 0;
16 }
```

(a) Contenuto del file.

Nome	*Percorso*	*Riga di inizio*	*Tipo*
MAX	file.c	2	define
foo	file.c	9	function
main	file.c	13	function
myT	file.c	4	struct

(b) Rappresentazione in output dell'elaborazione.

Figura 3.2: utilizzo di Ctags.

3.1.2 Frama-C

Frama-C[8] è un framework modulare utilizzato per eseguire analisi statica di codice C. Il programma offre una versione eseguita su linea di comando e comprende

[6] https://github.com/universal-ctags/ctags

[7] http://docs.ctags.io/en/latest/parsers.html

[8] https://frama-c.com/

una versione con interfaccia grafica con funzionalità a più alto livello. Queste sono installabili sui sistemi operativi Linux e Windows secondo i diversi metodi di installazione disponibili[9].

Frama-C ha un'architettura plugin modulare, che consente agli sviluppatori più esigenti di creare nuove interfacce per estendere la funzionalità dello stesso[10]. Ogni plugin dello strumento consente di effettuare diversi tipi di operazioni ed analisi e molti sono già disponibili nel pacchetto di installazione.

I plugin più utilizzati hanno scopi anche molto diversi tra loro, che passano dall'analisi di valori di una variabile nel programma sino a rilevare codice inutile nel programma. La peculiarità di Frama-C è l'approccio collaborativo con cui diversi plugin possono collaborare per analizzare e modificare il codice a seguito di trasformazioni risultanti da uso di plugin in serie. Il plugin comunicano in `pipeline`, ovvero sono collegati in cascata e sono eseguiti in serie in modo che l'output di un comando sia l'input diretto del comando immediatamente successivo. Si analizzano quindi le funzionalità dei plugin più utilizzati[11].

Impact analysis Quest'analisi consente di evidenziare le parti di programma che sono affette da delle modifiche.

Slicing Lo `slicing` consente di generare un programma a partire da un codice sorgente creandone una copia e mantenendo alcune parti di codice che soddisfano delle proprietà. Per esempio, si mantengono le parti di codice che leggono e modificano un insieme di variabili e si genera un programma con funzionalità ridotte.

Spare code Questa operazione consiste nel rimuovere le parti di codice che non influiscono sul risultato finale del programma.

Metric Questa analisi genera un report di informazioni sul flusso del programma e sulle singole funzioni. Per esempio, viene analizzata la raggiungibilità di una funzione, ovvero la lista di funzioni da cui è raggiungibile e vengono riassunte alcune proprietà descrittive di una funzione, come il numero di cicli, di istruzioni di salto, di chiamate a funzione, di punti di terminazione, di costrutti decisionali e di accesso ai puntatori.

Variable occurrence Questa operazione consente di visualizzare l'accesso ad una variabile scelta per verificare quali parti di codice ne leggono e ne scrivono il valore.

E-ACSL Il plugin consente di dimostrare le proprietà formali sul codice con l'uso di specifiche scritte in linguaggio E-ACSL[12]. Il programmatore modifica il codice sorgente, aggiungendo delle annotazioni scritte nel linguaggio citato. Poi, il

[9] https://frama-c.com/download.html

[10] https://frama-c.com/download/frama-c-plugin-development-guide.pdf#page=45

[11] https://frama-c.com/plugins.html

[12] https://frama-c.com/download/e-acsl/e-acsl.pdf

plugin trasforma le annotazioni in dei predicati che vengono valutati durante l'esecuzione. Il codice non varia di comportamento se i predicati vengono rispettati. In caso contrario, il sorgente genera un errore che specifica il tipo di guasto e la linea di codice che non rispetta la regola.

Callgraph Questa operazione calcola il grafo delle chiamate a partire da una funzione iniziale. L'analisi può essere svolta sia in maniera sintattica che semantica, supportando quindi i puntatori a funzione e migliorando l'accuratezza del risultato attraverso l'analisi del valore delle variabili e del valore dei predicati.

3.1.3 SonarQube

SonarQube[13] è uno strumento di analisi statica del codice utilizzato per la verifica della qualità del codice. Questo strumento è in grado di analizzare il codice in 27 linguaggi di programmazione differenti. L'analisi effettuata comunica allo sviluppatore gli errori comuni, le vulnerabilità e le cattive condotte rilevati nel sorgente. Poi, lo strumento fornisce un indice qualitativo del sorgente al fine di poter confrontare le diverse versioni del programma e mantenere una qualità elevata del codice. Questo strumento viene scaricato ed eseguito sulla macchina attraverso la linea di comando o utilizzando un'immagine Docker[14]. Viene quindi eseguito un server in locale e si fornisce allo sviluppatore un'interfaccia grafica sul browser. Attraverso quest'ultima è possibile creare un nuovo progetto, caricare il codice sorgente dell'applicazione ed analizzarlo.

Si analizzano quindi le funzionalità offerte da SonarQube.

Bug Lo strumento fornisce un'analisi statica in grado di rilevare bug del codice. Per esempio, vengono rilevate le incongruenze delle espressioni booleane quando si confrontano tipi differenti. A seconda del linguaggio di programmazione vengono anche forniti consigli su come migliorare il codice e rispettare gli standard, come per esempio la validazione dei documenti HTML[15].

Code smell I code smell rilevati da SonarQube sono tutti i costrutti che rendono il codice più confuso e di conseguenza più difficile da comprendere e mantenere. Alcuni esempi rilevati sono il codice duplicato, il codice non coperto da test ed il codice molto complesso.

Test coverage L'analisi del software si dedica anche ai sorgenti di test. Lo strumento è in grado di calcolare la percentuale di copertura dei test e fornire un indice di qualità anche per questo scopo.

Security Hotspots SonarQube analizza il codice e ne ricerca le vulnerabilità di sicurezza. Lo strumento rileva le parti di codice vulnerabili che possono inserire dei rischi, come la non validazione degli input provenienti da rete o l'utilizzo di tecniche di cifratura obsolete.

[13] https://www.sonarqube.org/

[14] https://www.docker.com/

[15] https://validator.w3.org/

In Figura 3.3 è rappresentata la schermata di SonarQube che mostra la panoramica di un progetto analizzato. Nell'interfaccia vengono mostrate statistiche di alto livello su bug, vulnerabilità e rischi per la sicurezza. Inoltre, viene rappresentato il `debt`[16] misurato in tempo necessario per modificare il codice e migliorarlo. Poi, vengono raffigurati i code smell, indicando un indice che rappresenta quanto il codice è facile da mantenere. Infine, sono rappresentati i dati sulla copertura dei test e sulla duplicazione del codice.

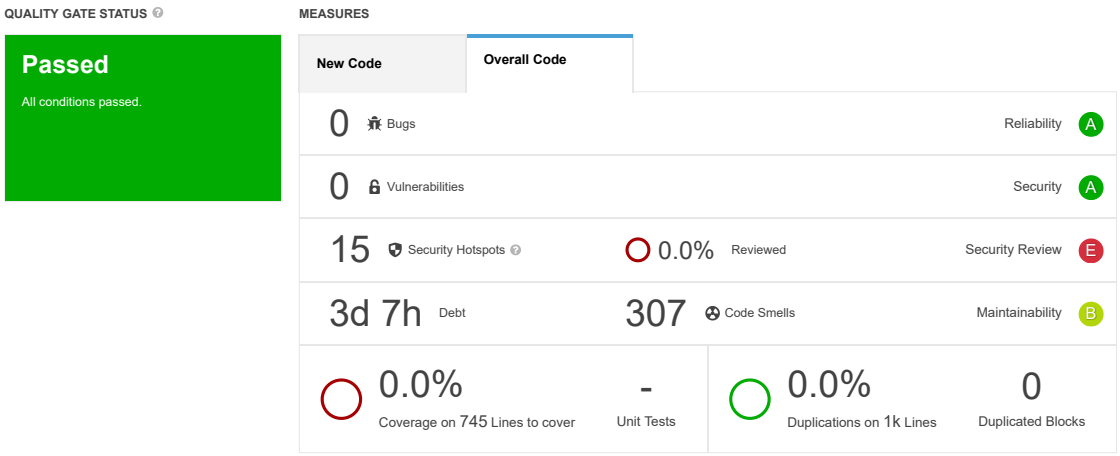


Figura 3.3: Interfaccia web di SonarQube.

3.1.4 Cppcheck

Cppcheck[17] è uno strumento di analisi statica del codice capace di analizzare il codice nei linguaggi C e C++. Questo software gratuito è disponibile per i principali sistemi operativi Linux, Windows e macOS. Cppcheck è ottimizzato per ricercare bug a livello di sorgente quali:

- uso di puntatori non inizializzati o nulli;
- divisioni per zero;
- overflow[18] di numeri interi;
- operazioni invalide di spostamento di bit;
- conversioni di tipo invalide;
- uso di puntatori dopo la loro de-allocazione;
- uso di indici esterni ai limiti dei vettori;

[16] Il technical debt o debito tecnico è l'insieme di errori, bug, e vulnerabilità raccolte nel tempo durante lo sviluppo del software.

[17] `http://cppcheck.sourceforge.net/`

[18] L'overflow di una variabile numerica è dovuto al suo utilizzo improprio per memorizzare un numero esterno all'insieme di valori rappresentabili.

- uso di variabili non inizializzate;
- scrittura su dati costanti;
- perdita di memoria a seguito di allocazione;
- perdita di risorse a seguito di apertura file e socket di rete.

Lo strumento si integra con alcuni IDE[19] utilizzati per lo sviluppo di applicazioni C e C++, come CLion[20], Code::Blocks[21], Eclipse[22], Visual Studio[23] ed Emacs[24].

3.2 Analisi dinamica

L'analisi dinamica del programma viene svolta durante l'esecuzione di un programma. Per poter risultare efficace, tale analisi va ripetuta più volte utilizzando un insieme di dati sufficientemente ampio a percorrere la maggior parte delle linee di codice e ricevere in output quanti più valori possibili. Gli utilizzi comuni di questo tipo di analisi sono:

- calcolare la copertura del codice[25] di una test suite[26];
- rilevare gli errori di gestione della memoria, come il leakage di memoria[27];
- trovare i bug nel codice ed analizzare i dati del gusto, come il nome di eccezioni o il nome della funzione che causa il guasto;
- trovare errori di concorrenza, ad esempio situazioni di race condition[28];
- analizzare le prestazioni.

[19] Gli Integrated Development Environment (IDE) sono i software che comprendono le funzionalità utili allo sviluppo del codice, come la apertura di file per lettura e scrittura, la compilazione e l'operazione di debug.

[20] https://www.jetbrains.com/clion/

[21] http://www.codeblocks.org/

[22] https://www.eclipse.org/

[23] https://visualstudio.microsoft.com/it/vs/

[24] https://www.gnu.org/software/emacs/

[25] La copertura del codice è la percentuale di istruzioni eseguite durante i test.

[26] Una test suite è una collezione di unità di test focalizzate a verificare il comportamento ed il corretto funzionamento delle componenti software di un programma.

[27] La mancata deallocazione delle strutture dati non più necessarie comporta una perdita di spazio nominata leakage di memoria.

[28] La race condition è una condizione delle applicazioni concorrenti in cui il risultato finale è impredicibile e dipende dall'ordine e dal tempo di esecuzione delle parti concorrenti.

Questi tipi di analisi richiedono l'esecuzione del programma per poter essere valutati e di conseguenza non possono essere facilmente eseguiti con l'analisi statica. Si descrivono quindi alcuni strumenti di analisi dinamica del codice. In particolare la Sezione 3.2.1 introduce lo strumento Valgrind che permette un'analisi approfondita sull'utilizzo della memoria in esecuzione. Poi, in Sezione 3.2.2 si descrive Gcov, uno strumento utilizzato per verificare la copertura dei test e la frequenza di utilizzo delle istruzioni. Infine, in Sezione 3.2.3 si definisce l'analizzatore dinamico VTune, il quale fornisce il campionamento dell'esecuzione del programma al fine di offrirne informazioni sulle prestazioni.

3.2.1 Valgrind

Valgrind[29] è un framework per la creazione di strumenti di analisi dinamica. Questo è nato inizialmente come strumento di debug della memoria ma evolvendosi nel tempo ha acquisito nuove funzionalità portate sia da strumenti integrati che esterni. Il funzionamento di Valgrind si basa su una rappresentazione intermedia (IR) che non dipende dal processore. Quando viene eseguito, Valgrind trasforma il programma in questa rappresentazione intermedia. Poi, gli strumenti, abilitati dalle opzioni di esecuzione del framework, trasformano il modello IR inserendo nuovo codice che permette di monitorare l'esecuzione del programma. Infine, Valgrind ricompila il nuovo programma per eseguirlo sul processore della macchina e lo apre in esecuzione. Le trasformazioni, eseguite dai plugin di Valgrind, degradano le prestazioni del codice durante l'esecuzione. Per esempio l'esecuzione dello strumento `Nulgrind`[30] non applica alcuna trasformazione ma l'esecuzione con Valgrind degrada le prestazioni rallentandole di circa 5 volte rispetto all'esecuzione nativa.. La degradazione delle prestazioni dipende comunque dagli strumenti abilitati e può raggiungere un rallentamento con fattore 40 rispetto all'esecuzione nativa, con lo strumento `BBV`.

Lo strumento base più utilizzato è `Memcheck`[31] il quale fornisce un'interfaccia per rilevare accessi invalidi in memoria come: l'uso di memoria non inizializzata, accesso a memoria rilasciata con l'istruzione `free`, accesso all'esterno di blocchi allocati e perdita di memoria. L'esecuzione di `Memcheck` abbatte le prestazioni di esecuzione di circa 30 volte[32].

Altri strumenti molto utilizzati sono `DRD` e `DHAT`. Il primo è un plugin che riesce a rilevare gli errori sulle applicazioni multi thread, come la race condition. Invece, lo strumento `DHAT` fornisce un report dettagliato sull'utilizzo delle risorse da parte di un programma, come la memoria utilizzata, l'uso della CPU e l'utilizzo della banda di rete.

[29] https://valgrind.org/

[30] https://www.valgrind.org/docs/manual/nl-manual.html

[31] https://valgrind.org/docs/manual/mc-manual.html

[32] https://valgrind.org/docs/manual/quick-start.html#quick-start.mcrun

3.2.2 Gcov

Gcov[33] è uno strumento adatto ad analizzare la copertura del codice fornita dai test. Inoltre, le ulteriori statistiche fornite dal programma sono utilizzate per scoprire quali parti del programma influiscono maggiormente sulle prestazioni e di conseguenza quali parti necessitano di ottimizzazione. Le informazioni utili a questo scopo sono: la frequenza di esecuzione di ogni riga di codice, il numero di esecuzioni di ogni istruzione e il tempo di esecuzione di ogni sezione di codice.

Questo strumento funziona solo con i programmi compilati con GCC e prevede l'integrazione con un altro strumento di analisi dinamica, `Gprof`, utilizzato per generare il grafo delle chiamate a funzione eseguite.

```
#include <stdio.h>

  void foo(char *buf){
  putchar(*buf);
}

int main() {
  char *t = "13 caratteri\0";
  int i=0;
  for(; i<13;i++)
    foo(t+i);

  return 0;
}
```

(a) Contenuto del file sorgente.

```
#include <stdio.h>

13 void foo(char *buf){
13 putchar(*buf);
13 }

1 int main() {
1  char *t = "13 caratteri\0";
1  int i=0;
14  for(; i<13;i++)
13   foo(t+i);

1  return 0;
}
```

(b) Contenuto del file annotato.

Figura 3.4: Utilizzo di Gcov.

L'utilizzo di Gcov inizia con la compilazione del binario con le opzioni necessarie specificate, come la disattivazione delle ottimizzazioni e l'aggiunta delle istruzioni per il conteggio della copertura. Viene quindi eseguito il binario, in modalità standard, il quale genera un file di informazioni. Quest'ultimo viene aggiornato ad ogni esecuzione del programma compilato in modo tale da poter accumulare i dati di più flussi, eseguiti con dati in input differenti. Infine, viene eseguito lo strumento Gcov che apre il file appena generato, legge le informazioni riportate e genera un file sorgente annotato con il numero di esecuzioni delle singole istruzioni.

In Figura 3.4 sono rappresentati i file sorgente ed il file annotato dopo l'esecuzione di Gcov. Come è possibile notare in Figura 3.4b, il file annotato contiene il numero di esecuzioni di ogni istruzione.

Alla fine dell'esecuzione di Gcov, vengono descritte le informazioni riguardo le esecuzioni dell'eseguibile, come il nome del file sorgente annotato, la percentuale di

[33] https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

copertura del sorgente, il totale delle linee di codice, ed il nome del file sorgente di partenza.

3.2.3 VTune

VTune[34] è uno strumento di analisi dinamica del codice compatibile con 8 differenti linguaggi di programmazione. Questo può essere eseguito sia su linea di comando sia da interfaccia grafica ed è disponibile per i principali sistemi operativi Windows, Linux e macOS. L'interfaccia grafica è in grado di mostrare i diversi thread in esecuzione e fornire informazioni sullo stack delle chiamate a funzione. Inoltre, lo strumento è in grado di analizzare il tempo di utilizzo della CPU diviso per thread, in modo da mostrare l'efficienza d'uso dell'hardware a disposizione, il tempo in attesa sui metodi di sincronizzazione[35] delle variabili condivise e l'uso dei diversi core della CPU.

In questo modo è possibile analizzare l'esecuzione del programma ed individuare le parti di codice che portano a problemi di prestazioni. A tale scopo, lo strumento campiona l'esecuzione del programma e acquisisce anche informazioni di basso livello come i colpi di clock del processore, le istruzioni macchina eseguite e gli accessi alla cache del processore.

Dopo l'installazione del software, l'interfaccia grafica dello strumento consente di generare un nuovo progetto di analisi e caricare al suo interno il codice sorgente. Poi, è possibile eseguire l'analisi sul programma in esecuzione in modo tale da generare le informazioni sulle prestazioni del programma. Lo strumento consente anche di confrontare i risultati delle diverse versioni del programma al fine di mantenere traccia delle ottimizzazioni implementate dal programmatore.

[34] https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html

[35] Con metodi di sincronizzazione, si intendono i metodi con cui si gestisce l'accesso alle variabili. I principali strumenti utilizzati per la sincronizzazione tra processi sono: condition variable, semafori, lock e monitor.

Capitolo 4

Architettura

In questo capitolo viene descritta l'architettura del framework analizzando i moduli interni, le loro connessioni e funzionalità. In Sezione 4.3 verrà posta attenzione ad alcuni limiti nell'attuale implementazione del framework.

Il framework è in grado di analizzare e modificare del codice sorgente in linguaggio C, opportunamente annotato, per renderlo compatibile con la tecnologia Intel SGX. La funzionalità cardine del framework è proteggere parti di codice sensibili dal punto di vista della sicurezza, spostandole in una zona di memoria protetta chiamata enclave.

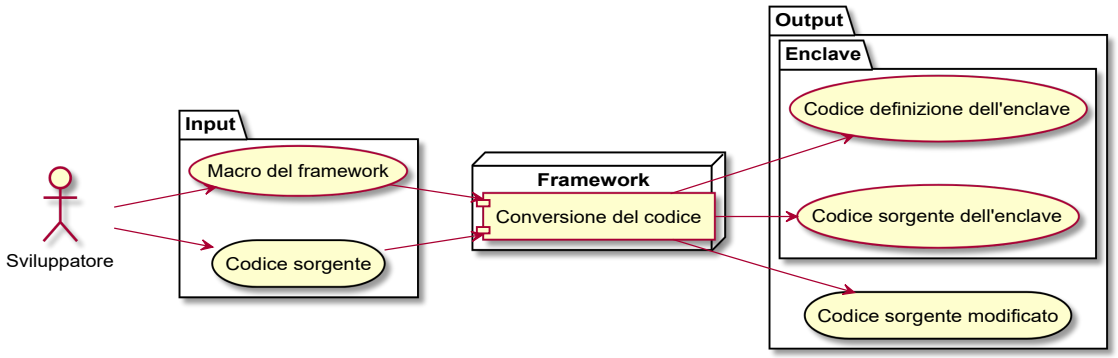


Figura 4.1: File convertiti e generati dal framework.

Lo sviluppatore annota il codice originale con delle macro che permettono al framework di individuare le funzioni da proteggere. Poi, si esegue il framework che modifica automaticamente il codice ed esporta in uscita i file originali modificati ed i file relativi all'esecuzione protetta.

Il framework è interamente scritto in linguaggio Python con versione 3.8.1 e utilizza codice aggiuntivo come gli analizzatori statici Ctags e Frama-C ed il compilatore GCC per decidere automaticamente le operazioni da svolgere ed adattare il codice. Lo strumento di conversione è compatibile con entrambi i sistemi Linux e Windows, a patto che siano installate le medesime versioni degli strumenti di analisi statica che forniscono i comandi utilizzati dallo strumento di conversione come descritto in Appendice B.3.

4.1 Annotazioni del framework

Il framework necessita delle annotazioni dell'utente per due principali ragioni: definire le chiamate ECall ed OCall, e definire la modalità di passaggio dei buffer (Sezione 2.2.2) alla funzione chiamata ed eventualmente specificare la grandezza dei buffer passati come puntatori agli argomenti di queste chiamate. Come richiesto dalla sintassi EDL in Sezione 2.2.2, è necessario definire la dimensione dei buffer passati negli argomenti delle chiamate ECall ed OCall. Questo perché quando vengono invocate tali chiamate, le librerie interne di SGX responsabili della transizione tra l'enclave e l'ambiente esterno copiano i buffer passati come argomenti nella regione di destinazione. Questo è necessario per due principali ragioni: limitare la superficie di attacco come spiegato in Sezione 2.2.2; copiare i dati dei buffer cifrati in parti di memoria non cifrata in modo che il codice OCall possa leggere le informazioni ricevute. In alternativa alle copie dei buffer, è possibile passare i puntatori senza che questi vengano alterati dalle librerie interne di SGX. In questa modalità non è necessario specificare la dimensione del buffer, poiché non verrà effettuata la copia. L'utente quindi specifica quali funzioni siano da proteggere e quali invece siano da escludere dall'esecuzione protetta. Poi, per ogni puntatore passato ai parametri delle chiamate definite ECall o OCall, l'utente dichiara la necessità di fare la copia dei buffer ed in caso di copia, l'utente specifica la dimensione totale del buffer.

La sintassi EDL obbliga la specifica del modo con cui si passano i parametri per i puntatori singoli e vettori di dimensione statica. Le modalità di passaggio supportati sono:

1. `in`: il buffer esterno del chiamante viene copiato all'entrata della chiamata in un nuovo buffer allocato nella regione del chiamato;
2. `out`: il buffer interno del chiamato viene copiato al ritorno nel buffer esterno del chiamante;
3. `in,out`: il buffer esterno del chiamante viene copiato nel buffer interno alla chiamata ed al ritorno il buffer interno del chiamato viene copiato nel buffer esterno.
4. `user_check`: non esegue nessuna operazione e passa il puntatore alla chiamata.

4.1.1 Sintassi delle Macro

Le macro inserite dall'utente sono annotazioni proprietarie del framework che sfruttano direttive del codice C che non influiscono sulla normale compilazione dell'applicazione. In questo modo il codice sorgente annotato continua ad essere compilabile anche per hardware non compatibile con la tecnologia SGX.

Ogni macro di definizione ECall o OCall definisce in un'unica riga tre dati fondamentali: il tipo di chiamata ECall o OCall; il nome della funzione a cui si fa riferimento; il modo con cui si passano i parametri. La macro quindi viene definita

con la direttiva di compilazione `#define` che specifica una stringa e una definizione di argomenti. La stringa viene definita con la notazione `sgx_ecall_nomeFunzione` per le chiamate ECall ed è invece `sgx_ocall_nomeFunzione` per le chiamate OCall. La parte di definizione degli argomenti è invariata per entrambi i tipi di chiamate.

```
#define sgx_ocall_stampaTesto ([text, i, text_size])
int stampaTesto(char* text, int text_size){
       return printf("%.*s", size, text);
}

#define sgx_ecall_calc ([myNumbers, b], [myList, i, list_size],
   [myBuf, u])
int calc(float myNumbers[100], int *myList[10], int list_size,
   void *myBuf){
       ...
}
```

Figura 4.2: Esempio di definizioni di macro per chiamate ECall ed OCall.

La Figura 4.2 mostra un esempio di annotazioni del framework. La macro inizia con la direttiva di definizione e specifica in una sola stringa il tipo di chiamata ed il nome della funzione. Poi, tra parentesi tonde, si descrive una lista di argomenti da copiare ognuno inserito tra parentesi quadre. In ogni coppia di parentesi quadre si inseriscono due o tre argomenti, a seconda dell'opzione di copia selezionata. Il primo campo è il nome dell'argomento a cui si fa riferimento per dichiararne la dimensione. Il secondo campo è l'opzione di copia che può essere:

- `i`: corrisponde all'opzione `in` di SGX
- `o`: corrisponde all'opzione `out` di SGX
- `b`: corrisponde all'opzione `in,out` di SGX
- `u`: corrisponde all'opzione `user_check` di SGX

Per l'opzione `u` sono sufficienti solo due argomenti che vengono specificati con la notazione: `[nomeArgomento, u]`. Per le altre opzioni di copia invece va specificata anche la dimensione del buffer. In questo caso il valore della dimensione acquisisce diverso significato a seconda del tipo puntato:

1. se il puntatore è di tipo definito con dimensione calcolabile: la dimensione riferita al puntatore deve specificare il numero di elementi del vettore;
2. se il puntatore non ha tipo (e.g. `void *`) la dimensione specificata del buffer dichiara la dimensione da copiare in byte.

Questo diverso significato è deciso autonomamente dal framework analizzando il tipo puntato dal puntatore passato agli argomenti. Per questo scopo si utilizzano le opzioni `count` e `size` della sintassi EDL, come riportato in Sezione 2.2.2.

4.2 Descrizione dei blocchi architetturali

In questa sezione vengono approfonditi i moduli interni del framework. La Figura 4.3 rappresenta i moduli interni del framework, gli strumenti esterni, i file sorgenti ed i file generati. I tre moduli esterni sono GCC, Frama-C e Ctags. Questi sono utilizzati dal framework attraverso dei moduli di interfaccia (approfonditi nelle sezioni successive) che gestiscono il loro utilizzo intercettando gli eventuali errori ed interpretando i dati generati dagli strumenti. I file di codice sorgente vengono processati dagli strumenti esterni e modificati dall'apposito modulo di interfaccia definito in Sezione 4.2.1. Il file EDL descrittore dell'enclave viene generato dall'apposito modulo di interfaccia approfondito in Sezione 4.2.6. Infine, il modulo principale definito in Sezione 4.2.7, radice di tutto il framework, si occupa di gestire l'intero flusso di conversione e di definire funzioni di supporto utilizzate ad esempio per analizzare il codice sorgente e decidere l'insieme di funzioni da proteggere.

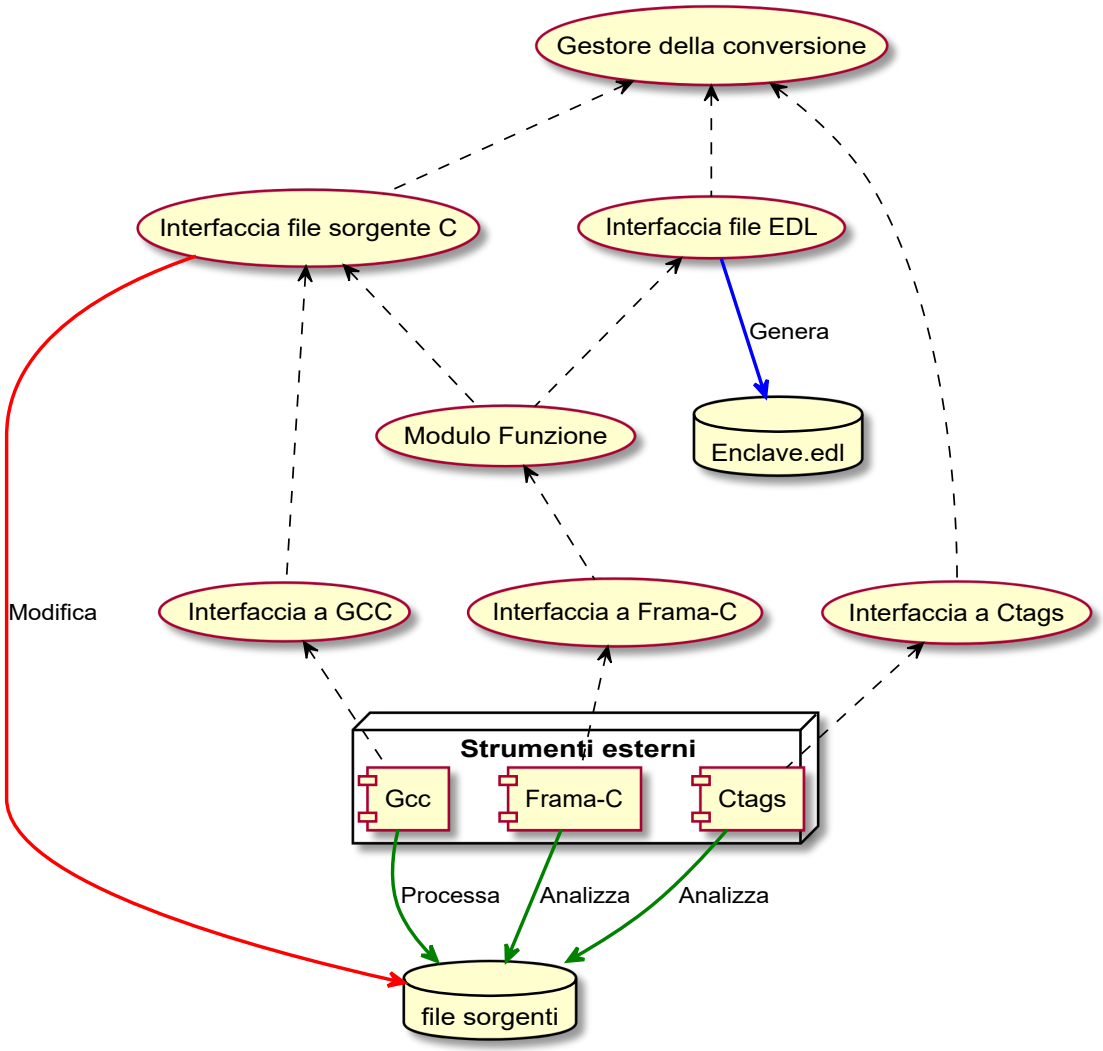


Figura 4.3: Moduli interni del framework, strumenti esterni e file. In tratteggio sono rappresentate le dipendenze software.

4.2.1 Interfaccia dei file sorgenti C

Questo modulo è capace di interagire con i file dell'utente con estensione `.c` e `.h`. Il modulo può essere istanziato a partire da un file preesistente oppure da un file nuovo. Nel caso di un file già esistente viene utilizzato il modulo di interfaccia allo strumento GCC (Sezione 4.2.4) per leggere tale file, preprocessarlo e rimuovere interamente tutti i commenti sul codice C. Una volta letto il file, il suo contenuto viene memorizzato e tutte le azioni eseguite tramite le funzioni di questo modulo modificano le variabili dell'istanza senza modificare il file. Infine, per poter salvare il contenuto e scriverlo sul file si utilizza la funzione apposita. Le funzioni disponibili offerte dall'interfaccia permettono di:

1. parsificare il contenuto;
2. riconoscere le ECall e le OCall;
3. gestire le funzioni: aggiungere, sostituire e rimuovere;
4. gestire le strutture dati: aggiungere e rimuovere.

Parsificare il contenuto

Con questa operazione si intende il gruppo di metodi in grado di riconoscere nel file le seguenti entità: strutture dati, unioni, enumerazioni, inclusioni di librerie, definizioni e funzioni. Questi vengono riconosciuti attraverso l'uso di espressioni regolari in grado anche di riconoscere le varie parti di un'entità come per esempio le parti di una definizione di funzione: valore di ritorno, nome, argomenti ed implementazione.

Riconoscere le ECall e le OCall

Per comprendere le ECall e le OCall è necessario conoscere la sintassi per le macro, la quale è definita nella Sezione 4.1.1. La sintassi è molto semplice, infatti ogni macro specifica il nome, il tipo di chiamata (ECall o OCall) e il modo di passaggio dei parametri alla funzione. Tuttavia, il modulo non controlla direttamente la correttezza della macro ma si limita a comprendere il nome delle funzioni a cui le macro fanno riferimento. Fatto questo, il modulo cerca nel suo contenuto gli oggetti rappresentanti le funzioni connesse alle macro ed invoca su tali istanze il metodo per settare i dati relativi alle macro; sarà quel metodo, l'incaricato a controllare la sintassi della macro. In questo modo si mantiene una divisione dei dati che semplifica i moduli. Una volta raccolte tutte le macro del proprio file, il modulo ritorna due liste di funzioni che rappresentano rispettivamente tutte le chiamate ECall ed OCall desiderate.

Gestire le funzioni

Il framework modifica il codice sorgente dell'utente cambiando il codice implementativo delle funzioni ed aggiungendone altre. Il modulo che gestisce il file sorgente implementa diversi metodi per gestire queste operazioni:

1. rimozione di funzione: è utile per rimuovere le funzioni spostate nell'enclave che quindi vanno eliminate nel codice sorgente non protetto;

2. aggiunta di funzione: è utilizzato per inserire all'interno dell'enclave la definizione dei wrapper alle chiamate OCall, i quali gestiscono gli errori della chiamata;

3. filtrare le funzioni per nome: è utilizzato durante l'analisi del codice dove si necessita di esplorare il codice sorgente e costruire il grafo delle chiamate;

4. modificare la definizione di una funzione: è utilizzato principalmente per cambiare l'implementazione originale di una funzione definita ECall in un wrapper alla vera funzione spostata nell'enclave.

4.2.2 Modulo Funzione

Il modulo raccoglie tutti i dati di una funzione come valore di ritorno, nome, argomenti e il codice implementativo. Questo modulo utilizza lo strumento esterno Frama-C, il quale si occupa di identificare le funzioni usate all'interno di altre funzioni. È possibile richiamare questo strumento senza ricorsione o anche in maniera ricorsiva ed è possibile specificare qual è il valore di ritorno desiderato. Come prima opzione il modulo identifica e ritorna il nome di tutte le funzioni raggiungibili da una funzione di partenza; questo è un insieme di nomi di funzioni che sono raggiunte per eseguire completamente la funzione di partenza. Come seconda opzione il modulo genera le coppie di funzioni legate da dipendenza software.

Un'altra operazione affidata a tali istanze è quella di controllare la sintassi della macro del framework nel caso la funzione sia definita ECall o OCall. I controlli effettuati sulla macro sono molteplici; inizialmente si divide la macro e si analizza ogni argomento definito. Per ogni argomento si controlla che la sintassi dell'argomento sia corretta e che comunque sia definito un argomento con lo stesso nome. Poi si controllano i limiti imposti dal framework e dalla sintassi EDL.

Riassumendo, le macro del framework specificano il tipo di chiamata ECall o OCall e il modo di passaggio dei valori referenziati dai puntatori. Come descritto in Sezione 4.1 le opzioni per il passaggio dei parametri sono copia `in`, `out`, `in/out` oppure l'opzione `user_check`. Il framework deve controllare che i dati inseriti dall'utente attraverso le macro siano sufficienti per generare la sintassi EDL corretta.

Infatti, nel caso di copia (`in, out, in/out`) se l'oggetto non è un puntatore e la dimensione è specificata si avrà un errore, poiché la dimensione dell'oggetto da copiare deve essere definita solo in presenza di puntatore singolo. Questo perché tale dimensione deve essere sommata ad un puntatore per ottenere l'indirizzo di partenza e di arrivo da copiare. Poi, se l'oggetto da copiare è un puntatore, ma la dimensione non è specificata, si avrà nuovamente errore. È sempre necessario specificare una dimensione con l'opzione di copia in presenza di puntatore singolo. In caso contrario la copia non può avvenire direttamente ed è da preferire l'opzione `user_check`, la quale ignora il puntatore e non crea una copia. Infine, se l'oggetto

da copiare è un puntatore multiplo la copia non può avvenire in modalità automatica. L'opzione `user_check` è l'unica opportunità a meno che non si definiscano delle matrici multidimensionali a dimensione fissa abbastanza grandi da contenere tali dati. Tuttavia, è sconveniente copiare grandi quantità di dati dentro e fuori dall'enclave per due principali ragioni: la prima è dovuta ad un problema di spazio poiché la dimensione dell'enclave è limitata e lo spazio di memoria protetta è condiviso tra le enclavi allocate nel sistema operativo, come spiegato in Sezione 2.1.6. La seconda consiste in un deterioramento delle perforance, infatti copiare grandi quantità di dati rallenta il programma e abbatte le prestazioni.

Invece, nel caso dell'opzione `user_check`, la dimensione non deve essere specificata poiché non verrà effettuata nessuna copia. In caso contrario sarà mostrato un errore sulla console.

Alcuni di questi controlli potrebbero essere gestiti in caso di errore ignorando i dati in eccesso o specificando le opzioni compatibili che più si prestano analizzando il tipo di dato ma si è scelto comunque di fermare l'esecuzione ed informare l'utente del problema riscontrato. Questa soluzione è stata preferita poiché l'utente che vuole proteggere il proprio codice deve farlo essendo consapevole dei limiti e dei vantaggi delle opzioni disponibili. È quindi richiesto che il programmatore conosca almeno queste semplici regole base che sono definite dallo standard SGX.

Successivamente ai controlli si può generare la sintassi SGX completa e corretta unendo le specifiche fornite dall'utente, la definizione degli argomenti della funzione e le regole dei file EDL dello standard SGX. Un esempio completo della conversione si trova in Figura 4.4.

Il modulo definisce anche due metodi molto importanti del framework. Il primo si occupa di dividere la funzione ECall in due funzioni: trusted e wrapper e ritornarle al chiamante. La prima di queste serve a gestire gli errori della chiamata ECall nel codice non protetto. La seconda invece è la funzione che verrà esportata nell'enclave con nome differente. Queste funzioni sono descritte nel dettaglio in Sezione 5.3.3.

Il secondo metodo si occupa di generare il codice wrapper alle chiamate OCall da inserire nell'enclave per gestire gli errori della chiamata, come spiegato in Sezione 5.3.3.

4.2.3 Interfaccia a Frama-C

Questo è il modulo Python responsabile dell'interfacciamento con lo strumento di analisi statica Frama-C. Il modulo si occupa di richiamare lo strumento esterno Frama-C per generare dei grafi di chiamate a funzione. Infatti, lo strumento esterno è in grado di calcolare un grafo di chiamate a funzioni e di rappresentarla su file di testo come spiegato in Appendice B.3.2.

Il modulo offre due funzioni principali molto utili nella costruzione di un grafo delle chiamate a funzione. Esse si basano sia sull'utilizzo di Frama-C per generare i

dati che sull'utilizzo del modulo esterno pydot[1] per leggerli. Quest'ultimo è in grado di leggere i file in formato `dot` e rappresentare nodi e archi in oggetti Python.

Le due funzioni riducono i dati del grafo generato ad informazioni fruibili al framework. La prima funzione restituisce una lista di stringhe che rappresenta la lista di nomi di funzioni usate dall'interno di una funzione. Questo metodo offre anche l'opzione ricorsiva che è disattivata in uso predefinito. La seconda funzione ritorna tutti gli archi del grafo che collegano due funzioni, i quali possono essere raggiunti da una funzione di partenza. Entrambi i metodi richiedono un nome di funzione da cui iniziare l'analisi e il file in cui è contenuta, e gestiscono gli errori in caso il file o il nome della funzione non sia trovato.

Purtroppo, una limitazione del software Frama-C è quella di non poter redirigere su standard output i dati da noi richiesti, poiché tale canale di comunicazione è già occupato dai messaggi informativi e di debug che il software fornisce all'utente. Quindi i dati generati da Frama-C, vengono salvati su un file in formato DOT. Questo formato utilizza solo caratteri leggibili dall'utente ed è un formato utilizzato per rappresentare grafi su testo.

Per leggere il file in linguaggio DOT, il framework utilizza il modulo Python `pydot`[2] che è in grado di aprire e leggere un file con tale estensione. Una volta letto il file, si può interrogare lo strumento per ricevere tutti i nodi e gli archi del grafo. In particolare Frama-C ritorna gli archi e i nodi in maniera ricorsiva a partire da una funzione specifica. Ciò significa che nel grafo troveremo solo le funzioni direttamente chiamate da una funzione di partenza e nessun altro nodo. Nel caso non specificassimo il nome di una funzione da cui far partire l'analisi, lo strumento l'analizzatore statico provvederebbe a cercare la funzione `main` e ritornerebbe errore nel caso la funzione non fosse presente nel codice.

4.2.4 Interfaccia a GCC

Questo è il modulo che si interfaccia con il compilatore GCC. Tale strumento è utilizzato dal framework per preprocessare il codice e per ottenere informazioni utili per lo spostamento delle funzioni. Il modulo annuncia due metodi fondamentali. Il primo riceve come unico parametro il percorso di un file e restituisce una stringa testuale contenente il codice preprocessato letto dal file. Il preprocessore è impostato in modo da restituire lo stesso codice con tutte le linee di commento rimosse. In questo modo sarà molto più semplice individuare le funzioni per poterle spostare dentro all'enclave. In caso di errore (e.g. file non esistente) il modulo lancia un'eccezione e l'esecuzione del framework termina informando l'utente del problema.

Il secondo metodo utilizza una direttiva del compilatore per calcolare un file di informazioni relative alle funzioni di libreria ridefinite da Intel SGX. Questo file contiene un insieme di dichiarazioni di funzioni con annessi i percorsi ai file che le contengono. In questo modo è possibile utilizzare questi dati per calcolare i file da

[1] https://pypi.org/project/pydot/

[2] https://pypi.org/project/pydot/

includere seguendo le dipendenze delle funzioni. Il metodo quindi riceve una lista di nomi di funzioni e ritorna, per ogni funzione, la libreria in cui è definita. In questo modo si potranno includere solo le librerie necessarie nel file dell'enclave.

4.2.5 Interfaccia a Ctags

Questo modulo viene utilizzato come interfaccia verso lo strumento Ctags. Il framework richiama il modulo per ottenere tutti i nomi definiti dall'utente nei propri file sorgente: strutture dati, unioni, enumerazioni, funzioni e definizioni. Questo modulo controlla l'univocità dei nomi e restituisce errore in caso di ridefinizioni. Questi nomi vengono utilizzati nelle analisi iniziali effettuate dal framework in cui si controllano gli argomenti delle chiamate ECall ed OCall. In questo modo il framework si assicura che i tipi degli argomenti di queste funzioni siano solo quelli definiti direttamente dall'utente. In particolare, lo strumento Ctags è in grado di leggere tutti i file sorgenti dell'utente con un unico comando e restituire il tipo desiderato dall'analisi del framework.

4.2.6 Interfaccia file EDL

Questo è il modulo che fa riferimento al file con estensione EDL, responsabile della definizione di tutte le ECall ed OCall. In questo file si definiscono anche le strutture dati dell'utente. In questo modo le funzioni interne all'enclave avranno la definizione delle strutture necessarie e allo stesso modo il codice esterno potrà importare le funzioni ECall e le strutture dati da un unico file header. Il file EDL ha una sintassi simile ai file di intestazione C (header) con alcune notazioni aggiuntive. Infatti, questo è diviso in tre parti separate in cui si definiscono le strutture, le ECall e le OCall. Il modulo che gestisce la scrittura di tale file converte i nomi e gli argomenti delle funzioni nella sintassi corretta standard di SGX.

Per fare ciò, il modulo prende in considerazione diversi dati. Per prima cosa sono necessarie le informazioni contenute nelle macro del framework dichiarate in Sezione 4.1, dove è specificato il nome della funzione, il tipo di chiamata (ECall o OCall) e il tipo di gestione degli argomenti puntatori all'entrata e all'uscita della chiamata. Successivamente sono necessarie le definizioni delle funzioni utente che saranno i punti di ingresso ed uscita dell'enclave. Poi si utilizzano le definizioni trovate nei file sorgenti dell'utente (direttive `#define`), in modo tale da poter espandere i nomi delle definizioni che per esempio specificano le dimensioni di eventuali buffer. Infine si utilizzano le definizioni dei tipi definiti dall'utente quali strutture dati, unioni ed enumerazioni. Il modulo quindi fornisce 5 metodi principali: i primi due permettono di impostare le strutture dati e le definizioni da inserire nel file. Il terzo ed il quarto permettono di aggiungere un oggetto di tipo Funzione (definito in Sezione 4.2.2) alla lista delle ECall o OCall rispettivamente. L'ultimo metodo genera la sintassi e salva il file su disco al percorso indicato all'esecuzione del costruttore. Quest'ultimo richiama iterativamente su ogni oggetto Funzione la generazione della propria sintassi EDL per definire la chiamata ECall o OCall. L'oggetto funzione calcola la sintassi EDL necessaria utilizzando i dati degli argomenti e della propria macro.

```
#define
   sgx_ecall_calcola (
       [val, i, 10],
       [val1, b, d],
       [buff, o, size],
       [ptr1, u],
       [f, i],
       [ptr2, b, size],
       [vectors, b, 5])

int calcola(
       struct myS * val,
       enum myE *
          val1[10],
       union myU val2,
       char *buff,
       int size,
       float f[10],
       double d,
       void* ptr1,
       void* ptr2,
       int *
          vectors[100]
){...}
```

(a) Prima della conversione.

```
public int calcola(
       [in,count=10] struct myS * val,
       [in,out,count=d] enum myE *
          val1[10],
       union myU val2,
       [out,count=size] char *buff,
       int size,
       [in] float f,
       double d,
       [user_check] void* ptr1,
       [in,out,count=size] void* ptr2,
       [in,out,count=5] int*
          vectors[100]
);
```

(b) Dopo la conversione.

Figura 4.4: Esempio di conversione della macro in sintassi EDL.

In Figura 4.4 è mostrato un esempio completo di conversione effettuata dal framework partendo dai dati contenuti nella macro e dagli argomenti della funzione. Come è possibile notare, nella Figura 4.4a è mostrata la definizione della funzione mentre nella Figura 4.4b è mostrata la dichiarazione generata dal framework in sintassi EDL. La direttiva `#define` posta nei pressi della definizione della funzione dichiara il nome della funzione a cui fa riferimento, il tipo di chiamata (ECall, OCall), il modo di passaggio degli argomenti puntatore (con o senza copia) e l'eventuale dimensione puntata dai puntatori. Successivamente, il framework definisce il significato del valore della dimensione a seconda del tipo puntato: in byte attraverso l'uso della direttiva `size`; in unità per contare i dati puntati, utilizzando la direttiva `count`. Infine, si nota anche che viene aggiunta la direttiva `public` necessaria per esportare esternamente le funzioni interne all'enclave come descritto in Sezione 2.2.2.

Il file EDL viene utilizzato in fase di compilazione per generare le librerie associate all'enclave che permettono la commutazione tra l'esecuzione del codice protetta ed esposta. La funzionalità ed il modo d'uso delle librerie sono definiti nella Sezione 2.2.3.

4.2.7 Gestore della conversione

Questo modulo gestisce il flusso di conversione definito in Sezione 5.3. Riassumendo, il modulo si occupa di leggere i file sorgenti C per generare un modello ad alto livello di tutte le entità definite (e.g. funzioni, strutture dati) e controllare i limiti richiesti; analizzare il modello per dividere il codice sorgente in due ambienti separati (esposto e protetto dall'enclave); calcolare le librerie necessarie alla compilazione dell'enclave; modificare i file sorgenti C con la gestione delle chiamate ECall ed OCall e generare i file sorgente e di definizione dell'enclave. Questo modulo utilizza internamente i moduli descritti precedentemente come mostrato in Figura 4.3. Inoltre, definisce delle funzioni ad alto livello utilizzate durante la generazione del modello, l'analisi dei dati e la modifica dei file sorgenti.

Inizialmente una delle funzioni che si occupa di generare il modello, crea e ritorna tre liste di funzioni che rappresentano le chiamate ECall, le chiamate OCall e le funzioni definite dall'utente. Altri metodi si occupano di restituire gli altri nomi presenti nei file sorgenti come strutture dati, definizioni, file importati. Su questi dati il modulo esegue analisi ad alto livello.

In particolare le funzioni connesse all'analisi statica permettono di eseguire le operazioni di esplorazione del grafo delle chiamate. Infatti, il modello costruito dai file sorgenti include un grafo di chiamate a funzione generato dallo strumento esterno Frama-C, come esposto in Sezione 4.2.3. Un'altra funzione di particolare importanza è il calcolo delle librerie standard C ridefinite da SGX, con particolare attenzione alle funzioni non più disponibili all'esecuzione in ambiente protetto, come spiegato in Sezione 2.1.6. Inoltre, le funzioni di analisi controllano la divisione dei due contesti: protetto ed esposto. Infatti, le funzioni interne ed esterne all'enclave non possono chiamarsi reciprocamente poiché i due ambienti sono separati per definizione. È quindi necessario analizzare il comportamento di ogni funzione per verificare questa limitazione.

Invece, per la sezione di modifica del codice sorgente, la funzione più ad alto livello definita dal modulo si occupa di riconoscere le definizioni dei tipi utente ed esportarli nell'enclave. Questa funzione gestisce anche l'esportazione delle direttive `#define` necessarie molto spesso per la definizione di costanti usate nel codice. Un'ultima funzionalità del modulo è generare il file di log; il programmatore può decidere il livello di dettaglio dei messaggi informativi generati.

4.3 Limiti del framework

Il framework di conversione del codice converte i sorgenti modificando e aggiungendo nuovi file necessari all'enclave. Tuttavia, l'analisi e la modifica del codice sorgente non consentono l'utilizzo di alcune direttive C. Inoltre, il framework impone dei

limiti al codice in ingresso, necessari per ridurre la complessità della conversione.

4.3.1 Enclave unica per programma

Un primo limite del framework è la possibilità di istanziare una sola enclave. L'architettura SGX permette l'allocazione di più enclavi per lo stesso programma in esecuzione. Questa funzionalità è utile per dividere i contesti protetti in modo da ridurre la complessità delle enclavi, facilitare la gestione dei dati e diminuire la superficie di attacco dell'enclave come spiegato nella Sezione 2.1.1. Inoltre, dividere il codice in più pacchetti indipendenti permette di riutilizzare singolarmente la parte di codice di interesse in altri progetti senza modificarla, facilitando le attività di sviluppo e di debug nel caso il pacchetto sia già stato testato. In questo modo si ottiene un codice strutturato e modulare utile nei grandi progetti per ridurre il carico di lavoro. Ad oggi questa funzionalità non è disponibile per il framework di conversione automatica del codice. La gestione di più enclavi risulta più complessa in quanto è necessario generare più file indipendenti per ogni enclave e risolvere le dipendenze software tra tutte le singole parti. Inoltre, l'utente o il framework dovrebbe dividere le funzioni per contesto, valutando l'impatto prestazionale della divisione e specificando per ogni funzione da proteggere l'enclave di destinazione. Nel primo caso, l'utente si fa carico di gestire il problema autonomamente. Nel secondo caso invece occorrerebbe implementare un algoritmo in grado di comprendere la coesione tra le funzioni e dividerle in diverse enclavi in base ai contesti individuati.

4.3.2 Supporto a funzioni standard non ridefinite da Intel SGX

Come detto in precedenza, le funzioni interne all'enclave non possono eseguire funzioni di libreria esterne, ad eccezione di quelle ridefinite nelle librerie SGX. Per esempio non è disponibile al codice contenuto in un'enclave la chiamata di sistema `printf`. Il framework controlla durante la conversione che le funzioni chiamate dall'interno dell'enclave siano solo quelle da essa definite, chiamate OCall o funzioni di libreria SGX. In caso contrario l'esecuzione della conversione viene arrestata, l'utente viene avvisato e deve procedere alla modifica del codice. In questo caso la conversione del codice non può essere automatizzata per due principali ragioni. L'uscita dall'esecuzione protetta per ogni funzione standard comprometterebbe le performance in esecuzione come approfondito in Sezione 6.3.2. Inoltre, il framework non ha modo di valutare se i dati passati alle funzioni esterne siano sensibili.

4.3.3 Supporto a variabili `static`

Le variabili globali non possono essere condivise tra l'enclave e l'ambiente non protetto. SGX impone questa limitazione, poiché per definizione, tutti i dati inseriti nell'enclave devono essere protetti e non leggibili dall'esterno. L'utilizzo delle variabili globali condivise tra i due ambienti non è quindi supportato dal framework.

Questo perché i dati condivisi sarebbero visibili sia dentro che fuori l'enclave, il che renderebbe insensata la protezione del dato in enclave. La conversione automatica di variabili di file, statiche o definite `extern` non è implementata né gestita. Un modo per gestirlo sarebbe quello di individuare le funzioni protette che utilizzano la variabile globale. Poi, bisognerebbe modificare il codice sorgente in modo da aggiungere dove necessario la variabile globale negli argomenti delle funzioni individuate. Tuttavia questo richiede comunque un intervento dell'utente per specificare l'eventuale dimensione dei nuovi puntatori globali passati alle funzioni come richiesto da Intel SGX e spiegato in Sezione 2.2.2. In generale l'utente deve essere consapevole che le funzioni interne ed esterne all'enclave non possono condividere dati comuni a meno che passati come argomenti alle funzioni.

4.3.4 Tipo intero come valore di ritorno delle ECall ed OCall

Le funzioni definite come punto di ingresso ed uscita dell'enclave devono avere come tipo di ritorno un valore intero. Questo poiché il codice di queste funzioni è incapsulato nelle funzioni wrapper che gestiscono gli errori relativi. A questo proposito il codice wrapper definisce un intero e passa il puntatore a tale valore alla vera chiamata SGX. Per questo motivo si è deciso di fissare il valore di ritorno ad un codice intero eliminando anche la possibilità di non ritornare alcun valore (tipo void). Non è da escludere una miglioria che permetta di passare tipi qualsiasi e anche puntatori nel valore di ritorno a patto che questi siano allocati e liberati dal programmatore. In ogni caso la comunità che utilizza il framework SGX preferisce il passaggio di valori, anche di ritorno, come parametri puntatori alle funzioni in modo che l'architettura si faccia carico di copiare i buffer in due copie distinte dentro e fuori la regione protetta [12]. Questa copia permette un'esecuzione più sicura poiché i puntatori sono validati, invece che direttamente utilizzati. Un ulteriore approfondimento è presente in Sezione 2.2.2.

4.3.5 Tipi dei parametri delle ECall ed OCall

Gli argomenti delle funzioni di ingresso ed uscita dall'enclave utilizzano argomenti di tipo semplice o tipi definiti dall'utente nei propri file sorgente. In questo modo si semplifica notevolmente la ricerca delle strutture necessarie all'enclave. Attualmente i tipi compatibili degli argomenti delle funzioni ECall ed OCall sono `struct`, `enum` ed `union` definite localmente nei file dell'utente. Gli altri tipi ammessi sono i più comuni come buffer se necessario (`void*`, `char*`) oppure tipi semplici come `int`, `float`, `char`, `double`. Tutte le possibilità sono definite nella Appendice A.1.6. Potrebbe essere utile aumentare i tipi ammessi specificandoli in un file di configurazione per poter importare la libreria SGX corretta che li definisce.

4.3.6 Gestione degli errori nel codice generato

Il framework, durante la generazione automatica del codice protetto, ne verifica contestualmente la coerenza ed aggiunge codice di gestione degli errori. In particolare si gestiscono gli errori relativi alla compatibilità hardware con la tecnologia SGX, gli errori di allocazione e distruzione dell'enclave e gli errori nelle chiamate ECall ed OCall. Quando possibile l'errore viene gestito con una stampa su console per informare il programmatore ed una chiamata alla funzione `exit`. Nel caso invece di errori all'interno del codice protetto, il problema viene gestito direttamente con la funzione `abort` che chiude il programma. Questo perché le chiamate alle funzioni standard non sono consentite dall'interno dell'enclave e la funzione di stampa non è quindi disponibile. In generale si potrebbe gestire l'errore con chiamate a funzioni apposite che potrebbero riprovare l'esecuzione di una chiamata a seguito di un errore, scrivere un log su un file ed eventualmente chiudere ordinatamente un programma. Per esempio, un errore che potrebbe essere molto utile gestire è il problema della sospensione della macchina in cui è aperto un programma che utilizza SGX per proteggere l'esecuzione del codice nell'enclave. Quando si sospende la macchina la memoria protetta dell'enclave viene irrimediabilmente persa e deve essere riallocata nuovamente da zero. Questo può capitare prima o durante l'esecuzione di una chiamata all'enclave con conseguenza di fallimento e ritorno di un codice di errore. Attualmente in caso di errore il programma viene arrestato e l'utente è costretto a riaprirlo. Questo approccio sarebbe migliorabile aggiungendo un'opzione al framework in modo che in caso di errore di questo genere venga riallocata l'enclave e riprovata la chiamata, eventualmente specificando anche il numero di tentativi prima di chiudere il programma. Questo potrebbe essere utile per tutte le enclavi che non richiedono un'inizializzazione con dei dati ma svolgono attività di calcolo indipendenti dai risultati precedenti. In questo caso, una possibile perdita dell'enclave dovuta alla sospensione del sistema come spiegato in Sezione 2.1.6, potrebbe essere risolta con la semplice riallocazione dell'enclave.

4.3.7 Allocazione dell'enclave

L'allocazione dell'enclave è fatta una sola volta, contestualmente alla prima chiamata ECall. Potrebbe essere interessante lasciar specificare all'utente il momento della creazione dell'enclave in modo da poter eventualmente anticipare tale operazione. In questo modo se il sistema non è compatibile con SGX, è possibile fermare il programma in un punto prestabilito. Il framework genera un codice tale da allocare l'enclave ma non chiama direttamente il codice per distruggerla quando non più necessaria. Anche in questo caso, si potrebbe aggiungere un'opzione per lasciare specificare all'utente il momento per distruggere l'enclave in modo da rilasciare le risorse quando non più necessarie. In caso contrario le risorse occupate dall'enclave saranno rilasciate solo al termine del programma.

Capitolo 5

Workflow

Come descritto nel Capitolo 1, questa tesi ha l'obiettivo di automatizzare le modifiche necessarie per rendere compatibile un'applicazione con la tecnologia Intel SGX, in modo da eseguire il codice sensibile in ambiente protetto. L'utente specifica le funzioni che andranno eseguite nell'enclave ed il framework si occupa di modificare i file sorgenti e di generare i file per la creazione dell'enclave. Infatti, nelle applicazioni SGX il software deve essere diviso in due parti ben distinte, codice da proteggere e codice esposto. Generalmente quando si progetta un'applicazione da eseguire in ambiente protetto, lo sviluppatore deve prevedere a priori una struttura del programma in modo tale da essere compatibile con l'esecuzione protetta, seguendo i limiti imposti dall'architettura e scrivendo manualmente i file di descrizione dell'enclave. Tuttavia, nei progetti già esistenti, questi adempimenti potrebbero risultare complessi. Quindi, si è sviluppato un framework, scritto in linguaggio Python, che risolve il problema automatizzando la modifica del codice sorgente e la generazione dei file di descrizione dell'enclave. Questo capitolo descrive la soluzione al problema in esame, a partire dai dati inseriti dall'utente fino alla generazione dei file per l'enclave.

5.1 Fasi del flusso

Il flusso di conversione del codice che parte dai file sorgenti dell'utente e finisce con la compilazione del codice è diviso in tre fasi. Come mostrato in Figura 5.1 le fasi sono: la scrittura delle macro del framework, la conversione automatica del codice e la compilazione. I compiti del programmatore, intenzionato a convertire il codice per l'architettura SGX, si limitano all'esecuzione della prima ed ultima fase.

Inizialmente, vengono scritte le macro del framework, le quali specificano le funzioni da proteggere, le funzioni di uscita dall'enclave ed il modo con cui si passano i parametri puntatori alle chiamate ECall ed OCall, come spiegato in Sezione 5.2. Queste decisioni non possono essere lasciate al framework poiché non è possibile comprendere automaticamente quali parti di codice debbano essere protette.

Una volta decise le parti critiche dell'applicazione che necessitano della protezione nell'enclave, il programmatore esegue il framework di conversione sul codice

sorgente annotato con le macro. Il framework mostra messaggi informativi al programmatore e, in caso si riscontrino dei problemi durante l'analisi, lo strumento informa il programmatore per gestire l'incompatibilità del suo codice con i limiti dell'architettura SGX ed i limiti del framework. La conversione del codice è affidata al framework di conversione ed è divisa in più parti, ognuna delle quali collabora con le altre per convertire il codice come descritto in Sezione 5.3.

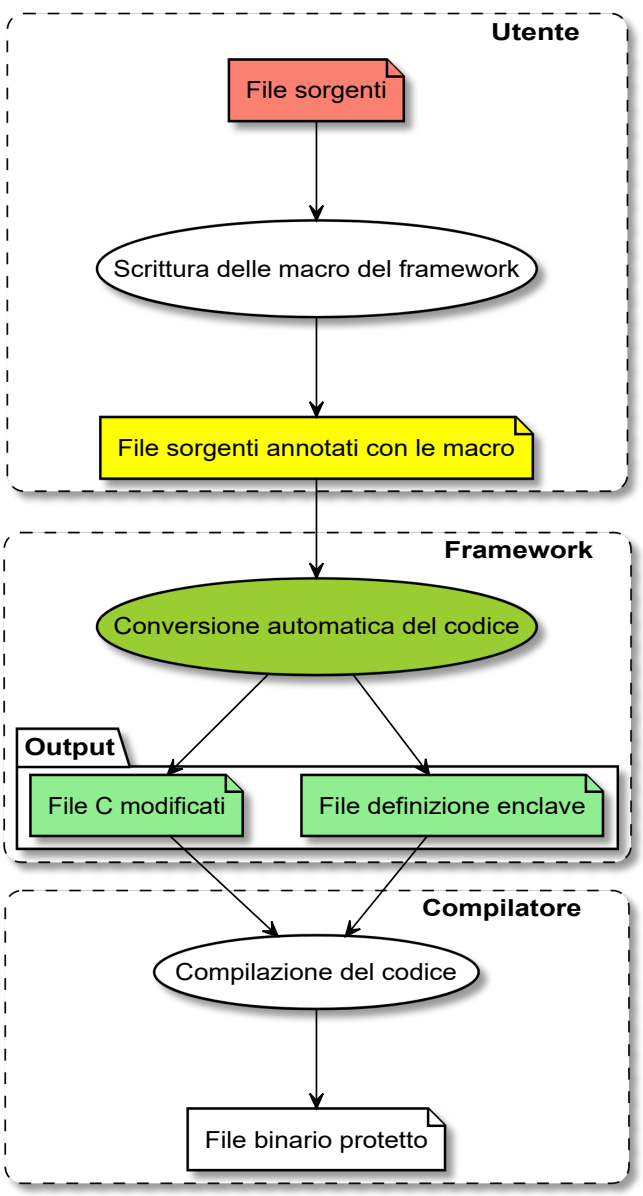


Figura 5.1: Flusso della conversione ad alto livello.

Infine, la compilazione è indipendente dal framework e si basa sul sistema operativo su cui si vuole compilare. Come spiegato in Sezione 5.4 le modalità di compilazione dipendono dal sistema operativo su cui verrà eseguita l'applicazione

ed utilizzano strumenti esterni come Microsoft Visual Studio[1] e GNU make[2].

5.2 Scrittura delle macro

Come detto introdotto in Sezione 5.1, il programmatore decide quali funzioni siano necessariamente da proteggere nell'enclave e lo comunica al framework attraverso le macro. Con le macro del framework possono essere definite sia le chiamate ECall ovvero le funzioni di ingresso all'enclave, sia le chiamate OCall ovvero le funzioni chiamate dall'enclave ed eseguite in ambiente non sicuro. Inoltre, tali annotazioni specificano il modo di trasmissione degli argomenti delle chiamate alle funzioni protette, come spiegato in Sezione 4.1.1.

In Sezione 4.1 sono disponibili sia la sintassi completa delle macro con relativi esempi che le eventuali incompatibilità.

```
#define sgx_ocall_log ([text, len, i])
int log(char* text, int len){
       return printf("%.*s\n", len, txt);
}

#define sgx_ecall_calcola ([buf, len, b])
int calcola(char *buf, int len){
       ...
       if (len > strlen(secret)){
              memcpy(buf, secret, strlen(secret));
              return strlen(secret);
       }else{
              log(problem, strlen(problem));
              return -1;
       }

}
```

Figura 5.2: Esempio di definizioni di funzioni ECall e OCall.

In Figura 5.2 si propone un piccolo esempio di definizione delle macro del framework. In particolare, è possibile trovare la definizione di due funzioni e la loro relativa annotazione ad ECall ed OCall. In questo caso il programmatore definisce una macro per la funzione "`calcola`" in modo che la sua chiamata venga eseguita nell'enclave. Inoltre, conoscendo le limitazioni di Intel SGX come spiegato in Sezione 2.1.6, il programmatore definisce una funzione OCall "`log`" per poter

[1] https://visualstudio.microsoft.com/it/vs/

[2] https://www.gnu.org/software/make/

utilizzare la funzione di libreria "`printf`" e comunicare dati all'esterno dell'enclave durante l'esecuzione in ambiente protetto. Le due definizioni espresse con la direttiva `#define` in Figura 5.2 sono le macro del framework ed hanno la notazione `sgx_[ecall/ocall]_funcName`. In aggiunta, all'interno delle parentesi tonde si utilizza una sintassi proprietaria del framework utilizzata per annotare gli argomenti delle funzioni. Come spiegato in Sezione 4.1.1, nelle parentesi viene definita una lista di argomenti di tipo puntatore e si definisce per ogni puntatore la dimensione del buffer da loro puntato. In questo modo durante la compilazione viene generato del codice aggiuntivo per copiare i buffer tra gli spazi di memoria cifrata e non cifrata, come spiegato in Sezione 2.2.2. Dopo aver aggiunto le macro per le funzioni che si desidera proteggere e per le funzioni di uscita dall'enclave, il programmatore esegue il framework per la conversione del codice.

5.3 Conversione automatica del codice

In questa sezione si descrive il funzionamento del framework analizzando le fasi della conversione. Prima di procedere con la conversione, il codice deve essere annotato come spiegato in Sezione 5.2. Il framework viene eseguito sul codice sorgente in modo tale da convertirlo e renderlo compatibile con l'architettura SGX. Il flusso di esecuzione del framework può essere diviso in quattro fasi consecutive, come mostrato in Figura 5.3.

Come prima operazione, i file sorgenti vengono letti e caricati in memoria, al fine di costruire il modello di dati che rappresenterà il codice sorgente, come spiegato in Sezione 5.3.1. Queste operazioni sono strettamente legate ai file sorgenti ed agli errori che possono verificarsi durante la lettura e la copia dei file per effettuare il backup.

Poi, viene analizzato il modello di dati appena generato per verificare la compatibilità del codice con i limiti imposti. Se tali verifiche vanno a buon fine, vengono generati i dati per eseguire la modifica del modello, come spiegato in Sezione 5.3.2. In caso contrario, il framework fornisce un messaggio informativo di errore per aiutare il programmatore a risolvere i conflitti. In questa fase si controllano le funzioni destinazione delle chiamate ECall ed OCall, si genera la divisione tra codice protetto ed esposto e si calcolano le librerie necessarie all'enclave.

Successivamente, avviene la fase di modifica dei file in memoria come descritto in Sezione 5.3.3. Questa fase utilizza i dati calcolati dalle analisi per generare il codice aggiuntivo per gestire le chiamate ECall ed OCall. Inoltre, si modificano i file sorgente dell'utente per spostare le funzioni nell'enclave e si genera il codice di definizione dell'enclave in formato EDL, come definito in Sezione 5.3.3. Come ultima operazione, si aggiunge il codice di gestione dell'enclave ai file sorgente dell'utente.

Infine, si esegue il salvataggio di tutti i file come spiegato in Sezione 5.3.4.

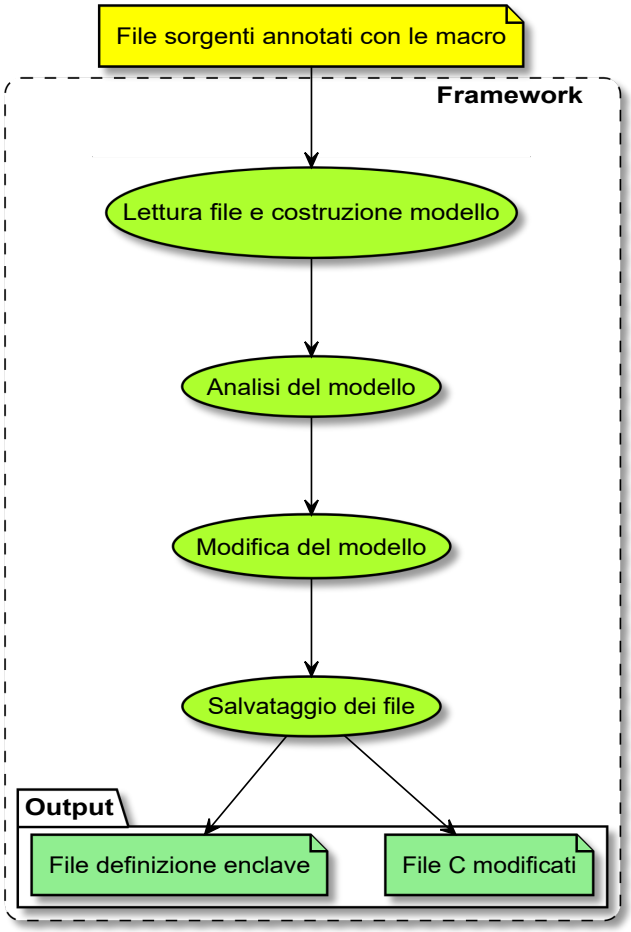


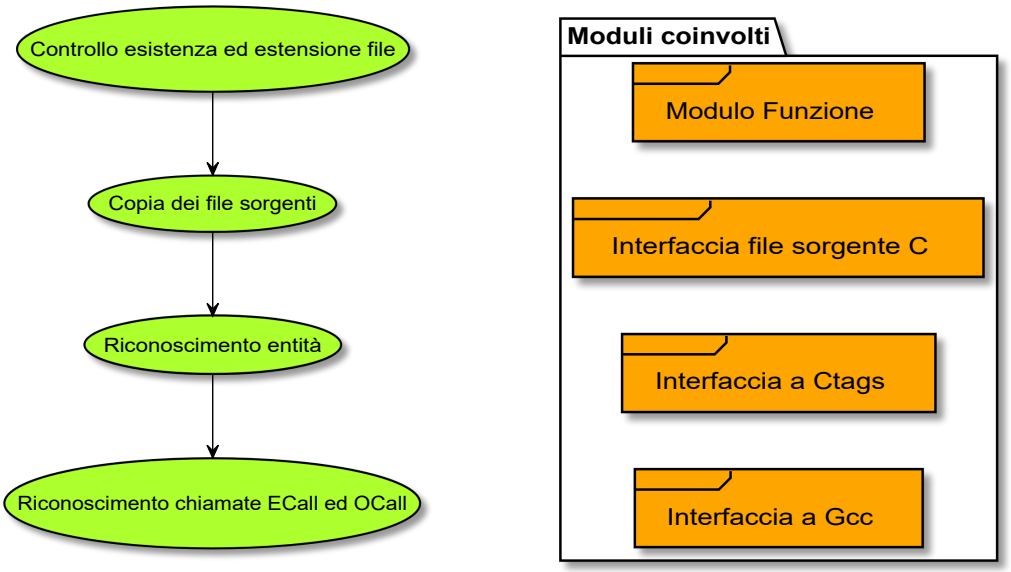
Figura 5.3: Fasi della conversione automatica del codice.

5.3.1 Lettura di tutti i file e costruzione del modello

In questa sezione vengono analizzate le operazioni effettuate per la fase di lettura dei file e la costruzione del modello. Questa fase è divisa in quattro operazioni essenziali, come mostrato in Figura 5.4a. In Figura 5.4b invece, sono segnati i moduli responsabili delle operazioni. Questa è la prima fase dell'esecuzione del framework ed inizia quindi con la lettura dei file sorgenti. Gestisce gli errori comuni dell'apertura dei file (e.g. file inesistente) e crea una copia di backup del codice. Infine, questa fase si occupa di generare un modello di dati contenente le entità del codice per poterle analizzare nella fase seguente. Con le entità del codice si intendono i tipi principali della sintassi C, come strutture dati, enumerazioni, unioni di dati, funzioni, definizioni e librerie importate.

L'inizio dell'esecuzione del framework riceve come argomenti il percorso della cartella principale contenente il codice dell'applicazione, ed i percorsi relativi dei singoli file di codice ed header C. Questi file devono avere solo estensione codice `.c` e header `.h`. In aggiunta l'utente può specificare la cartella di output dove sarà generato il codice compatibile con SGX.

Come prima cosa, viene salvata una copia di backup dell'applicazione in esame.



(a) Operazioni. (b) Moduli coinvolti.

Figura 5.4: Lettura dei file e della costruzione del modello.

Viene quindi verificata l'unicità di tutti i nomi definiti nei vari file. A questo scopo è utile utilizzare il modulo che si interfaccia con Ctags (descritto in Sezione 4.2.5) che riesce con un unico comando a leggere e costruire il modello dei nomi di tutte le definizioni. Con le definizioni si intendono tutte le dichiarazioni complete di funzioni, strutture dati, enumerazioni e unioni. Una dichiarazione di una funzione senza implementazione non è quindi restituita da Ctags, per come lo strumento è stato impostato. Questo è stato scelto poiché il framework di conversione non è interessato alla dichiarazione della funzione ma solamente alla sua definizione. Infatti, la parte di interesse è il codice implementativo della funzione che deve essere processato per procedere alla conversione dell'applicazione. Lo strumento procede con la lettura di tutti i file specificati in input, il cui contenuto è parsificato con espressioni regolari dal modulo di interfaccia con Ctags.

Una limitazione attuale del framework è l'impossibilità di rendere sicuro un codice che contiene ridefinizioni dello stesso nome o tipo anche se i file in conflitto non sono strettamente connessi. Infatti, nel codice scritto in linguaggio C è possibile definire più funzioni e tipi con lo stesso nome, a condizione che questi siano successivamente compilati in file oggetto separati. Attualmente il framework non supporta le ridefinizioni di nomi poiché questa limitazione semplifica la modifica del codice eliminando la necessità di verificare che più nomi siano in conflitto dopo aver spostato le loro definizioni nell'enclave.

A questo punto viene utilizzato il modulo che si occupa dei file sorgenti (descritto in Sezione 4.2.1) per aprire, leggere e chiudere tutti i file conosciuti dal framework. Durante la fase di lettura, si individuano le entità contenute nel codice usando varie espressioni regolari. La comprensione si limita ad un'analisi sintattica del codice basata sulle regole di sintassi del codice C. Il codice da leggere viene prima preprocessato dal compilatore (come spiegato in Sezione 4.2.4), così da eliminare

i commenti e facilitare la complessità delle regole. Il testo del file è quindi letto senza nessun commento e le espressioni regolari definite nel modulo riescono a parsificare tutti i tipi principali di nostro interesse quali strutture, enumerazioni, unioni, definizioni, inclusioni e funzioni. Tutti questi oggetti sono salvati dentro al modello del file sorgente, cioè dentro all'oggetto responsabile della sua lettura e scrittura; si costruiscono così dei dizionari Python[3] aventi come chiave il nome della definizione e come valore il codice che le rappresenta.

Questo approccio viene usato per tutti i tipi tranne che per le funzioni, per le quali è stato predisposto un tipo oggetto specifico (descritto in Sezione 4.2.2) per memorizzare il valore che viene rappresentato dai segmenti della definizione. Infatti, le funzioni C vengono divise concettualmente in quattro parti:

1. il tipo di ritorno;
2. il nome della funzione;
3. l'insieme degli argomenti;
4. il codice implementativo.

In aggiunta, gli oggetti rappresentativi delle singole funzioni memorizzano anche il file dove queste ultime sono definite (con il codice implementativo) e se calcolati esplicitamente salvano i risultati della computazione degli archi e nodi di funzioni da esse chiamate in modo da riutilizzare lo stesso risultato senza ricalcolarlo.

Inoltre, gli oggetti rappresentanti funzioni definite ECall o OCall, che come descritto nella Sezione 2.2.1, sono gli estremi di ingresso ed uscita dall'esecuzione in ambiente protetto, memorizzano un ulteriore parametro aggiuntivo proveniente dalla definizione della macro definita dall'utente. Si ricercano nel codice le definizioni delle macro in modo tale da riconoscere quali funzioni l'utente voglia rendere sicure. Anche in questo caso si utilizzano espressioni regolari per riconoscere le macro.

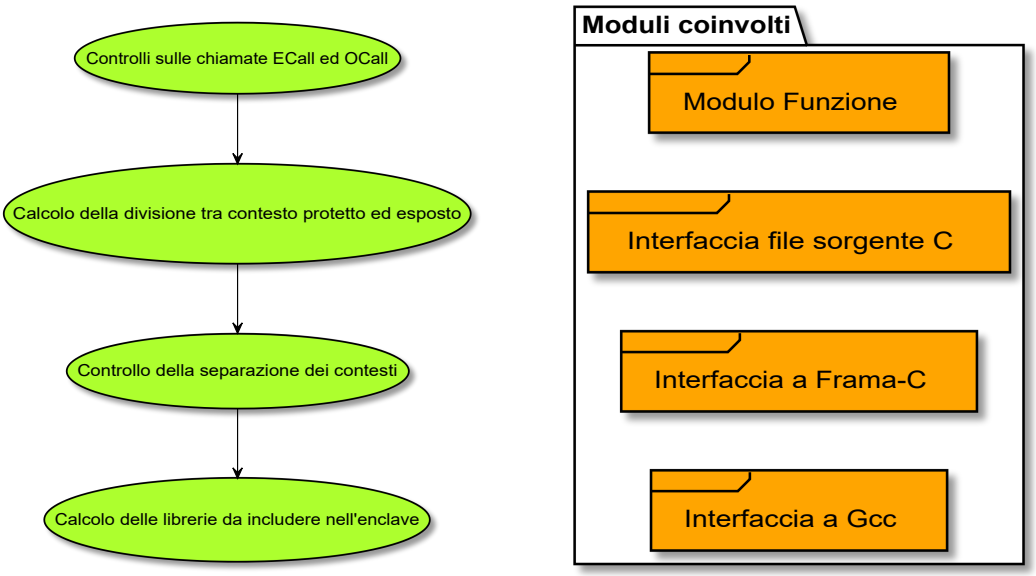
Una volta lette le macro che definiscono il tipo di funzione (ECall, OCall) e i parametri per gli argomenti, si associa ogni macro alla funzione del file corrispondente. In questo momento si controlla anche che la macro sia congruente con la sintassi definita dal framework e si verifica che gli argomenti ed il loro tipo siano compatibili con le opzioni scelte dall'utente. Inoltre, il framework si assicura che tutte le funzioni ECall ed OCall non siano definite statiche come imposto dall'architettura SGX.

I file non vengono mai riletti più volte, poiché il contenuto è completamente memorizzato negli stessi oggetti usati per leggerli. In questo modo, limitando la rilettura dei file, l'efficienza del framework e di conseguenza la velocità di esecuzione vengono aumentate.

[3] https://www.w3schools.com/python/python_dictionaries.asp

5.3.2 Analisi del modello

In questa sezione si definiscono le analisi che vengono effettuate sul modello dei dati appena generato. Questa fase di analisi segue la fase di lettura dei file e di costruzione del modello. Come mostrato in Figura 5.5b, l'analisi utilizza il modulo di interfaccia a Frama-C (descritto in Sezione 4.2.3) ed il modulo di interfaccia a GCC (descritto in Sezione 4.2.4).



(a) Operazioni. (b) Moduli coinvolti.

Figura 5.5: Analisi del modello.

Come descritto successivamente questi verranno utilizzati per generare il grafo delle chiamate e per calcolare le librerie SGX da includere nel sorgente dell'enclave.

L'analisi può essere divisa in quattro principali operazioni, mostrate in Figura 5.5a, le quali si occupano di generare le informazioni necessarie alla fase successiva durante la modifica dei sorgenti C. Inizialmente si effettuano dei controlli sulle chiamate definite ECall ed OCall, per verificare la compatibilità del codice ai limiti imposti dal framework. Poi, l'analisi si occupa di calcolare l'insieme di funzioni da esportare nell'enclave. Infatti, le funzioni da proteggere sono le ECall definite dall'utente e tutte le funzioni utilizzate dalle ECall che possono essere inserite nell'enclave. Poi, l'analisi continua per controllare la separazione delle dipendenze software tra i due contesti sicuro ed esposto. Questa operazione dipende dalla precedente dove si calcolano le funzioni interne ed esterne all'enclave. Infine, la fase si occupa di calcolare l'insieme di librerie necessarie da importare nell'enclave.

Controlli su ECall ed OCall

Per iniziare è opportuno verificare che una funzione definita come ECall non sia anche erroneamente definita come OCall con una doppia errata definizione di macro. Infatti, l'architettura Intel SGX impone che le funzioni definite ECall ed OCall

debbano avere nomi differenti, come definito in Sezione 2.2.2. Inoltre, il framework non permette di utilizzate lo stesso nome per due funzioni diverse e questo limite vale anche per le funzioni definite ECall ed OCall. Quindi si verifica che i nomi delle funzioni di entrata ed uscita dall'enclave siano univoci. Poi si impone che tutti i tipi degli argomenti di queste funzioni, siano tipi semplici (e.g. `int, char, float, double` come definiti in Appendice A.1.6) oppure definiti dall'utente stesso nei propri file esposti alla conversione. Quest'altra limitazione del framework semplifica la ricerca dei tipi a quelli definiti dall'utente, in modo che questi possano essere esportati nell'enclave.

Per esempio, non è possibile usare come argomenti delle chiamate ECall e OCall strutture, enumerazioni e unioni definite in librerie esterne ma devono essere necessariamente dichiarate all'interno del codice dell'utente. Per di più non è concesso l'uso di tipi personalizzati (i.e. typedef[4]) ma è necessaria l'intera notazione `<struct,union,enum> <nome tipo> <nome argomento>` ad esempio `struct my_struct s`.

In questo modo si è semplificato notevolmente la ricerca delle definizioni da includere all'interno dell'enclave, soprattutto per la limitazione dell'espansione dei typedef. Infatti, l'uso dei nomi personalizzati in C permette di ridefinire infinite volte ricorsivamente un nome. Questo problema di ricerca viene ulteriormente complicato se le ridefinizioni sono sparse tra più file utente. Per questo motivo il framework si limita ai nomi non ridefiniti. Non è comunque da escludersi un possibile miglioramento del framework in questo senso magari facendo uso nuovamente di direttive di preprocessore.

Generazione del codice interno ed esterno all'enclave

Questa sezione si occupa della parte più critica dell'analisi; è infatti necessario dividere il codice in due parti: la parte che rimarrà esterna all'enclave e quella che invece sarà completamente protetta.

Come prima fase si costruisce un grafo delle chiamate tra funzioni che collegano nomi di funzioni tra loro connesse. Ogni nodo rappresenta una funzione del programma e l'arco tra due funzioni indica che una è chiamata dall'altra. Si inizia sempre e solo da un'ECall e si analizzano le funzioni che ipoteticamente possono essere indirettamente chiamate da essa e si ripete questa analisi per tutte le ECall. Per questo scopo si utilizza necessariamente uno strumento di analisi che sia in grado di comprendere semanticamente il codice C e che possa quindi costruire un grafo delle chiamate a funzione a partire da una funzione di partenza. Si è valutato l'utilizzo di espressioni regolari anche per questo scopo ma, come descritto in Appendice B.3.2, l'analisi sintattica non comprende i puntatori a funzione che potrebbero essere utilizzati nel programma dell'utente. Si è proceduto con l'analisi semantica offerta da Frama-C per costruire il grafo delle chiamate a funzione. Il modello è costruito in modo incrementale per ogni funzione, progressivamente all'esplorazione di ogni nodo. Durante le iterazioni dell'esplorazione in ampiezza del

[4] https://it.wikipedia.org/wiki/Typedef

grafo, l'algoritmo calcola l'insieme minimo dei nodi che dovranno essere esportati nell'enclave e di conseguenza l'insieme delle funzioni escluse che rimarranno esterne.

Per comprendere pienamente il problema che si sta risolvendo si introduce un esempio.

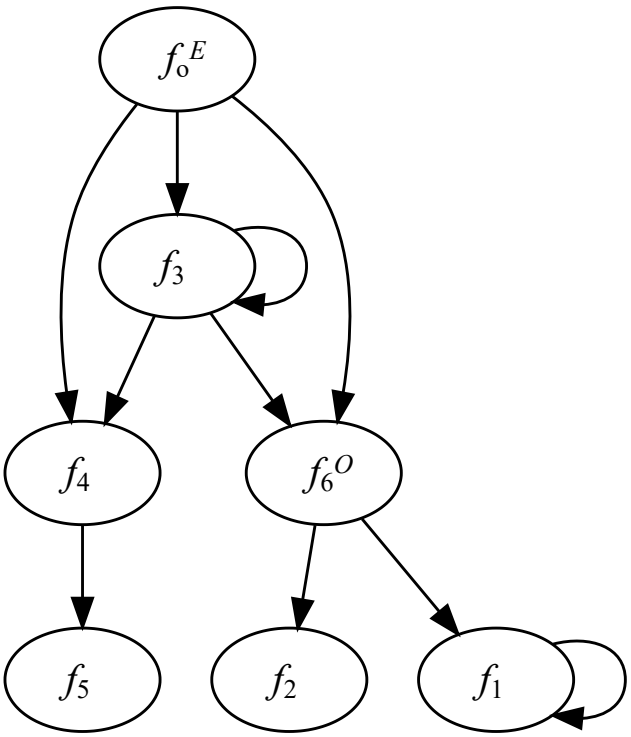


Figura 5.6: Grafo in uscita con l'esplorazione in ampiezza.

Il grafo in Figura 5.6 rappresenta un semplice modello di call-graph costruito dal framework utilizzando Frama-C. La sintassi utilizzata nell'esempio usa la dicitura f_n^E per le funzioni ECall, f_n^O per le OCall e f_n per tutte le altre funzioni generiche. Per di più la freccia $(f_3 \rightarrow f_4)$ indica che la funzione f_3 invoca la funzione f_4.

Inizialmente si parte da una funzione f_0^E (ECall) che per definizione deve essere inclusa nell'enclave. Si richiama l'analisi statica (descritta in Sezione 4.2.3), si calcolano i nodi a lei direttamente connessi e si escludono tutti quelli già visti o quelli sicuramente esterni all'enclave, ovvero le OCall. Si ricorre con lo stesso approccio su questi nodi esplorando tutto il grafo in ampiezza. In questo modo si ricorre solo sui nodi che sono utilizzati direttamente dalle ECall e che sicuramente non sono punti di uscita dall'enclave (OCall). In sintesi l'algoritmo è una ricerca in ampiezza con filtro sui nodi da esplorare.

L'algoritmo utilizza 3 differenti liste `todo, seen` e `ocalls` utilizzate rispettivamente per memorizzare le funzioni ancora da analizzare, le funzioni già analizzate e le funzioni definite OCall. L'obbiettivo dell'algoritmo è di analizzare tutte le funzioni nella lista `todo` e restituire la lista `seen` al termine. La computazione riceve la lista `todo` che è inizializzata con l'insieme di funzioni definite ECall, generata a partire dalle definizioni delle macro inserite dall'utente nei file sorgenti. Ad ogni iterazione viene rimossa dalla lista `todo` una funzione alla volta. Questa la si aggiunge

alla lista `seen` la quale rappresenta anche le funzioni che dovranno essere inserite nell'ambiente protetto. Poi, si calcola l'insieme delle funzioni chiamate dalla sua implementazione. Queste sono candidate all'inserimento nell'enclave. Dall'insieme appena calcolato si rimuovono tutte le funzioni già analizzate e tutte le funzioni definite OCall, che per definizione devono essere esterne all'enclave. L'insieme risultante viene aggiunto alla lista `todo`, ovvero l'insieme di funzioni ancora da analizzare.

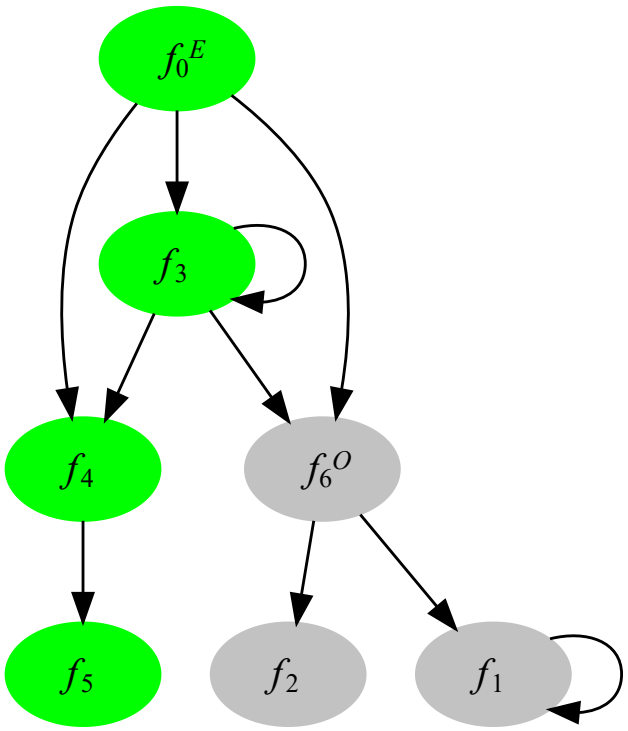


Figura 5.7: Grafo interpretato con la divisione in codice sicuro e non sicuro; sono rappresentate in verde le funzioni da proteggere mentre in grigio quelle esterne all'enclave.

Al termine della computazione, i nodi da proteggere saranno quelli rappresentati in verde nel grafo in Figura 5.7 mentre in grigio quelli esterni. La variabile lista `seen` conterrà tutte le funzioni viste e che devono essere spostate nell'enclave, ovvero tutte le funzioni verdi.

Controllo della separazione dei contesti di parte interna ed esterna

Questa fase avviene dopo la divisione effettuata nella precedente Sezione 5.3.2 e si occupa di controllare la separazione delle dipendenze software tra la parte interna ed esterna all'enclave. Infatti, dopo aver identificato le funzioni da spostare dentro e fuori dall'enclave, è necessario assicurarsi che quelle interne non usino funzioni esterne a meno che non siano OCall e che viceversa, funzioni esterne non usino funzioni interne a meno che ECall. Riferendosi all'esempio in Figura 5.7, questo significa che le funzioni f_6^O, f_2 e f_1 non possono chiamare direttamente le funzioni f_3, f_4 e f_5 mentre le funzioni f_0^E, f_3, f_4 e f_5 non possono chiamare direttamente f_1

e f_2. In breve, escludendo ECall e OCall, il codice esterno all'enclave non chiama il codice interno e viceversa.

Questo ulteriore controllo è fondamentale poiché all'interno dell'enclave non è possibile raggiungere codice definito esternamente e allo stesso modo il codice protetto non è visibile all'esterno. Le uniche funzioni visibili, sia dalla parte protetta che dalla parte esposta, sono le chiamate ECall e OCall. Questo limite è imposto dall'architettura Intel SGX e di conseguenza rende necessaria l'analisi della separazione dei contesti. Se un errore del genere viene rilevato il framework lo comunica all'utente e la richiesta non viene completata.

Per questo controllo si utilizza nuovamente l'analizzatore statico Frama-C ma per migliorare le performance, i dati delle funzioni chiamate dall'interno di un'altra funzione vengono salvati alla prima computazione e tale analisi viene eseguita solo una volta a funzione. Questo approccio riduce estremamente il tempo di esecuzione, poiché si riduce l'utilizzo di Frama-C che utilizzato in modalità semantica usa molti cicli di processore, ed inoltre si limita la rilettura dei file da disco.

Potrebbe essere possibile in casi particolari voler comunque condividere una funzione sia dentro che fuori dall'enclave. Un tipico esempio sono le funzioni di gestione dei file di log, le quali memorizzano internamente il puntatore al file, evitando di passarlo nei parametri tra le funzioni. Un altro esempio sono le funzioni responsabili di ricevere l'input da tastiera o da rete e che utilizzano i dati ricevuti per riempire le strutture dati da utilizzare sia in ambiente protetto che non protetto (e.g. conversione di dati da formato XML e JSON). A questo scopo potrebbero presentarsi due diverse opzioni:

1. definire la funzione da condividere come ECall o OCall in modo da averla visibile ad entrambi gli ambienti;
2. aggiungere una direttiva al framework in modo che la funzione sia definita e copiata in entrambe le parti di codice, duplicando semplicemente il codice della funzione.

Per il momento è possibile percorrere solo la prima modalità ma non si esclude che ulteriori miglioramenti introducano una notazione macro apposita per gestire questa ulteriore personalizzazione.

Generazione delle librerie da includere nell'enclave

Questa fase dipende dai dati generati nella Sezione 5.3.2 e si occupa di generare l'insieme di librerie da includere all'interno dell'enclave. Infatti, dopo aver calcolato e controllato la divisione dei contesti, sono state individuate le funzioni, ed è definito quali di queste debbano essere spostate all'interno dell'enclave per essere protette. Tuttavia, vanno ancora decise le librerie da includere nella regione dell'enclave. Siccome tutte le funzioni dell'enclave andranno definite e spostate in un unico nuovo file sorgente C, è necessario comprendere quali librerie standard esterne debbano essere incluse all'interno dell'enclave. Questo è indispensabile alla compilazione del codice perché come nel linguaggio standard C, non possono essere invocate

funzioni definite in altri file C senza aver prima incluso correttamente i rispettivi file di header. Per esempio: se la funzione ECall utilizza le funzioni `memcpy` e/o `strlen`, sarà necessario includere la libreria `string.h` responsabile di definire tali funzioni, affinché il compilatore riesca correttamente a creare l'eseguibile.

Un approccio superficiale potrebbe essere quello di includere tutte le direttive d'inclusione che troviamo in tutti i file definiti dall'utente ma in presenza di file di header non necessari alle funzioni da spostare nell'enclave, tali header sarebbero inutilmente inclusi nell'enclave, portando quindi a scarse ottimizzazioni ed errori in fase di compilazione. Il più evidente è la duplicazione di tutte le librerie incluse, sia quelle utente sia quelle standard C ridefinite da Intel SGX. Questo porterebbe ad un inutile spreco di memoria dopo la compilazione dell'eseguibile, poiché tutte le librerie sarebbero duplicate per renderle visibili ad entrambi gli ambienti. In secondo luogo, includere anche il codice sorgente non protetto renderebbe inutile la divisione degli ambienti calcolata in precedenza nella Sezione 5.3.2. Questa soluzione eccederebbe nel numero di file inclusi e molte di queste librerie non sarebbero necessarie al codice inserito nell'enclave. Per di più, è obbligatorio limitare le funzioni utilizzate all'interno dell'enclave alle sole funzioni definite al suo interno e a quelle disponibili con le librerie di SGX. Intel infatti, ridefinisce le librerie standard e alcune funzioni base non sono disponibili all'utilizzo dentro l'enclave come spiegato in Sezione 2.1.6. Un esempio notevole è la primitiva `printf` che non è consentita all'utilizzo dentro l'enclave. Un'altra funzione degna di nota, non utilizzabile all'interno di enclavi SGX, è la chiamata `exit` che può essere facilmente sostituita con `abort` per terminare immediatamente l'esecuzione. Per questo motivo si è nuovamente utilizzato il compilatore GCC che fornisce una modalità di esecuzione, in cui prova a compilare il codice fornito dall'utente e restituisce in output un file informativo contenente una lista di nomi di definizioni e le relative librerie dove esse sono definite. Parsificando tale file con l'utilizzo di espressioni regolari sarà poi possibile identificare la libreria standard che dichiara la funzione di interesse. Il compilatore viene quindi impostato per compilare il file con le sole librerie SGX in modo da verificare che le funzioni richieste siano effettivamente presenti nelle librerie e che il prototipo sia congruente con l'utilizzo, come descritto in Sezione 4.2.4.

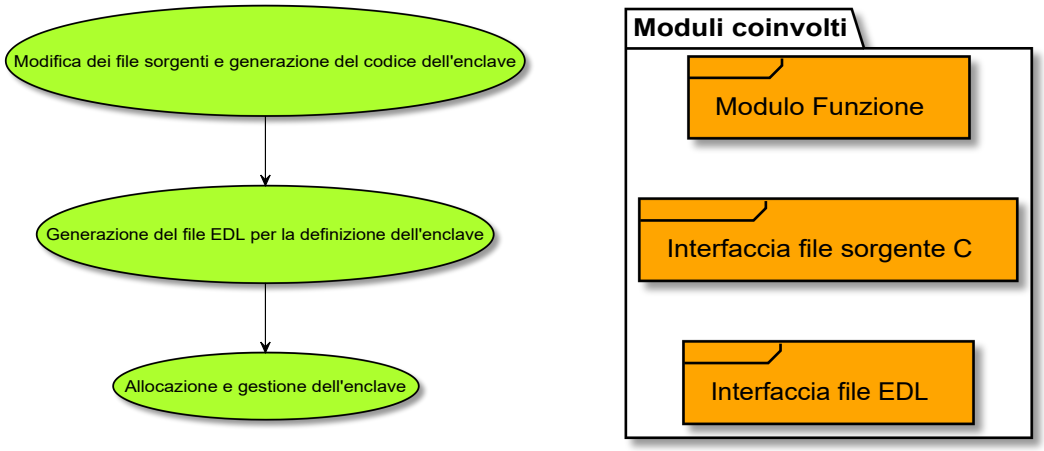
Anche in questo caso, la compilazione viene eseguita una sola volta e viene creato un dizionario di nome funzione e libreria dichiarante. Viene successivamente interrogato tale dizionario ricercando le funzioni standard utilizzate dall'interno dell'enclave, per individuare le librerie da includere.

5.3.3 Modifica del modello e creazione dei nuovi file

In questa sezione si analizzano le fasi eseguite durante la modifica del modello dei dati e la creazione di nuovi file. Per effettuare le modifiche sono necessari i dati generati nella Sezione 5.3.2. In particolare, si utilizza l'insieme delle funzioni da proteggere e l'insieme di librerie da includere. Inoltre, per generare il file EDL sono necessarie le annotazioni memorizzate nel modello dalla fase di lettura spiegata in Sezione 5.3.1. Quindi, come mostrato in Figura 5.8b vengono utilizzati tre moduli. L'interfaccia al file EDL (descritto in Sezione 4.2.6) viene utilizzata per la creazione del nuovo file di descrizione dell'enclave. Il modulo d'interfaccia ai file sorgenti C

(descritto in Sezione 4.2.1) si occupa di eseguire le modifiche del codice. In ultimo viene utilizzato il modulo Funzione (descritto in Sezione 4.2.2) per la generazione del codice di gestione delle chiamate ECall ed OCall.

Come mostrato in Figura 5.8a, i passi eseguiti si occupano di modificare i file sorgenti utente per spostare il codice da proteggere nell'enclave e generare il codice di gestione delle chiamate ECall ed OCall. Successivamente, si genera il codice del file EDL necessario per la definizione dell'enclave. Infine, si modifica ulteriormente il codice sorgente per aggiungere la gestione dell'enclave. Queste tre fasi sono eseguite in successione ma non dipendono tra loro. I dati utilizzati dalle tre fasi provengono dalle fasi precedenti di analisi o sono presenti nel modello dati costruito inizialmente.



(a) Operazioni. (b) Moduli coinvolti.

Figura 5.8: Modifica del modello.

Modifica dei file sorgente dell'utente e creazione del file sorgente dell'enclave

In questa sezione si analizzano i passi eseguiti per modificare i file sorgenti e generare il file sorgente dell'enclave. I tre passi eseguiti sono indipendenti tra loro e possono così essere riassunti:

1. spostamento del codice sicuro: viene spostato tutto il codice da esporre e proteggere all'interno dell'enclave tranne le funzioni definite chiamate ECall;

2. divisione di ECall in wrapper e trusted: viene generato il codice di gestione delle chiamate ECall per sostituire i codici originali nel sorgente esposto; vengono modificati i codici originali delle chiamate ECall per cambiare il nome alle funzioni;

3. generazione del codice wrapper alle OCall: viene generato il codice per esporre e gestire le chiamate OCall dall'interno dell'enclave.

Spostamento del codice sicuro Per eseguire lo spostamento è necessario che lo script proceda con la creazione dei nuovi file. Innanzitutto, viene creato un nuovo oggetto che rappresenta il file del codice sorgente dell'enclave. Una volta creato l'oggetto, vengono spostate al suo interno tutte le funzioni che dovranno essere conosciute all'enclave tranne quelle definite ECall che dovranno essere prima modificate. Poi, vengono incluse tutte le librerie calcolate precedentemente, necessarie a far compilare tale file. Infine, si aggiungono anche tutte le definizioni dell'utente trovate nei file sorgenti.

In questo momento, le funzioni che vengono spostate da un file all'altro non sono funzioni ECall bensì sono solo quelle funzioni definite dall'utente utilizzate internamente dalle ECall. Citando l'esempio in Figura 5.7, sono tutte le funzioni verdi escluse le ECall.

Divisione di ECall in wrapper e trusted Nella Sezione 5.3.3, non vengono spostate direttamente le funzioni ECall dentro all'enclave poiché tale operazione renderebbe l'applicazione non compilabile con SGX. Infatti, le chiamate ECall necessitano di ulteriori argomenti da aggiungere a quelli inizialmente definiti dall'utente, come spiegato nella Sezione 2.2.3.

Riassumendo, le funzioni definite ECall non possono essere chiamate direttamente dal codice dell'utente che si trova nella parte non protetta. Questo perché il compilatore non rende direttamente visibili tali funzioni dall'esterno dell'enclave. Queste chiamate possono essere invocate solo attraverso funzioni interne, generate automaticamente dal compilatore SGX a partire dal file EDL. Queste funzioni si fanno carico di eseguire le istruzioni del processore che invocano l'entrata e l'uscita dall'enclave. Per fare ciò, queste funzioni interne definiscono dei parametri aggiuntivi ed hanno un diverso valore di ritorno rispetto alla funzione originale. Il primo argomento della chiamata deve essere necessariamente l'identificatore dell'enclave in cui la funzione originale è definita. Questo identificatore è restituito alla creazione dell'enclave come descritto successivamente in Sezione 5.3.3. Il secondo argomento è un puntatore a memoria dove sarà memorizzato l'originale valore di ritorno della chiamata alla funzione ECall.

Infatti, la tecnologia SGX impone che il valore di ritorno degli involucri alle funzioni ECall, sia una struttura dati di tipo `sgx_status_t` che viene usato per verificare il risultato dell'esecuzione delle istruzioni in ambiente protetto. Come descritto nella Sezione 2.2.4 è possibile che la chiamata ad una funzione ECall o OCall fallisca per diverse ragioni, come la perdita dell'enclave. Per questo motivo il valore originale di ritorno della funzione di partenza deve essere incapsulato in un puntatore passato come secondo argomento. Nel caso la funzione ECall o OCall non restituisca valore, non ci sarà nessun parametro puntatore aggiuntivo negli argomenti come descritto nella Sezione 2.2.3.

Il framework, quindi, deve generare delle nuove funzioni wrapper in modo da chiamare le funzioni interne generate dal compilatore come spiegato nella Sezione 2.2.3, le quali aggiungono argomenti, cambiano il valore di ritorno ed utilizzano le istruzioni del processore per invocare le vere funzioni ECall. In aggiunta, tali funzioni wrapper, generate dal framework, avranno il compito di verificare il valore di `sgx_status_t` e di intraprendere una gestione degli errori di conseguenza. Per

fare ciò si procede a creare due funzioni distinte denominate `ecall_trusted` ed `ecall_wrapper`. Il wrapper sostituirà la funzione da proteggere nel file dove inizialmente era definita, mentre la parte trusted, quella da usare dentro all'enclave, sarà l'implementazione della funzione ECall iniziale opportunamente modificata.

Partendo dalla funzione da proteggere, le operazioni da svolgere sono:

1. generare due funzioni; una wrapper di gestione ed una trusted che esegue il codice originale;
2. sostituire il codice implementativo originale della funzione con il codice wrapper nel file esterno all'enclave;
3. aggiungere la funzione trusted dentro al file dell'enclave.

Per approfondire queste operazioni si introduce un esempio.

```
#define sgx_ecall_foo ([buf, u])
int foo(char *buf, int len){
...
return secret;
}
```

Figura 5.9: Esempio di definizione macro per la funzione ECall.

In Figura 5.9 è rappresentata una semplice definizione di funzione con annessa la macro per richiedere al framework di spostarla all'interno dell'enclave e renderla una ECall. A partire da questa funzione si producono le funzioni `foo` (wrapper) e `foo_trusted`, come in Figura 5.10.

Come è possibile notare in Figura 5.10, la funzione `foo` (wrapper), non cambia nome poiché deve essere utilizzata dal codice untrusted già presente e scritto dall'utente. In aggiunta tale codice mantiene inalterato il numero, il tipo e l'ordine degli argomenti. Cambia solo l'implementazione di questa funzione che viene completamente sostituita con una chiamata alla funzione nell'enclave, che conterrà l'implementazione originaria della funzione. Per fare ciò la funzione wrapper invoca la funzione ECall `foo_trusted` con i nuovi parametri aggiuntivi necessari all'entrata nell'enclave. La definizione congruente con tale nuova chiamata si trova nelle librerie generate dal file EDL descritte in Sezione 5.3.3.

È da sottolineare che la funzione trusted dell'enclave cambia nome rispetto all'originale ed è quindi necessario aggiornare tutte le funzioni nel file dell'enclave che chiamano tale funzione. In caso contrario le funzioni chiamerebbero il wrapper esterno dall'enclave, il che non è concesso da SGX e risulterebbe quindi un errore di compilazione. La ridefinizione del nome viene fatta con una `#define`, come mostrato in Figura 5.11.

```
int foo(char *buf, int len){ //wrapper
        int retval = 0;
        int ret = foo_trusted(enclave_id, &retval, buf, len);
        if (ret != SGX_SUCCESS){
                printf("Enclave error");
                exit(-1);
        }
        return retval;
}
```

(a) Codice wrapper di gestione nel sorgente utente.

```
int foo_trusted(char *buf, int len) {
...
return secret;
}
```

(b) Codice originale spostato nell'enclave.

Figura 5.10: Codice generato per ECall wrapper ed ECall trusted.

Il wrapper controlla e gestisce in modo semplice il valore di ritorno in caso di non successo. La libreria `sgx_error.h`[5] definisce settanta codici di errore diversi. Non potendo assumere a priori il comportamento desiderato dall'utente in caso di errore nella chiamata, si gestisce in modo essenziale e rigido il problema con l'uscita completa dal programma. Una futura estensione del framework potrebbe prevedere un'opzione che lasci inserire all'utente il codice desiderato per gestire eventuali errori all'esecuzione della ECall. Tuttavia, una volta terminata l'esecuzione del framework, l'utente può visionare i file generati ed eventualmente modificare il codice di gestione di tali errori.

```
#define foo foo_trusted
```

Figura 5.11: Ridefinizione di nome in C.

In Figura 5.12 si ricapitolano tutte le funzioni eseguite dal codice sorgente non protetto fino alla chiamata ECall. Il codice esposto invoca la funzione con il nome scelto inizialmente dall'utente (`foo`). Questa funzione è stata modificata dal framework sostituendo la sua implementazione con una funzione wrapper di gestione

[5] https://github.com/intel/linux-sgx/blob/master/common/inc/sgx_error.h

degli errori che chiama la funzione di libreria dell'enclave (`foo_trusted` centrale). Questa funzione di libreria è generata automaticamente dal compilatore ed esegue sia le istruzioni hardware del processore per commutare l'esecuzione in ambiente protetto che la funzione protetta. La funzione protetta (`foo_trusted` a destra) contiene l'implementazione iniziale dell'utente. Infatti, prima di tutte le conversioni, la funzione da proteggere si trovava nel codice sorgente con l'implementazione originale. Dopo la conversione il codice da proteggere si trova nell'enclave e le due funzioni di sinistra aggiunte `foo` e `foo_trusted` servono rispettivamente per gestire gli errori della chiamata ECall e per eseguire la chiamata ECall.

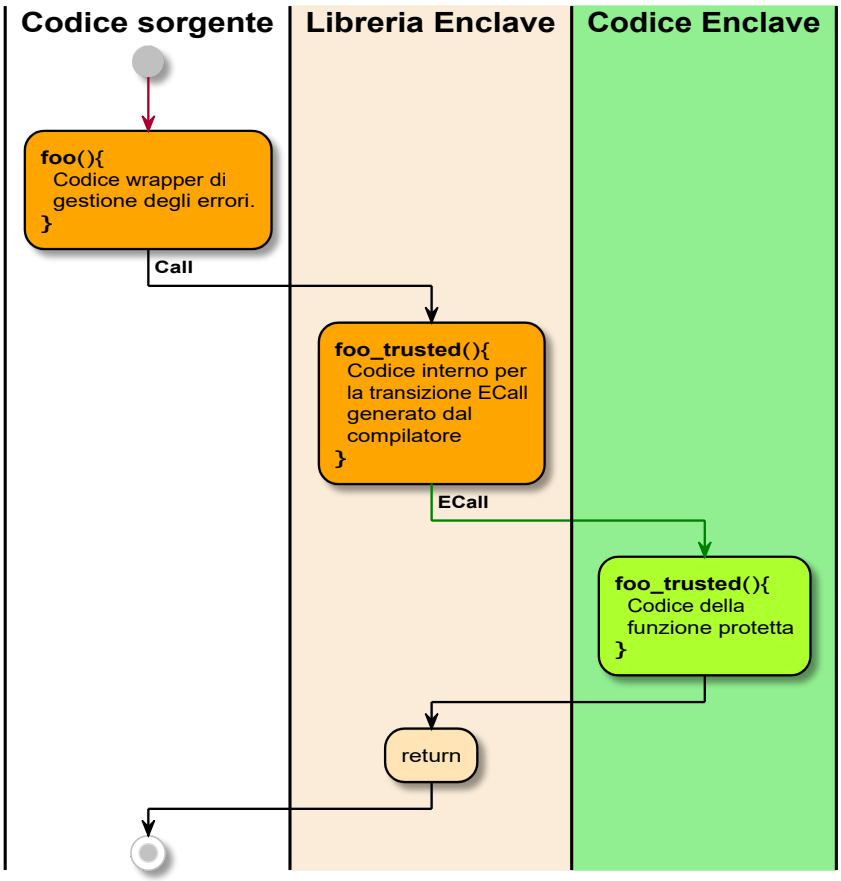


Figura 5.12: Schema delle funzioni attraversate per gestire ed eseguire una chiamata ECall.

I prototipi delle funzioni ECall congruenti con i parametri aggiuntivi ed il codice di stato nel valore di ritorno, si trovano nelle librerie generate dal file EDL descritte in Sezione 5.3.3.

Generazione del codice wrapper alle OCall Come per le funzioni ECall, le funzioni OCall non sono direttamente visibili dall'interno dell'enclave, ma si utilizzano degli involucri alle funzioni OCall generati dal compilatore a partire dal file EDL. Questi involucri, responsabili dell'esecuzione delle istruzioni hardware per uscire dall'esecuzione protetta, ridefiniscono il valore di ritorno della vera chiamata

OCall con un codice di stato che indica se la funzione esterna è stata correttamente eseguita. Quindi, le funzioni intermediarie deputate all'uscita dall'enclave hanno bisogno di un ulteriore argomento nella loro chiamata, che verrà utilizzato come puntatore a memoria per memorizzare il valore di ritorno della funzione OCall originale e renderlo disponibile al chiamante. Per questo motivo il framework genera una funzione wrapper per ogni OCall definita dall'utente. Questo ulteriore codice di gestione si limita solo a passare nell'ordine corretto gli argomenti alla funzione, di gestire il codice di ritorno e di ritornare il vero valore di ritorno della funzione OCall esterna.

```
int ocallPrintf_wrapper(char* txt, int len) {
    int retval;
    sgx_status_t ret = ocallPrintf (&retval, text, len);
    if(ret != SGX_SUCCESS){
        abort();
    }
    return retval;
}
#define ocallPrintf ocallPrintf_wrapper

int secure_ecall() {
  ...
  ocallPrintf(secret, size); //calls ocallPrintf_wrapper
}
```

(a) Codice wrapper per la chiamata OCall inserito nell'enclave.

```
int ocallPrintf (char* txt, int len) {
  ...
  return printf("%.*s\n", len, txt);
}
```

(b) Codice originale della funzione OCall chiamata dall'enclave.

Figura 5.13: Codice generato per gestire una chiamata OCall.

In Figura 5.13a è riportato un esempio della creazione della funzione wrapper di una chiamata OCall. In questo esempio è mostrato come poter utilizzare la funzione `printf` e chiamarla dall'interno dell'enclave. È però necessario uscire dall'enclave poiché tale funzione non è presente nelle librerie dell'architettura SGX. In Figura 5.13a è riportato all'inizio il codice wrapper generato dal framework in modo da passare i parametri corretti e controllare il valore di ritorno della chiamata interna alla funzione OCall. Come si nota, la funzione wrapper ha un nome diverso rispetto la seconda funzione `ocallPrintf`, che è quella intatta definita dall'utente e che non viene alterata dal framework. Per questo proposito le funzioni OCall wrapper

sono inserite all'inizio del file e vengono seguite da una direttiva al preprocessore `#define` che ne ridefinisce il nome originale. Le funzioni sicure seguiranno a loro volta la direttiva `#define` che aggiornerà le chiamate alla funzione con il nuovo nome di quest'ultima.

I prototipi delle funzioni OCall, congruente con il parametro aggiuntivo ed il codice di stato nel valore di ritorno, si trovano nelle librerie generate dal file EDL descritte in Sezione 5.3.3.

Generazione del file EDL

In questa sezione viene spiegato ad alto livello come viene generato il file EDL. I dati necessari alla generazione del file sono contenuti sia nelle macro del framework sia nelle definizioni delle funzioni ECall modificate in Sezione 5.3.3. Come prima operazione il framework crea un oggetto di interfaccia al file EDL (descritto in Sezione 4.2.6) ed inserisce al suo interno tutte le definizioni con direttiva `#define` che vengono utilizzate spesso nel codice per definire numeri come la dimensione di buffer. Poi vengono inseriti al suo interno tutte le definizioni delle strutture dati dell'utente insieme alle enumerazioni ed alle unioni di dati come spiegato in Sezione 5.3.3. Infine si genera la definizione in formato EDL di ogni funzione come descritto in Sezione 5.3.3.

Definizione delle strutture dati Il codice sorgente iniziale dell'utente viene privato di tutte le definizioni: strutture dati, enumerazioni e unioni di dati. Poiché queste definizioni sono utilizzate anche all'interno dell'enclave, è opportuno rendere visibili tali tipi anche al codice contenuto nel file dell'enclave. A tale scopo è possibile sfruttare il file EDL per contenere tutte le definizioni dell'utente. Questo viene fatto poiché il file EDL può essere utilizzato come generatore di librerie da importare nel codice per acquisire le definizioni utente come spiegato in Sezione 2.2.3. Successivamente il codice privato delle definizioni, importa dai file generati dal file EDL la conoscenza inizialmente persa. Questi file generati comprendono anche la definizione delle ECall e OCall e quindi sono di fatto necessari a tutti i file che chiamano le suddette funzioni.

Generazione delle dichiarazioni delle funzioni ECall e OCall Il framework utilizza la classe di gestione del file EDL, per generare ed inserire tutte le definizioni all'interno del contenitore. Oltre a tali dati, le dichiarazioni delle funzioni ECall e OCall vengono inserite nel file EDL. Per generare le dichiarazioni secondo la sintassi EDL, si utilizzano le macro definite dall'utente. L'utente dichiara quali parametri delle funzioni di confine devono essere utilizzati come buffer di scambio dati tra l'enclave e l'ambiente esterno. Intel SGX permette diverse opzioni per definire se la copia dei buffer deve essere eseguita sia in chiamata che al ritorno come spiegato nella Sezione 2.2.2. Con questa copia si intende che i puntatori passati negli argomenti alle chiamate ECall ed OCall, vengono utilizzati come riferimenti a buffer di dati. In fase di compilazione vengono aggiunte delle istruzioni eseguite prima e dopo l'esecuzione della chiamata. Con queste istruzioni viene fatta una

copia del buffer in modo da distinguere il puntatore in memoria protetta ed il puntatore esterno. In questo modo, un utente malevolo che riesca a compromettere l'esecuzione, non può interagire con l'interno dell'enclave passando puntatori a memoria protetta alle funzioni ECall. Tale operazione sarebbe pericolosa poiché senza effettuare una copia del buffer, il puntatore potrebbe puntare ad un indirizzo di memoria di codice o dati dell'enclave, compromettendo l'esecuzione protetta. Si intende che la chiamata ECall potrebbe scrivere su tale puntatore e modificare il proprio codice o dati d'esecuzione della memoria protetta.

Tutti i parametri non definiti da copiare verranno passati come valore alla funzione. Esiste l'opzione `user_check` del framework SGX che permette di passare puntatori alle chiamate ECall e OCall, e fare gestire tali indirizzi manualmente al programmatore. Tale operazione è da limitare il più possibile poiché la funzione invocata nell'enclave non ha modo di verificare la validità del puntatore passato come argomento.

Gestione dell'enclave

In questa sezione si descrivono le operazioni effettuate e la libreria aggiunta dal framework per la gestione dell'enclave. Il framework deve assumersi la responsabilità di generare il codice per allocare l'enclave in memoria protetta e distruggerla alla fine del processo. Per tale operazione si adopera una libreria esterna `enclave_manager.h` che è definita dal framework e viene inclusa dai file sorgenti dove sono presenti le definizioni delle chiamate ECall.

Tale libreria si fa carico di diverse operazioni:

1. controllare la compatibilità dell'hardware: nella libreria di gestione dell'enclave è possibile abilitare attraverso una direttiva `#if` il controllo della compatibilità hardware del sistema con Intel SGX, prima di eseguire l'allocazione dell'enclave;

2. allocare l'enclave: caricare l'enclave in memoria protetta attraverso le istruzioni di creazione dell'enclave di SGX;

3. gestire l'errore di creazione: verificare il codice ritornato dalla funzione di creazione dell'enclave e fornire un messaggio di errore per poter aiutare l'utente;

4. distruggere l'enclave: eseguire le istruzioni per la distruzione dell'enclave in memoria protetta per rilasciare le risorse occupate.

La gestione degli errori risulta molto semplice ma allo stesso tempo efficace. In caso di errore il programma fornisce una descrizione del problema per aiutare il programmatore a gestire il guasto. Poi, si forza la terminazione del programma poiché proseguire con l'esecuzione non sarebbe sicuro, porterebbe a risultati impredicibili e comunque l'esecuzione non sarebbe protetta dall'architettura SGX.

Nel file vengono quindi definite tre funzioni:

1. `create_enclave`: istanzia l'enclave con le funzioni di libreria SGX, gestisce il codice di ritorno e salva l'identificatore dell'enclave da usare per le chiamate ECall;

2. `destroy_enclave`: rilascia le risorse utilizzate dall'enclave;

3. `query_and_enable_sgx_device`: controlla la compatibilità hardware del sistema e fornisce un messaggio di errore per fornire aiuto al programmatore.

Allocazione e distruzione dell'enclave Per poter utilizzare l'enclave è opportuno instanziarla in memoria con la funzione `sgx_create_enclave` che restituisce tramite puntatore un identificativo `sgx_enclave_id_t`. Tale numero è l'identificatore univoco dell'enclave ed è memorizzato dalla libreria responsabile della sua gestione. Ogni chiamata ECall fa uso di questo identificativo per riferirsi all'enclave dove la funzione è definita.

La funzione che si occupa di generare l'enclave, si occupa anche di richiamare il controllo sulla compatibilità hardware del sistema in uso. Poi, l'enclave viene allocata in memoria con le funzioni di gestione di SGX. Questa rimane in memoria per tutta la durata del programma e viene rilasciata soltanto al termine dell'esecuzione del programma. La funzione che alloca l'enclave genera solo una singola enclave e si limita a ritornare se questa è stata già allocata.

All'inizio del codice della libreria si può specificare se eseguire il controllo dell'hardware quando si istanzia l'enclave. Infatti, questo controllo richiede l'esecuzione privilegiata del codice che potrebbe essere indesiderata dall'utente. Si è predisposto quindi un modo semplice per disattivare l'esecuzione del controllo, nel caso non si abbiano questi privilegi. Il controllo è consigliato poiché il fallimento della creazione dell'enclave termina il programma con un errore generico. Invece, attraverso l'uso della funzione privilegiata è possibile ottenere informazioni dettagliate sull'errore aiutando il programmatore a gestire il problema.

L'enclave può essere distrutta invocando la funzione `destroy_enclave` che riceve come argomento l'identificatore di un'enclave. Questa funzione richiama internamente `sgx_destroy_enclave` e verifica il suo codice di ritorno. In entrambi i casi, le funzioni che creano e distruggono l'enclave gestiscono l'errore facendo ritornare forzatamente il programma.

Gestione della creazione e distruzione dell'enclave Nell'esecuzione del codice C, prima di poter effettuare le chiamate all'enclave è necessario allocare tale struttura protetta in memoria attraverso la funzione prima definita. A tal proposito sono possibili due diverse strategie per collocare nel tempo la generazione della struttura. Si può generare l'enclave prima dell'esecuzione della funzione `main` o alla prima chiamata protetta.

Nel primo caso, la creazione dell'enclave avviene prima della chiamata alla funzione `main` e di conseguenza l'enclave sarà già disponibile in qualsiasi momento successivo a tale chiamata. Allo stesso tempo però si impedisce l'esecuzione totale del programma in caso l'enclave non possa essere creata. Con lo stesso concetto è

possibile invocare la distruzione dell'enclave in maniera diretta dopo il ritorno del `main` eseguendo una chiusura ordinata del programma.

Al fine di eseguire entrambe le operazioni prima e dopo il `main` si possono utilizzare le direttive `attribute`[6] dei compilatori C11[7] `constructor` e `destructor` che operano in modo simile alle funzioni costruttore/distruttore del linguaggio C++.

Le dichiarazioni delle funzioni per allocare e distruggere l'enclave saranno definite, come mostrato in Figura 5.14.

```
/* executed before main() */
void create_enclave (void) __attribute__ ((constructor));

/* executed after main() */
void destroy_enclave (void) __attribute__ ((destructor));
```

Figura 5.14: Codice per allocare e distruggere l'enclave prima e dopo la funzione `main` con l'uso di `attribute`.

Si è optato per queste direttive in quanto garantiscono portabilità del codice, a differenza di dichiarazioni specifiche di compilatori (e.g. `#pragma startup` di Borland Turbo C++[8])

Se invece si decide di far generare l'enclave alla prima chiamata protetta, essa non sarà immediatamente disponibile alla prima chiamata ECall. Infatti, la prima invocazione in ordine cronologico di una chiamata protetta richiede l'allocazione dell'enclave per poter essere eseguita. Per questa ragione la funzione di creazione dell'enclave viene chiamata all'inizio di ogni wrapper responsabile dell'invocazione dell'ECall, come mostrato in Figura 5.10a.

La funzione che si occupa di creare l'enclave (definita in `enclave_manager.h`) ritorna immediatamente e non viene eseguito alcun codice se l'enclave è stato precedentemente allocato. Questo comportamento è ottenuto mantenendo e controllando il codice identificativo dell'enclave salvato nella libreria. Il codice viene inizializzato alla prima creazione e la funzione permette l'allocazione di una sola enclave per programma. Questa soluzione è compatibile sempre con tutti i compilatori perché non utilizza funzionalità specifiche dei compilatori quali macro o attributi di compilazione. Inoltre, questa opzione consente l'esecuzione di codice non protetto senza che l'enclave sia generata in memoria.

Concretamente, è possibile che alcuni programmi non necessitino ad ogni esecuzione di generare l'enclave ed eseguire il codice protetto, poiché le funzioni utilizzate

[6] https://gcc.gnu.org/onlinedocs/gcc/Attribute-Syntax.html

[7] https://gcc.gnu.org/onlinedocs/gcc/Standards.html

[8] https://www.hackerearth.com/practice/notes/c-program-callling-a-function-before-main/

in quel momento dall'utente non lo richiedono. Tuttavia, questo metodo non fornisce soluzione per la distruzione dell'enclave. Tale mancanza è in contrasto con la corretta gestione di allocazione e rilascio delle risorse richieste esplicitamente dal programma. Infatti, è buona norma rilasciare tutte le risorse richieste esplicitamente dal programma, per esempio: l'uso della funzione `fclose` per chiudere i file aperti dal programma. Questo però non è un problema grave, poiché la distruzione dell'enclave è comunque eseguita al termine del programma, come il rilascio automatico delle risorse eseguito dal sistema operativo. Inoltre, la sicurezza dell'applicazione non risente della non distruzione dell'enclave. Infatti, è la parte di codice non protetta a richiamare la creazione e la distruzione dell'enclave. Dal punto di vista di SGX, tutto il codice che viene eseguito al di fuori dell'enclave non è sicuro. Questo significa che non bisogna fare affidamento nè sul sistema operativo nè sul codice sorgente non protetto dell'applicazione. Inoltre, la regione di memoria dell'enclave è protetta con proprietà di riservatezza, come descritto in Sezione 2.1.2. Per queste motivazioni il sistema operativo può essere delegato alla distruzione dell'enclave senza essere eseguita direttamente dalla nostra applicazione.

Controllo compatibilità dell'hardware L'ultima funzione, si occupa di controllare la compatibilità hardware del sistema con Intel SGX. Questa utilizza i privilegi di amministratore per interrogare la libreria del framework e valutare il risultato dell'interrogazione. Nel caso l'eseguibile non sia aperto con i privilegi di amministratore la funzione avvisa l'utente di rieseguire il programma con i privilegi necessari in modo da ricevere informazioni più dettagliate in caso di errore. Il codice di ritorno dell'interrogazione può assumere molteplici valori e per poter dare un'informazione precisa relativa all'errore è necessario implementare un codice lungo che gestisca più casi possibili. La funzione prende ispirazione da quella definita nei tutorial di familiarizzazione consultabili sul sito dell'architettura[9].

I vari messaggi d'errore, informano l'utente di problemi di compatibilità causati da hardware non compatibile con SGX o da errate configurazioni del sistema, come per esempio il supporto di SGX disabilitato all'avvio del sistema. È quindi necessario, una volta compilata l'applicazione, verificare gli eventuali messaggi di errore forniti sulla console in caso di mancato funzionamento. In questo modo, lo sviluppatore non è obbligato a consultare il manuale per trovare le informazioni sull'errore e di conseguenza velocizzare il rilascio dell'applicazione.

5.3.4 Salvataggio dei file

Il salvataggio dei file consiste nel scrivere su disco i file sorgente modificati e aggiunti dal framework. In particolare, il framework di conversione oltre a modificare i file dell'utente, spostando e implementando il codice per eseguire le chiamate ECall ed OCall, aggiunge al codice sorgente dell'utente i seguenti nuovi file:

[9] https://software.intel.com/content/www/us/en/develop/articles/properly-detecting-intel-software-guard-extensions-in-your-applications.html

- file sorgente dell'enclave: qui vengono spostate tutte le implementazioni delle funzioni sicure che vanno protette dall'enclave e vengono aggiunte tutte le funzioni di gestione che gestiscono i codici di stato delle chiamate OCall;
- file definizione dell'enclave: questo file in formato EDL è il codice in linguaggio proprietario di Intel SGX incaricato di definire l'ambiente protetto e le sue vie di ingresso ed uscita con l'ambiente esterno;
- libreria di gestione dell'enclave: questa libreria si occupa dell'allocazione dell'enclave, del controllo della compatibilità dell'hardware e della distruzione dell'enclave; gestisce i relativi errori alle chiamate citate e memorizza al suo interno i dati necessari per le chiamate ECall come per esempio il codice identificativo dell'enclave.

I file sorgente modificati e la libreria di gestione dell'enclave vengono memorizzati nella cartella di output scelta dall'utente all'inizio dell'esecuzione della conversione. Invece, i file dell'enclave (il file EDL di descrizione ed il file sorgente) vengono memorizzati in una cartella separata allo stesso livello della cartella dei sorgenti dell'utente. In questo modo si facilita la compilazione dato che i sorgenti dei due ambienti vengono compilati separatamente come spiegato in Sezione 5.4.

5.4 Generazione del binario protetto

In questa sezione si descrive come ottenere il binario protetto, attraverso la compilazione del codice modificato automaticamente dal framework. Il framework ha come obiettivo la generazione di codice compatibile con compilatori Linux/Windows. La generazione del binario protetto può avvenire attraverso l'utilizzo di GNU make oppure l'uso del programma Microsoft Visual Studio[10]. In definitiva, come spiegato nella Sezione 2.2.5, Intel SGX supporta diverse modalità operative per l'esecuzione dell'applicazione: debug, pre-release, release e simulata.

Queste modalità sono supportate sia da GNU make che da Visual Studio. Tuttavia, quest'ultimo facilita l'utente nella compilazione del programma generando i file necessari alla firma dell'enclave.

5.4.1 Compilazione con Visual Studio

Intel ha sviluppato un plugin per Visual Studio che gestisce una modalità guidata per la creazione di soluzioni con Intel SGX. Questa permette di aggiungere i file dell'enclave generati dal framework ed eventualmente specificare la chiave privata RSA se necessaria dalla modalità SGX usata. Viene quindi creata una soluzione protetta da importare in un progetto di Visual Studio contenente tutto il codice sorgente non protetto. Le istruzioni complete per importare e compilare il progetto con Visual Studio sono nella Appendice A.4.

[10] https://visualstudio.microsoft.com/it/vs/

5.4.2 Compilazione con il GNU make

Questa soluzione è da preferirsi per sistemi operativi Linux dove lo strumento GNU make è utilizzato per le gestione della compilazione di un programma da sorgente. Al fine di compilare il progetto è necessario avere installato sulla macchina l'insieme di librerie e strumenti forniti con Intel SGX SDK[11]. Il framework di conversione è fornito con un file per GNU make configurabile dall'utente per la compilazione dell'applicazione. Le istruzioni complete per la modifica del file di GNU make sono disponibili nella Appendice A.3.

[11] https://software.intel.com/en-us/sgx/sdk

Capitolo 6

Risultati sperimentali

In questo capitolo si descrivono i test che sono stati eseguiti per valutare le prestazioni del framework. Questi utilizzano pacchetti di codice per analizzare sia l'efficienza a runtime del codice eseguito in ambiente protetto sia l'efficienza del convertitore di sorgente. Per il primo scopo si è proceduto a convertire ed eseguire pacchetti di codice esistenti dei sistemi Debian[1], come descritto in Sezione 6.1.1. Inoltre, si è generato un ulteriore pacchetto di codice per eseguire un algoritmo di ordinamento in enclave, come spiegato in Sezione 6.1.2.

L'esecuzione di questi pacchetti convertiti per SGX viene confrontata con l'esecuzione del codice originale per valutare l'impatto prestazionale della conversione. Viene quindi, confrontato il tempo di esecuzione dei vari codici, come analizzato in Sezione 6.3.

Successivamente, si descrivono i pacchetti di codice utilizzati per valutare l'efficienza del framework durante la conversione del sorgente. Questi, sono pacchetti generati da un modulo Python in grado di sviluppare codici validi di complessità variabile. Si analizzano le varie esecuzioni del framework su pacchetti di codice di complessità crescente e si ricercano le criticità del framework. Poi, si descrivono i casi di design del codice migliori e peggiori dal punto di vista dell'efficienza del convertitore. Si analizza quindi la memoria utilizzata dal framework ed il tempo di conversione, come spiegato in Sezione 6.4.

Infine, si descrivono i principali problemi dovuti alle limitazioni del framework e di SGX. Si analizzano i principali aspetti e si forniscono semplici soluzioni comuni applicabili ai problemi ricorrenti del codice, come descritto in Sezione 6.2.

6.1 Pacchetti di codice C convertiti con il framework

In questa sezione di descrivono i pacchetti di codice convertiti dal framework. La conversione del codice è eseguita su un computer compatibile con l'architettura SGX con le specifiche tecniche riportate in Tabella 6.1.

[1] https://www.debian.org/index.it.html

Specifiche	
Modello	Asus UX510UXK
CPU	Intel Core i7-7500U @ 2.70GHz
RAM	16 GB DDR4
Sistema operativo	Microsoft Windows 10 Pro

Tabella 6.1: Specifiche PC su cui sono eseguiti i test.

6.1.1 Pacchetti Debian

Al fine di testare il funzionamento del framework di conversione, si è deciso di convertire pacchetti di codice C Debian, il cui sorgente è disponibile gratuitamente[2]. Alcuni pacchetti provengono da un insieme di giochi eseguibili su linea di comando che sono raggruppati in un unico archivio[3]. Si è quindi proceduto a convertire alcuni pacchetti con funzionalità diverse, ottenendo risultati molto differenti per quando riguarda le performance in esecuzione, come descritto in Sezione 6.3. I pacchetti di giochi sono anche disponibili separatamente su un repository[4]. Si introducono quindi i pacchetti che sono stati convertiti, per esporne le loro funzionalità.

`Binclock`[5] è un software che mostra un orologio, raffigurato con una matrice di numeri binari. Riceve l'ora corrente e la converte in una tabella di numeri 0 e 1.

`Morse`[6] è un eseguibile che riceve in ingresso caratteri che si vogliono trasmettere tramite codici dell'alfabeto Morse. Esegue delle trasformazioni per convertire i caratteri e stampare in output le rappresentazioni dei dati richiesti.

`BCD`[7] legge da linea di comando una frase di testo ASCII e produce in uscita una o più figure testuali che simulano delle schede perforate. Le rappresentazioni emulano con i caratteri la posizione dei buchi per memorizzare frammenti della frase sulle schede.

`POM`[8] è un codice sorgente che utilizza una chiamata al sistema operativo per ricevere la data e l'ora corrente e calcola da essa la percentuale della fase lunare.

`Banner`[9] è uno strumento che converte dei dati testuali ASCII in scritte orizzontali formate dai carattere '#'.

[2] https://packages.debian.org/stretch/

[3] https://packages.debian.org/jessie/bsdgames

[4] https://salsa.debian.org/games-team/bsdgames/-/tree/master

[5] https://packages.debian.org/stretch/binclock

[6] https://salsa.debian.org/games-team/bsdgames/-/tree/master/morse

[7] https://salsa.debian.org/games-team/bsdgames/-/tree/master/bcd

[8] https://salsa.debian.org/games-team/bsdgames/-/tree/master/pom

[9] https://salsa.debian.org/games-team/bsdgames/-/tree/master/banner

6.1.2 Pacchetti test generati

Oltre ai pacchetti Debian (descritti in Sezione 6.1.1), sono stati convertiti ulteriori codici creati appositamente per testare sia le prestazioni a runtime sia i limiti di computazione del framework. Per il primo scopo, si è ritenuto interessante analizzare questo ulteriore codice sorgente.

`Bubble sort` è un pacchetto che riceve in ingresso il nome di un file contenente i numeri da ordinare con l'omonimo algoritmo di ordinamento bubble sort. Il file di ingresso specifica la quantità di numeri presenti ed un ulteriore parametro che permette di eseguire l'algoritmo dividendo il numero totale di elementi con un passo. Quest'ultimo definisce il numero di elementi da ordinare per esecuzione e rappresenta quindi la dimensione dei vettori da ordinare. L'insieme dei numeri totale viene diviso per il passo e si formano piccoli gruppi di numeri di dimensione passo. Si procede quindi ad ordinare singolarmente i vettori. Questa opzione è utile per poter variare il numero di volte che si entra ed esce dall'algoritmo di ordinamento, che nel caso di Intel SGX corrisponderà ad un'esecuzione di funzione in enclave.

Per testare le prestazioni ed i limiti del framework di conversione invece, si è proceduto a costruire un ulteriore modulo Python che permette di generare dei programmi C, con il numero di funzioni e la complessità del programma arbitrarie. Questo modulo riceve in input un file contenente una lista di coppie di numeri. Questi interi connettono tra loro due nomi di funzioni in cui la prima invoca la seconda. Il modulo legge il grafo delle chiamate a funzione rappresentato dalle coppie e genera un codice in linguaggio C che lo rappresenta. Questo viene fatto generando la definizione e costruendo l'implementazione delle funzioni che si richiamano reciprocamente. Questo modulo non genera un codice con l'utilità di essere compilato ed eseguito, bensì con lo scopo di essere convertito dal framework come test per poter ricercare il limite massimo di complessità e di analizzare come l'annidamento e la grandezza del codice possano influire sul tempo e sulla memoria utilizzati dal framework durante la conversione. Quindi, questi programmi generati sono stati utilizzati per analizzare sia i limiti del framework che i limiti degli strumenti di analisi statica utilizzati (descritti in Sezione 4.2). Si è proceduto quindi a creare dei pacchetti appositamente grandi ed annidati a piacere per poter verificare queste limitazioni, come descritto in Sezione 6.4.

6.2 Problemi ricorrenti

Quando si esegue la conversione del codice C per l'architettura Intel SGX, è necessario dividere le parti interne ed esterne all'esecuzione protetta. Il framework di conversione facilita questa operazione, decidendo le funzioni che devono essere obbligatoriamente inserite all'interno dell'enclave, come descritto in Sezione 5.3.2. Tuttavia, la conversione richiede comunque che l'utente decida i punti di ingresso e di uscita dall'enclave. Questi sono necessari poiché il framework non è in grado di comprendere quali siano i dati e le funzioni sensibili del codice, ma è in grado di ricercare una soluzione compatibile con il flusso di esecuzione del codice, i limiti dell'architettura SGX e le richieste del programmatore. Quindi, il framework trova

le linee di divisione tra la parte protetta e non sicura del codice a seguito delle analisi del programmatore.

Per decidere i punti di ingresso dell'esecuzione protetta è necessario che lo sviluppatore conosca a pieno le funzionalità del programma da convertire. Solo in questo modo sarà in grado di decidere quali parti di codice siano a rischio e necessitino della protezione hardware. In via teorica si potrebbe proteggere tutto il codice creando opportunamente delle funzioni OCall, nel caso si necessiti di funzioni del sistema operativo non fornite direttamente dalle librerie di Intel SGX. Tuttavia, questo approccio è da sconsigliare ed è invece consono proteggere solo alcune parti di codice, limitando il più possibile l'esecuzione in ambiente protetto. Questo ideale è incoraggiato per limitare la perdita delle prestazioni dovute dall'esecuzione in ambiente protetto, come scritto in Sezione 2.1.6.

I programmi utilizzati per i test della conversione sono stati sviluppati senza seguire il modello della separazione degli ambienti specificato dall'architettura SGX spiegata in Sezione 2.2.1. Sono quindi necessarie alcune modifiche, imposte direttamente dalla compatibilità con SGX e con il framework. I problemi comuni che devono essere risolti dall'utente non possono essere risolti direttamente dal framework, poiché alcune limitazioni di Intel SGX richiedono la modifica del codice sorgente per poterlo riadattare senza variarne la funzionalità. Sono descritti quindi i conflitti più ricorrenti riscontrati dal framework durante la conversione e sono proposte soluzioni applicabili per gestirli.

6.2.1 Uso di costanti globali nelle funzioni da proteggere

Il codice da convertire può contenere definizioni di costanti globali articolate, come strutture dati o vettori di inizializzazione. Questi dati costanti, se utilizzati dalle funzioni protette, devono essere spostati o copiati all'interno del codice dell'enclave. A questo problema il framework di conversione fornisce due opzioni utili di copia e spostamento, come spiegato in Appendice A.1.2. Il framework non è ancora in grado di analizzare autonomamente i simboli globali utilizzati all'interno del codice. Per questa ragione si introducono le direttive di copia e spostamento delle costanti globali. Queste opzioni sono valide solo le per costanti globali e non suggeriscono soluzione per la condivisione di variabili globali in cui sono necessarie modifiche manuali, come spiegato in Sezione 6.2.2.

6.2.2 Uso di variabili globali nelle funzioni da proteggere

Le variabili globali condivise tra funzioni sicure e non sicure non sono consentite dall'architettura Intel SGX poiché, per definizione, tutti i dati inseriti nell'enclave devono essere protetti e non leggibili dall'esterno. È tuttavia possibile utilizzare la direttiva di spostamento del framework (definita in Appendice A.1.2) nel caso tali variabili siano solo utilizzate dalle funzioni sicure. I pacchetti Debian che sono stati sottoposti alla conversione per il test del framework, utilizzavano spesso le variabili globali condivise tra funzioni. Si è deciso di procedere con una strategia semplice che consiste nel mantenere le variabili globali solo nella parte non protetta del codice e di aggiungere dei parametri aggiuntivi alle funzioni sicure che le utilizzavano. Questa

opzione richiede la modifica da parte dell'utente poiché è necessaria la specifica di passaggio dei parametri alle funzioni protette, come descritto in Sezione 2.2.2. Inoltre, l'operazione non può essere automatizzata poiché il framework non è in grado di riconoscere se la variabile globale sia un segreto dell'applicazione e di conseguenza fornire una soluzione per proteggerla in enclave. Per questi motivi non è possibile gestire automaticamente la conversione di codice che condivide variabili globali tra i contesti sicuro e non sicuro. La complessità di tale automazione risiede nelle potenzialità della variabile globale. Infatti, questa non è protetta dalla località del codice e può essere letta e scritta da qualsiasi funzione. In linea teorica, la variabile va passata come argomento alle funzioni protette e nel caso queste abbiano bisogno di modificarla occorre valutare caso per caso come ritornare il valore modificato in modo che il chiamante (esterno all'enclave) possa aggiornarlo. Per questo motivo è necessario l'intervento dell'utente per definire le macro del framework, come descritto in Sezione 4.1.

6.2.3 Uso di funzioni standard non ridefinite dalle librerie SGX nelle funzioni da proteggere

Intel SGX ridefinisce le librerie standard-C da utilizzare all'interno del codice dell'enclave modificandone il contenuto e rimuovendo alcune funzioni non sicure come quelle di input ed output. Di conseguenza non tutte le funzioni di libreria standard-C sono utilizzabili all'interno dell'enclave, come scritto in Sezione 2.1.6. Questa limitazione connessa alla architettura SGX non può essere direttamente gestibile dal framework, poiché generare una chiamata OCall per ogni funzione non disponibile all'enclave comprometterebbe le prestazioni. Il programmatore può gestire il conflitto spostando la chiamata alla funzione non disponibile all'enclave nella parte di codice non protetta. A questo fine sono possibili due diverse alternative:

1. eseguire la chiamata alla funzione non consentita prima della chiamata ECall;
2. definire una nuova funzione OCall invocabile dall'enclave per poter eseguire la chiamata vietata nel codice non protetto.

La prima opzione consiste nel preporre la chiamata alla funzione di libreria alla chiamata ECall e passare eventualmente ulteriori dati negli argomenti della funzione protetta. Questa gestione ricade nel tipico caso di una funzione protetta che legge un input (e.g. caratteri da tastiera) durante la sua esecuzione. Questa operazione potrebbe essere eseguita dalla parte non sicura del codice, subito prima della chiamata ECall. Poi, i dati letti potrebbero essere passati come argomenti alla funzione protetta, in modo che questa non debba richiamare dal suo interno funzioni non standard delle librerie SGX.

La seconda opzione prevede la creazione di una funzione OCall per ogni funzione di libreria non disponibile invocata all'interno dell'enclave. Queste chiamate OCall hanno il solo scopo di essere intermediarie alle funzioni di sistema invocate dal loro interno. Questo approccio è valido poiché le funzioni OCall sono per definizione esterne all'enclave (come descritto in Sezione 2.2.1) e di conseguenza hanno accesso

completo alle librerie standard-C. Questa alternativa è meno performante della precedente poiché richiede una doppia transizione dall'enclave (uscita e rientro) ed inoltre, aggiunge una nuova chiamata a funzione. Tuttavia, non è sempre possibile affrontare il problema con la prima soluzione, che prepone la chiamata alla libreria all'entrata in enclave. Infatti, un conflitto potrebbe essere l'utilizzo della funzione protetta per processare uno stream di dati continuo in enclave e rilasciare il risultato all'esterno in modo frammentato e continuativo.

6.2.4 Uso della direttiva `const` negli argomenti delle chiamate ECall ed OCall

Come definito dallo standard SGX, il formato EDL impone delle limitazioni alla direttiva `const` negli attributi delle funzioni [12]. In generale, non è possibile utilizzare la direttiva `const` quando il parametro non è un puntatore e quando il metodo di passaggio del parametro è impostato su `out` (descritto in Sezione 2.2.2). Di conseguenza, l'implementazione delle funzioni ECall ed OCall deve essere adattata a questo limite. In caso contrario il framework avvisa l'utente del conflitto. Per tutte le altre funzioni è consentito l'utilizzo di questa direttiva.

6.2.5 Uso di funzione ECall ed OCall statiche

Le funzioni di limite dell'enclave (ECall ed OCall) non possono essere dichiarate statiche. Questa impostazione è un limite intrinseco del linguaggio C e dell'architettura di Intel SGX, poiché le funzioni definite statiche sono visibili solo alle funzioni del proprio file oggetto. Di conseguenza, i due ambienti protetto ed esposto, che sono fisicamente memorizzati in file separati, non possono avere visibilità sulle chiamate ECall ed OCall definite statiche. Per questo motivo non trova significato l'esportazione di una funzione ad altri file se essa è definita statica e per definizione privata al suo file oggetto[10]. È quindi necessario rimuovere la direttiva `static` per le funzioni ECall ed OCall. Può invece rimanere per tutte le altre funzioni, anche all'interno del codice dell'enclave.

6.2.6 Uso di sintassi C precedente allo standard C11

La sintassi supportata dal framework è compatibile con lo standard `ANSI C`. Di conseguenza i pacchetti che utilizzano sintassi precedenti non possono essere direttamente convertiti con il framework, ma è necessario un aggiornamento del codice. I pacchetti Debian utilizzati per i test di conversione utilizzano la vecchia sintassi `K&R`. Il framework non riconosce il codice in linguaggio C con tale sintassi ed è stato quindi necessario riformulare la vecchia sintassi nella nuova. In Figura 6.1 sono rappresentate l'implementazione di una funzione in formato `K&R` e `ANSI C`.

[10] https://www.tutorialspoint.com/static-functions-in-c

```
// ANSI syntax
int sum(int a, int b)
{
        return a + b;
}

// K&R syntax
int sum(a, b)
        int a;
        int b;
{
        return a + b;
}
```

Figura 6.1: Esempio di sintassi `ANSI C` e sintassi `K&R`.

6.3 Analisi delle prestazioni del codice protetto in esecuzione

In questa sezione si analizzano le prestazioni dei programmi convertiti con il framework per l'utilizzo di Intel SGX. In particolare si esamina il comportamento di un algoritmo di ordinamento al variare della dimensione dei dati da ordinare. Poi, si eseguono pacchetti di codice che generano in output del testo e si confrontano i tempi di esecuzione standard e con esecuzione sicura. Infine, si dimostra come il design del codice, possa portare ad una perdita di performance molto pronunciata.

6.3.1 Algoritmo di ordinamento

Come introdotto nella Sezione 6.1.2, è stato generato e convertito un pacchetto di codice che esegue un algoritmo di ordinamento. Si è scelto l'algoritmo di ordinamento Bubble sort poiché semplice da implementare e si è impostato il codice al fine di eseguirlo per ordinare vettori di numeri interi generati casualmente. La parte non protetta dall'enclave legge i dati da un file e richiama l'algoritmo più volte. I dati sono forniti dal file che specifica 10^5 numeri interi ed un passo. Il passo specifica la dimensione del vettore da ordinare e l'insieme dei numeri viene diviso per il passo. Di conseguenza si generano un numero pari a $10^5/passo$ da ordinare singolarmente. In questo modo, è possibile avere valori medi su esecuzioni brevi di ordinamenti con vettori di piccole dimensioni. In questo modo è possibile confrontare il tempo di esecuzione del codice standard-C con il codice che ordina i vettori in enclave.

Nella Figura 6.2 sono disponibili i rapporti tra le coppie di tempi di esecuzione. Per ampliare i risultati, si è eseguito l'algoritmo anche su un vettore di 10^6 elementi. Come è possibile notare, il tempo di esecuzione del programma compilato per

SGX è minore ed il rapporto tra i tempi è più pronunciato verso vettori maggiori dimensioni.

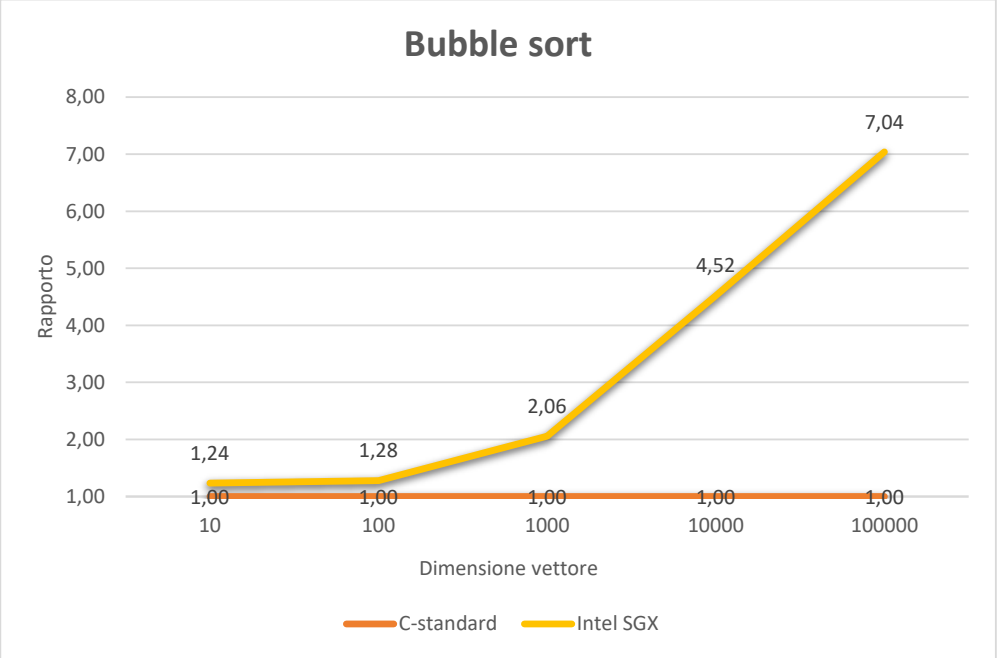


Figura 6.2: Rapporto tra i tempi di esecuzione di Bubble sort SGX e standard-C al variare di dimensione del vettore.

6.3.2 Convertitori di testo

In questa sezione si analizzano le prestazioni delle funzioni che ricevono del testo in input, lo trasformano e stampano il risultato in console. In particolare si eseguono i test sui pacchetti `Morse`, `BCD` e `Banner`.

Il pacchetto `Morse` riceve su linea di comando un testo da convertire in codice Morse ed esegue la trasformazione in enclave. Allo stesso modo il codice `BCD` genera la rappresentazione di schede perforate che rappresentano il testo letto dalla linea di comando. Infine, il programma `Banner` genera una rappresentazione del testo con il simbolo `#` e lo mostra su console.

Questi pacchetti si comportano in modo simile sia nel flusso di esecuzione che nell'andamento dei tempi confrontati con le esecuzioni di codice non sicuro. Nelle Figure 6.3, 6.4 e 6.5 è possibile notare che l'esecuzione protetta con SGX richiede più tempo dell'esecuzione standard. Questo, è principalmente causato da tre fattori. Il più intuitivo è la riduzione di prestazioni dovuta all'esecuzione del codice aggiuntivo di gestione delle chiamate ECall ed OCall, e all'esecuzione rallentata per l'uso della memoria cifrata. Il secondo motivo è il tempo di inizializzazione dell'enclave, compreso il tempo di caricamento della sua libreria dinamica, come spiegato in Sezione 2.2.1. Il terzo motivo è il cattivo design del codice fornito alla conversione.

Infatti, il codice convertito dal framework non è pensato per l'architettura SGX e per questo motivo le prestazioni in ambito protetto sono molto deteriorate. Tuttavia, lo scopo di questi test non è quello rendere il codice sicuro performante,

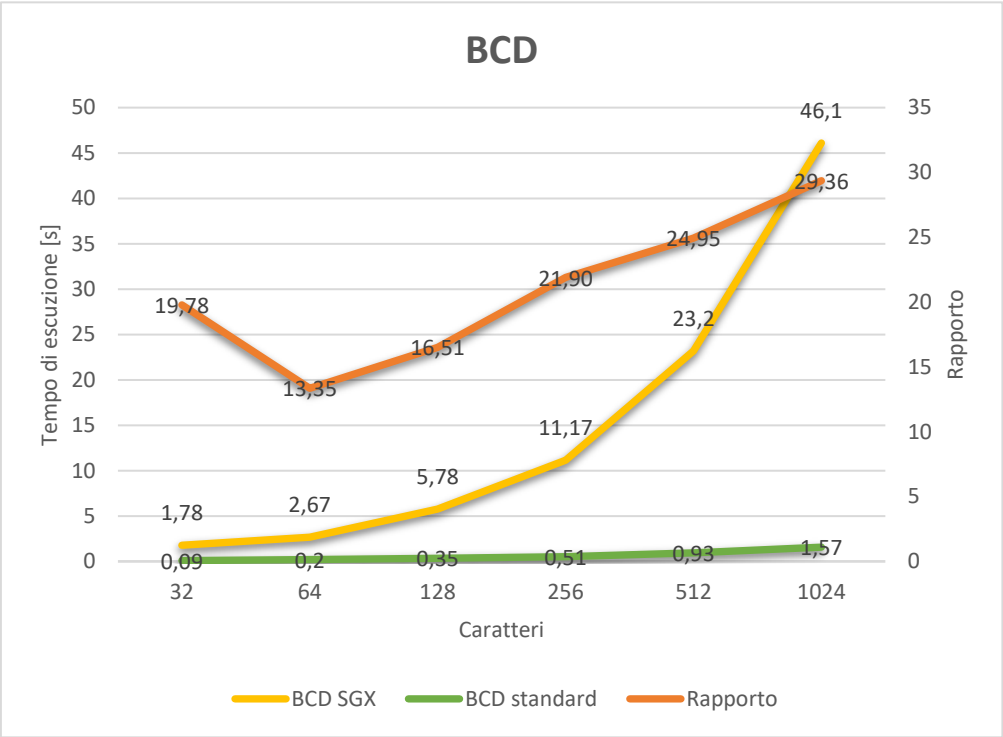


Figura 6.3: Pacchetto `BCD` eseguito per numero crescente di caratteri in input.

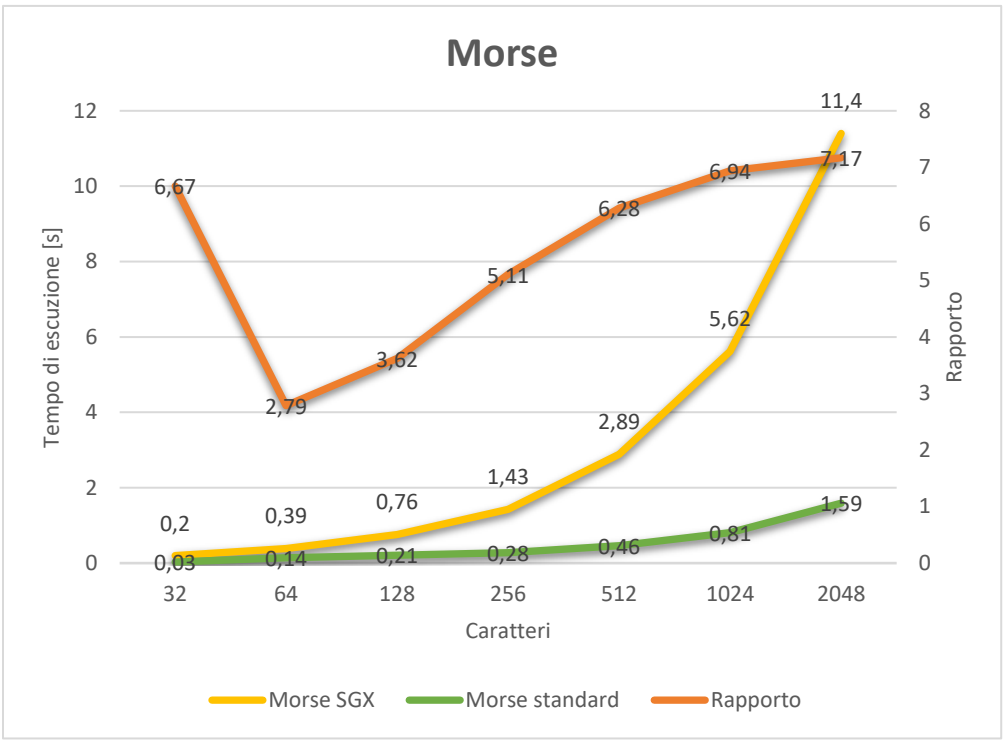


Figura 6.4: Pacchetto `Morse` eseguito per numero crescente di caratteri in input.

bensì di modificare il codice originale il meno possibile ed eseguire su di esso il framework di conversione. In questo modo è possibile valutare le performance del codice sorgente modificato dal programmatore solo al fine di essere compatibile con

le limitazioni del framework e di SGX. I pacchetti presentati, utilizzano la funzione di libreria standard-C `putchar` per stampare i caratteri su console. Questo, è il principale motivo della perdita di prestazioni così elevata. Infatti, la funzione di stampa dei caratteri non è direttamente disponibile alle funzioni dell'enclave e per questa ragione si è definita una chiamata OCall in modo da accedere alla funzione `putchar`, come descritto in Sezione 6.2.3. Riassumendo, i programmi convertiti per SGX ricevono da linea di comando del testo. Quest'ultimo è fornito a delle funzioni dell'enclave attraverso chiamate ECall. Queste funzioni, generano molto output testuale ed usano una chiamata OCall per stampare il singolo carattere su console. Quindi, le performance di questi programmi risultano molto deteriorate, poiché per ogni singolo carattere generato in output è necessario eseguire una chiamata OCall. Di conseguenza, si esegue una doppia transizione (uscita e rientrata nell'enclave) per la stampa di ogni carattere.

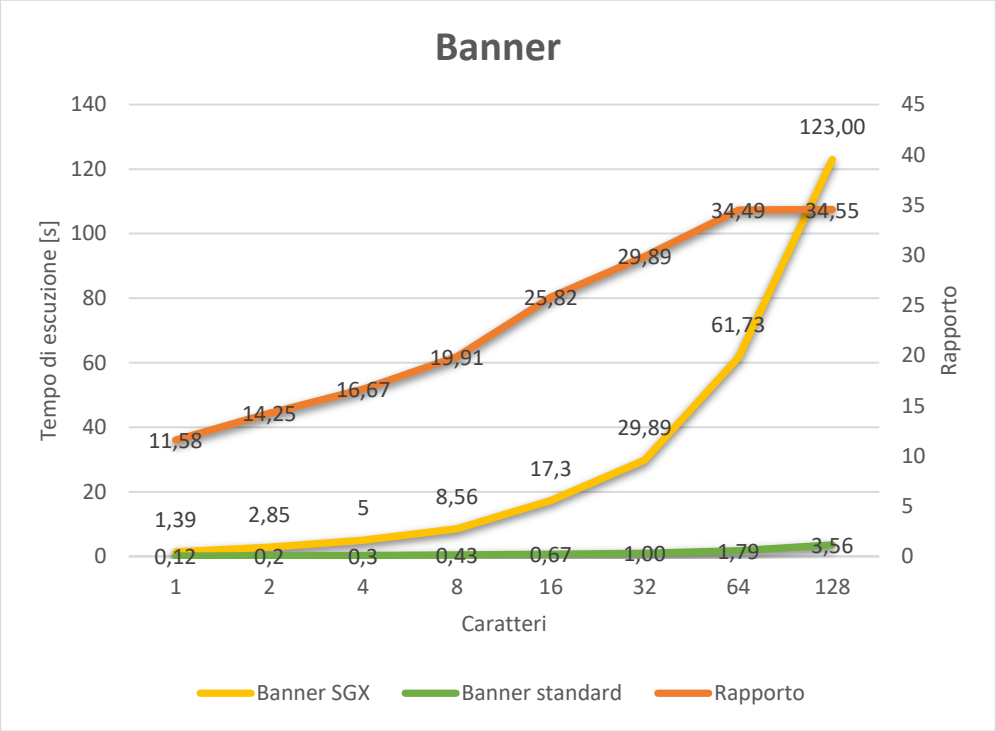


Figura 6.5: Pacchetto `Banner` eseguito per numero crescente di caratteri in input.

Nel caso migliore (pacchetto `Morse` con input 64 caratteri), il tempo di esecuzione è di 2,79 volte il tempo necessario all'esecuzione non protetta, come visibile in Figura 6.4. Nel caso peggiore (pacchetto `Banner` con input 64 caratteri), il tempo di esecuzione in ambiente protetto risulta essere circa 35 volte del tempo in esecuzione non protetta, come rappresentato in Figura 6.5. Il numero di stampe e di conseguenza il numero di volte in cui è necessario eseguire una chiamata OCall influisce notevolmente il tempo di esecuzione. Infatti, i casi peggiori risiedono nei programmi `Banner` e `BCD` che sono ricchi di testo in output, mentre la perdita di prestazioni è meno pronunciata nel caso del pacchetto `Morse`.

Un altro aspetto da considerare è la comune crescita del rapporto tra i tempi eseguendo il codice con dati più corposi, escluso il primo valore. Il rapporto decresce

tra il primo e il secondo valore della Figura 6.3 e della Figura 6.4 poiché il primo valore è affetto dal tempo di caricamento e generazione dell'enclave alla prima chiamata ECall. Con esecuzioni più lunghe, questo tempo è di minore rilevanza e influisce inferiormente sul valore del rapporto. Nell'esecuzione del programma `Banner` di Figura 6.5, non si ritrova la discesa dei valori del rapporto tra la prima e la seconda esecuzione. Questo è dovuto alla lunga esecuzione del programma eseguito con un solo carattere.

6.4 Analisi delle prestazioni durante la generazione del codice protetto

In questa sezione si descrivono i test eseguiti per analizzare le prestazioni del framework di conversione. I parametri di interesse sono il tempo di esecuzione e la memoria utilizzata durante la conversione dei programmi.

Come introdotto in Sezione 6.1.2, si è utilizzato un modulo Python per la generazione di codici sorgenti complessi e grandi a piacere. Il modulo costruisce il codice sorgente a partire da un grafo delle chiamate memorizzato su file. I tipi di grafi utilizzati sono rispettivamente i due casi migliore e peggiore nel contesto dell'analisi statica.

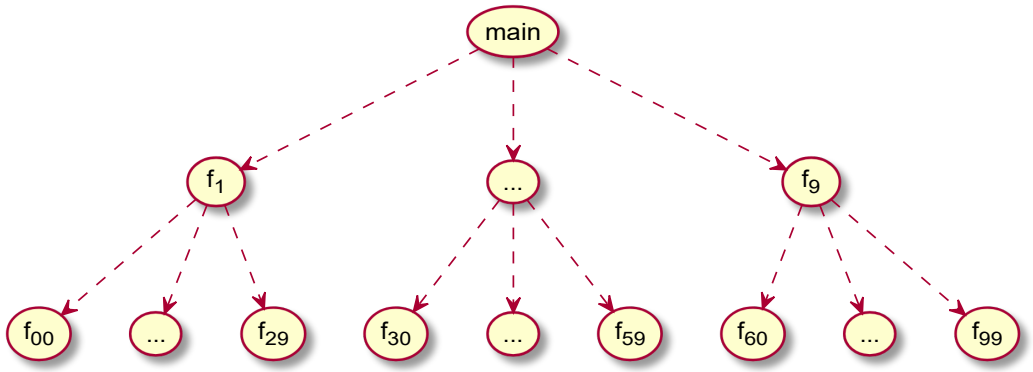


Figura 6.6: Modello di grafo delle chiamate ad albero con profondità massima pari a due.

Il caso migliore è il grafo delle chiamate piatto che consiste in un albero di chiamate con massima profondità pari a 2, come visibile in Figura 6.6. Il caso peggiore invece è un albero di chiamate con profondità pari esattamente a $\#nodi - 1$, come raffigurato in Figura 6.7. Questo secondo tipo di grafo è più pesante da analizzare, poiché richiede più livelli di ricorsione durante l'esplorazione dei nodi.

In Figura 6.8, è possibile notare un grafico che mostra il tempo di conversione al variare del numero di funzioni presenti nel codice sorgente. La crescita del tempo di conversione risulta costante con l'aumentare del numero di funzioni. Infatti, la complessità del codice non risiede solo nel numero di funzioni definite, ma anche nelle loro interconnessioni.

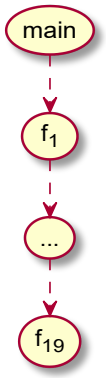


Figura 6.7: Modello di grafo delle chiamate ad albero con profondità $\#nodi - 1$.

In Figura 6.9, si analizza l'utilizzo di memoria RAM durante la conversione nel caso peggiore. Come è possibile notare, l'uso della memoria segue una curva esponenziale crescendo con il numero di funzioni. Il livello di annidamento del codice influisce sulla memoria fino a rendere il codice non convertibile. Infatti, i test effettuati rilevano che la profondità massima dell'albero non può superare le 19 unità, altrimenti la conversione termina con errore. Questo problema non è causato direttamente dal codice del framework, ma dall'analizzatore di codice statico Frama-C. Questo strumento ha come limite massimo 19 livelli di annidamento, dopodiché non riesce a portare a termine l'analisi richiesta.

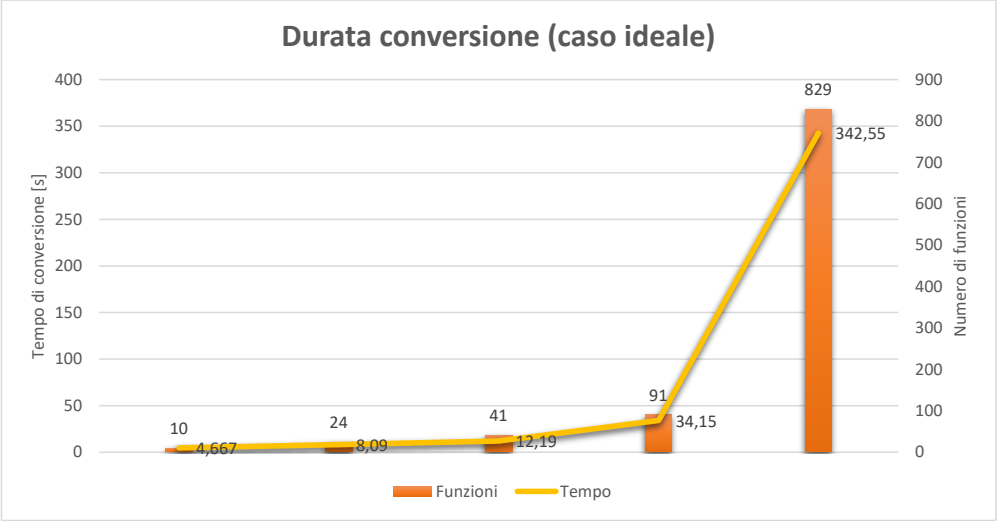


Figura 6.8: Durata della conversione in base al numero di funzioni nel caso ideale.

Un simile comportamento della curva di utilizzo della memoria si ritrova nel tempo di esecuzione quando il grafo è un albero sviluppato in profondità. Come mostrato in Figura 6.10, il framework di conversione risulta evidentemente più lento a convertire questo tipo di design del codice. Questo comportamento è dovuto nuovamente all'analisi semantica effettuata dall'analizzatore statico Frama-C.

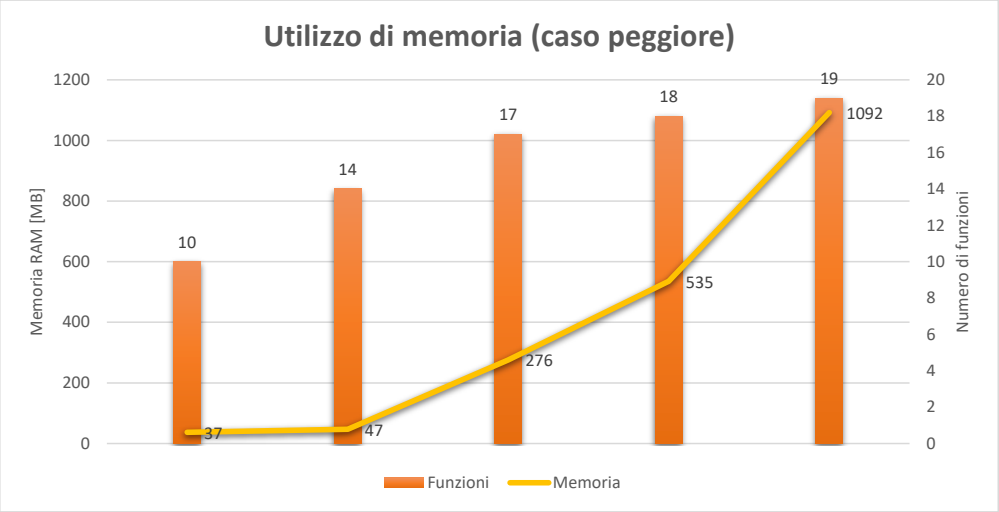


Figura 6.9: Memoria usata in conversione in base alla profondità dell'albero.

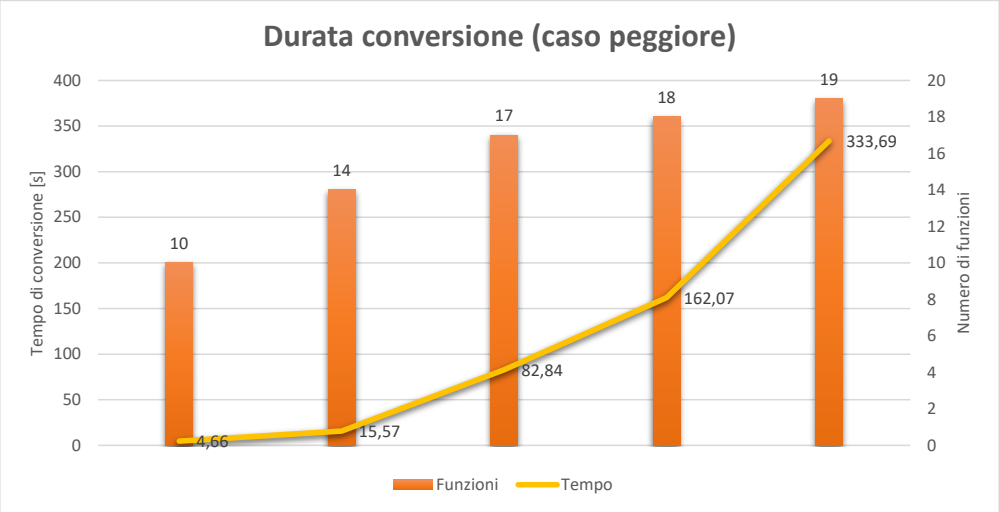


Figura 6.10: Durata della conversione in base alla profondità dell'albero.

Una configurazione del framework che potrebbe ridurre la memoria ed il tempo necessario alla conversione è l'utilizzo dell'analisi sintattica piuttosto che semantica offerta da Frama-C. Tuttavia, la modalità sintattica non è in grado di analizzare semanticamente il flusso del codice e di conseguenza comprendere puntatori a funzione e costrutti `if`, `while` e `for`. Utilizzando l'opzione di analisi sintattica quindi, si otterrebbe un risultato non ottimale che nel caso di puntatori a funzioni sicure in contesto non sicuro, porterebbe alla generazione di codice non sintatticamente corretto.

Un altro modo per migliorare le prestazioni potrebbe essere ricercare strumenti di analisi del codice più performanti, o sviluppare questo tipo di analisi direttamente all'interno del codice sorgente del framework senza l'uso di strumenti esterni. In conclusione, un'ultima miglioria del framework potrebbe essere l'utilizzo del multithreading, per eseguire il problema del calcolo del grafo delle chiamate (descritto in Sezione 5.3.2) su thread separati. Dato che questo calcolo viene effettuato per

ogni funzione e non dipende da altri dati, si potrebbe ridurre il tempo di calcolo dividendo questo tipo di analisi sui diversi core della CPU.

Capitolo 7

Lavori collegati

In questo capitolo si analizzano i lavori presenti in letteratura riguardanti la protezione del software. In particolare, la Sezione 7.1 introduce vari tipi di protezione del codice, quali protezioni da reverse engineering, da manomissione e da processi di debug. In Sezione 7.2 si analizzano alcuni strumenti di protezione automatica del codice, come Diablo e Tigress, in grado di attuare alcune tecniche di offuscamento del sorgente (descritte in Sezione 7.1.1). Inoltre, viene definito ESP, uno strumento avanzato di protezione del codice, in grado di selezionare e di applicare automaticamente le migliori tecniche, assicurando le proprietà di sicurezza richieste dall'utente. In Sezione 7.3 si introducono alcuni utilizzi pratici dell'architettura Intel SGX, come la protezione delle configurazioni delle applicazioni distribuite e la protezione dei contenitori Docker. Infine, la Sezione 7.4 descrive Glamdring, un framework in grado di modificare il codice sorgente di un'applicazione scritta in linguaggio C, al fine di renderla compatibile con le protezioni offerte da Intel SGX.

7.1 Tecniche di protezione del codice

Per proteggere un'applicazione con le tecniche di offuscamento, è necessario modificarne il codice. Questa operazione deve essere svolta in modo da mantenere inalterata la semantica dell'applicazione, ovvero la correttezza dell'esecuzione e dei risultati.

Tuttavia, le tecniche di offuscamento sono complesse da eseguire manualmente e modifiche anche ristrette del codice eseguite manualmente possono facilmente portare all'inserimento di falle nel sorgente che determineranno comportamenti indesiderati durante l'esecuzione dell'applicazione. Per questo motivi, le tecniche di protezione del codice si implementano utilizzando strumenti automatici (descritti in Sezione 7.2) che velocizzano il lavoro ed evitano l'inserimento di nuovi bug.

Vengono quindi presentate alcune tecniche automatiche per la protezione del codice tra cui la difesa dalle tecniche di reverse engineering, spiegata in Sezione 7.1.1, dalle tecniche di manomissione, descritta in Sezione 7.1.2 e dalle tecniche di debug del codice, analizzata in Sezione 7.1.3.

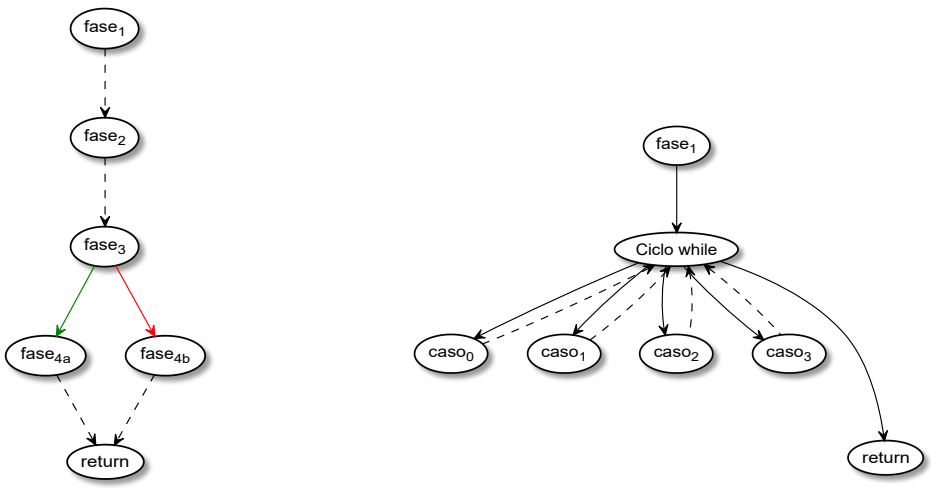
7.1.1 Protezione da reverse engineering

Il reverse engineering [18] del software è il processo che mira ad identificare le componenti del programma, le loro relazioni e le loro interazioni, al fine di rappresentare queste informazioni ad un alto livello di astrazione. Le società di software devono contrastare il reverse engineering delle loro applicazioni in modo tale da preservarne il valore commerciale. Tuttavia, non è sufficiente rilasciare i soli file binari dell'applicazione poiché gli attaccanti possono ottenere il codice macchina di file binario, mediante un disassemblatore, o possono eseguire una ricostruzione del codice sorgente originale dell'applicazione, usando un decompilatore (come lo strumento Hex-Rays Decompiler[1]).

Le tecniche per contrastare il reverse engineering applicano delle trasformazioni al codice per offuscare le sue parti sensibili. Le trasformazioni devono manipolare il codice, con l'obbiettivo di renderlo di difficile comprensione da parte di un attaccante, senza alterarne però la correttezza di esecuzione. Con queste tecniche è anche possibile applicare l'offuscamento ai dati costanti e alle variabili del codice al fine di mascherarne il proprio valore reale. In questo modo, il valore delle costanti e il valore delle variabili non sono esplicitamente comprensibili sia analizzando staticamente il codice (Sezione 3.1) con un disassemblatore che analizzando dinamicamente il codice (Sezione 3.2) con un debbugger.

Si introducono quindi alcune tecniche per l'offuscamento del codice e dei dati che sono fornite dagli strumenti `DIABLO` e `Tigress`, descritti in Sezione 7.2.

Control Flow Flattening (CFF) Questa tecnica [19] è un tipo di trasformazione che nasconde il flusso originale di un frammento di codice, come una funzione. Il flusso originale deve avere un singolo punto di ingresso ed uscita. Con la tecnica, la sequenza di fasi della parte di codice selezionata, viene convertita in un ciclo di casi, come mostrato in Figura 7.1.



(a) Prima. (b) Dopo.

Figura 7.1: Applicazione della tecnica CFF.

[1] https://www.hex-rays.com/products/decompiler/

Opaque predicates Questa tecnica [20] utilizza dei predicati booleani, strutturati per essere difficili da interpretare in analisi statica, che una volta valutati restituiscono sempre lo stesso valore booleano (vero o falso). L'operazione consiste quindi nell'inserimento di dead code[2] al fine di aumentare la quantità di codice che l'attaccante deve analizzare e di conseguenza rallentare il processo di reverse engineering. In Figura 7.2 è mostrato un esempio di predicato opaco. La funzione `opaque` ritorna sempre il modulo (in base due) del valore passato alla chiamata precedente, ovvero, il resto della divisione per due del valore passato alla chiamata precedente. Di conseguenza, il valore di ritorno della chiamata successiva a `opaque(1024)` sarà 0, mentre per la chiamata successiva a `opaque(3)` sarà 1. La Figura 7.2 mostra i costrutti `if` che valutano i predicati opachi al fine di aggiungere dead code e rendere il codice più ampio.

```
...

opaque(1024);
if(opaque(3) == 0){
        // percorso valido
}else{
        // dead code
}

if(opaque(25) == 1){
        // percorso valido
}else{
        // dead code
}

...

int opaque(int num){
        static int opaqueVar = 0;

        int tmp = opaqueVar;
        opaqueVar = num % 2;
        return tmp;
}
```

Figura 7.2: Esempio di predicato opaco.

Branch functions Con questa tecnica si rende più complessa la trasformazione effettuabile da un disassemblatore nella conversione di un binario in un codice

[2]Il dead code è l'insieme delle istruzioni del codice che non verranno mai eseguite.

macchina interpretabile dall'attaccante. La tecnica si basa sul nascondere gli indirizzi delle istruzioni di salto con l'utilizzo di puntatori relativi o con l'utilizzo di funzioni di hash[3] [21].

Virtualization obfuscation Con questa tecnica [22] si modificano i codici delle istruzioni macchina per nasconderne i veri valori. Gli opcode, ovvero i valori dei codici macchina, vengono sostituiti con altri valori (anche diversi per lo stesso codice di partenza) in modo che un analisi del codice macchina non possa rilevare direttamente le istruzioni invocate. Si mantiene il flusso e la struttura del codice inalterata, tuttavia in fase di esecuzione, i codici macchina devono prima essere preprocessati da un interprete in modo da ripristinare i valori originali dei codici macchina, per poi essere eseguiti dalla CPU.

Data obfuscation Questa tecnica impedisce la memorizzazione esplicita delle costanti, come stringhe e numeri, nel file binario. Questi valori vengono recuperati attraverso delle funzioni dette *encoder* che in esecuzione riescono a generare il valore delle costanti. Inoltre, la tecnica utilizza delle funzioni matematiche per codificare i valori delle variabili in memoria e questi vengono decodificati solo prima del proprio utilizzo. In questo modo sia le costanti che le variabili sono più complesse da analizzare con entrambi i metodi di analisi del codice, statica e dinamica.

7.1.2 Protezioni da manomissione

La manomissione (o tampering) è il processo con cui l'attaccante effettua delle modifiche non autorizzate al codice, causando danni alle società di software o agli utenti. Questo tipo di alterazione del codice viene effettuato dagli attaccanti sia su codice proprietario che su codice open source. Per esempio, il *crack* di un'applicazione proprietaria è una manomissione molto comune nei software a pagamento, in cui lo scopo dell'attaccante è quello di rimuovere o eludere la verifica della licenza. L'attaccante quindi causa un danno economico all'azienda in termini di mancate vendite del software, a causa della diffusione di copie illegali basate sul suddetto crack. Inoltre, la copia manomessa può causare un danno anche alla macchina su cui è installata, infettandola con dei malware. La manomissione del software open source gratuito è l'installazione di malware all'interno del codice al fine di distribuire l'applicazione infetta agli utenti. In questo modo l'attaccante può per esempio installare un Remote Access Trojan (RAT)[4] al fine di poter controllare la macchina su cui è installato.

Per questi motivi, c'è interesse nella protezione del software da manomissioni. L'obbiettivo di questa protezione è rendere il software inutilizzabile a seguito di modifiche non autorizzate. Durante l'esecuzione il programma controlla il proprio codice e si arresta in caso di modifiche al sorgente. Si specifica che questo controllo deve essere eseguito durante tutta l'esecuzione e non solo alla partenza del

[3] Una funzione di hash è un'operazione unidirezionale che genera un blocco di dati di lunghezza costante in funzione di un messaggio in ingresso.

[4] https://owasp.org/www-community/attacks/Trojan_Horse

programma. Questo è implicito al fatto che l'attaccante potrebbe compromettere il codice sia staticamente prima di eseguirlo che dinamicamente attraverso l'uso di un debugger.

Si descrivono quindi alcune tecniche per la protezione da manomissione.

Controllo dello stack delle chiamate [23] All'interno delle funzioni sensibili, vengono controllati gli indirizzi di ritorno delle funzioni presenti nello stack. In questo modo è possibile verificare se la chiamata della funzione sensibile è stata invocata dall'interno delle funzioni consentite o è stata eseguita forzatamente alterando l'esecuzione del codice da un attaccante.

Attestazione remota [24] L'attestazione è una tecnica utilizzata per verificare se il codice di un'applicazione è stato manomesso. Questa tecnica può essere implementata via software o via hardware come ad esempio l'attestazione remota di Intel SGX descritta in Sezione 2.5.

Code mobility [25] Questa tecnica prevede lo spostamento di blocchi sensibili di codice in modo che questi vengano eseguiti su un server remoto fidato. Con questa strategia si impedisce ad un attaccante remoto analizzare staticamente il codice e di manometterlo attraverso l'uso di debugger.

7.1.3 Protezioni da tecniche di debug

I debugger sono software che si collegano all'applicazione in esecuzione per ispezionare il suo comportamento. Questi possono essere quindi utilizzati sia per il reverse engineering sia per la manomissione del codice in esecuzione. Infatti, ispezionare il comportamento del flusso di un'applicazione e collezionare tracce di esecuzione da analizzare dopo l'esecuzione dell'applicazione, possono facilitare l'attaccante nel comprendere il funzionamento ed i componenti dell'applicazione. Inoltre, con un debugger è possibile modificare la memoria dell'applicazione al fine di manomettere porzioni di dati o di codice e cambiarne il comportamento durante la sua esecuzione.

Per questi motivi, si ricerca una modalità di protezione del codice contro i debugger utilizzati ai fini citati. Un modo per proteggere l'applicazione da questo tipo di attacco è il *self-debugging*[26], una tecnica in cui l'applicazione può continuare l'esecuzione solo se connessa al proprio strumento di debug. L'applicazione è divisa in due parti: un programma ed un debugger apposito. Quando viene aperta l'applicazione, vengono eseguite entrambe le parti e il debugger si connette immediatamente all'applicazione, occupando l'unica interfaccia di debug disponibile[26]. Poi, viene impedito all'attaccante di forzare la rimozione del debugger e collegare il proprio, spostando alcune parti fondamentali della logica dell'applicazione nel debugger fidato. In questo modo, l'esecuzione del programma dipende dal processo di debug collegato, per cui la sua rimozione termina l'esecuzione del programma.

7.2 Strumenti di protezione del software

L'obbiettivo della protezione delle applicazioni software è ritardare il più possibile l'attaccante nel compromettere le risorse del programma, cambiando la sua

funzionalità o rubando informazioni sensibili. Nello scenario Man-At-The-End (MATE), tutti i programmi sono attaccabili ipotizzando tempo e risorse adeguati, infatti, è dimostrato che le protezioni del codice non possono essere a prova di manomissione[27]. Di conseguenza, nello scenario MATE si può solo cercare di ritardare l'attacco per un tempo sufficientemente ampio da rendere l'attacco economicamente insostenibile. La ricerca aspira ad un miglioramento nella protezione delle risorse, mantenendo una degradazione delle prestazioni accettabile. Si cerca quindi di raggiungere il miglior compromesso tra prestazioni e sicurezza, in modo da mantenere l'esperienza di utilizzo dell'applicazione protetta quasi inalterata o simile all'applicazione originale.

In questa sezione si introducono quindi tre strumenti utili alla protezione del codice. In Sezione 7.2.1 si descrive lo strumento `DIABLO` che permette una protezione del codice inserita nella fase di costruzione del binario dell'applicazione. La Sezione 7.2.2 definisce un secondo strumento, denominato `Tigress`, che introduce la protezione del codice nel sorgente dell'applicazione, quindi, prima della sua compilazione. Infine, la Sezione 7.2.3 indica ESP, uno strumento altamente automatizzato capace di analizzare il codice e applicare le migliori protezioni al fine di garantire le proprietà di sicurezza richieste dall'utente.

7.2.1 DIABLO

Diablo Is A Better Link-time Optimizer (DIABLO)[5] è un framework in grado di leggere e modificare i file binari di un'applicazione. Questo strumento è idoneo a modificare il binario al fine di ottimizzarne la velocità, il consumo energetico, la dimensione e la sicurezza. Quest'ultima ottimizzazione è ottenuta attraverso tecniche di offuscamento del programma (descritte in Sezione 7.1.1) e di riscrittura degli indirizzi. Questi ultimi, come gli indirizzi assoluti e relativi di istruzioni e di dati, vengono prodotti dal linker[6] durante la costruzione del binario e vengono utilizzati da DIABLO per generare una rappresentazione ad alto livello del programma che non dipende dall'architettura della CPU, per la quale è stato generato l'eseguibile. Le tecniche di offuscamento descritte in Sezione 7.1.1 e supportate da Diablo sono: control Flow flattening, branch functions, opaque predicates, virtualization obfuscation e data obfuscation.

7.2.2 Tigress

Tigress[7] è un offuscatore automatico per applicazioni C che lavora a livello di codice sorgente. Questo è in grado di applicare le trasformazioni richieste dall'utente su un codice sorgente intermedio nominato `CIL`[8]. Il codice C dell'utente viene quindi

[5] https://diablo.elis.ugent.be/node/1

[6] Il linker è un programma che unisce uno o più file oggetto, generati da un compilatore, al fine di originare un file binario eseguibile, una libreria o un altro file oggetto.

[7] https://tigress.wtf/introduction.html

[8] https://github.com/cil-project/cil

processato per essere rappresentato attraverso questo linguaggio e successivamente si applicano le trasformazioni di offuscamento desiderate. Il linguaggio `CIL` è una versione semplificata del linguaggio C dove viene utilizzato un sottoinsieme delle forme sintattiche. Un esempio di semplificazione è l'unica sintassi supportata per i cicli; infatti, in linguaggio `CIL` le modalità di iterazione, come i costrutti `do`, `while` e `for`, vengono trasformate in un costrutto `while(1)` che termina attraverso una chiamata esplicita all'istruzione `break`. Le tecniche di offuscamento supportate da Tigress sono:

- control flow flattening;
- branch functions;
- opaque predicates;
- virtualization obfuscation;
- data obfuscation.

Le tecniche citate sono quindi approfondite in Sezione 7.1.1.

7.2.3 ESP

Expert system for Software Protection (ESP) [1] è un sistema altamente automatizzato in grado di proteggere automaticamente le risorse del codice di un'applicazione scritta in linguaggio C. Questo strumento è capace di modificare il codice sorgente proteggendolo dai possibili attacchi che minacciano le risorse dell'applicazione. Il sui obiettivi sono la definizione e l'attuazione di un insieme di strumenti di protezione, in grado di proteggere il software modificando il codice sorgente e generando il file binario protetto dell'applicazione, senza l'intervento dell'utente. L'utente deve solo specificare le parti sensibili del programma (le funzioni e le variabili da proteggere) e i loro requisiti di sicurezza (come le proprietà di riservatezza e di integrità dei dati).

Le attività eseguite automaticamente dal sistema sono:

1. analizzare il codice sorgente per generare un modello dati che descriva le entità del codice, come le funzioni, le variabili ed il flusso;
2. definire le vulnerabilità delle risorse sensibili del programma al fine di valutare gli attacchi che minacciano le proprietà di sicurezza richieste dall'utente;
3. individuare, tra le tecniche disponibili, le protezioni da applicare per ciascuna risorsa al fine di ritardare gli attacchi rilevati durante l'analisi precedente;
4. generare la migliore combinazione di protezioni da applicare sul codice;

5. aggiungere ulteriori protezioni a parti di codice non sensibili al fine di rallentare l'attaccante nell'identificazione delle risorse protette[9];

6. generare il codice binario protetto dell'applicazione.

Al fine di applicare le protezioni del codice, ESP utilizza internamente gli strumenti Diablo e Tigress descritti rispettivamente in Sezione 7.2.1 e in Sezione 7.2.2.

7.3 Utilizzi pratici di Intel SGX

In questa sezione si presentano alcuni utilizzi pratici di Intel SGX. Questa architettura può essere applicata a diversi ambiti, come il mantenimento e la gestione delle configurazioni delle applicazioni distribuite, descritti in Sezione 7.3.1 o la protezione dei contenitori Docker, descritta in Sezione 7.3.2. Inoltre, Intel SGX può essere utilizzato per proteggere le funzionalità di rete offerte attraverso l'uso di Network Function Virtualization (NFV), un approccio che mira a virtualizzare le funzioni di rete generalmente affidate ai nodi fisici. Infine, La tecnologia SGX può garantire protezione anche alle applicazioni costruite con l'architettura basa sui microservizi, come descritto in Sezione 7.3.4.

7.3.1 Distributed computing configuration management

Apache ZooKeeper[10] è uno strumento centralizzato che fornisce servizi alle applicazioni distribuite. Questo è in grado di gestire le configurazioni dei sistemi distribuiti, come le informazioni sulla gerarchia, la coordinazione e la sincronizzazione dei server. Inoltre, fornisce un servizio di mantenimento delle configurazioni in modalità *key-value*[11]. In particolare, il servizio di ZooKeeper viene implementato da una rete distribuita di server che garantisce uno spazio di nomi gerarchico, simile a quello utilizzato dai sistemi operativi Linux. Infatti, ogni nome è formato da una sequenza di stringhe separate da caratteri `'/'`, ad esempio `/parent/child/child2`. Ogni nome corrisponde univocamente ad una risorsa condivisa chiamata *znode*, la quale può essere un direttorio e contemporaneamente contenere dei dati.

Con la soluzione SecureKeeper [28] si è implementano una versione di ZooKeeper che utilizza Intel SGX con lo scopo di preservare l'integrità e la riservatezza dei dati gestiti. La soluzione utilizza la cifratura per la condivisione delle informazioni

[9] Alcune tecniche di offuscamento del codice possono introdurre un comportamento peculiare facilitando l'attaccante nell'individuare le parti sensibili dell'applicazione. Per questo motivo, ESP include una fase di occultamento delle tecniche utilizzate, applicando le protezioni in sezioni di codice non sensibili. In questo modo, l'attaccante non è facilitato nel riconoscere le aree sensibili protette.

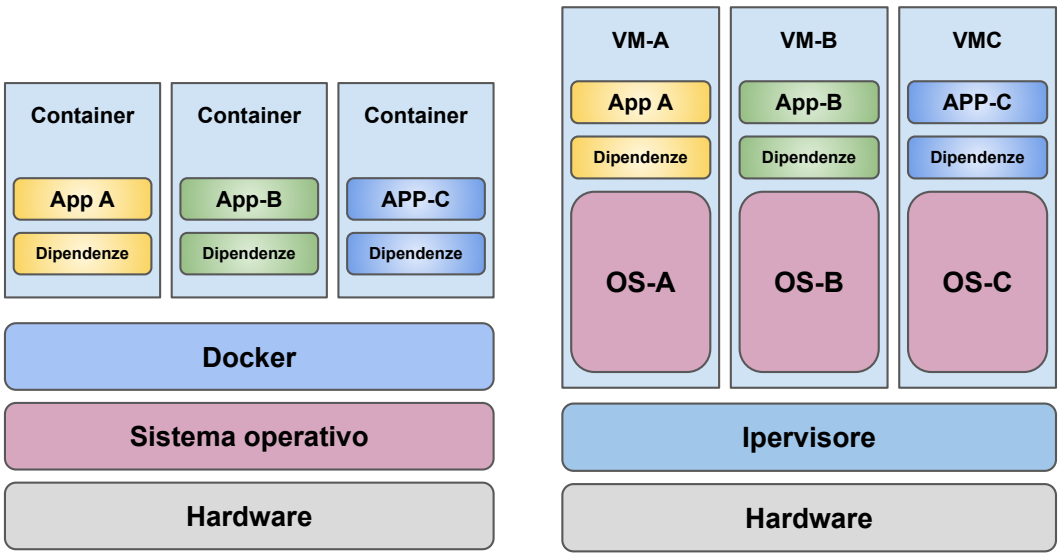
[10] https://zookeeper.apache.org/

[11] La memorizzazione dei dati in modalità *key-value* si basa su tabelle con due sole colonne nominate chiave e valore. Questa architettura permette la memorizzazione di valori eterogenei mantenendo prestazioni e scalabilità elevati con lo svantaggio di poter ricercare i dati per sola chiave.

all'esterno delle regioni protette, mentre utilizza le enclavi per accedere ed elaborare i dati non cifrati nelle fasi in cui lo si richiede. Le prestazioni dell'applicazione sicura sono paragonabili alla versione standard con un decremento medio delle prestazioni pari a 11%.

7.3.2 Secure docker container

Docker[12] è uno strumento di virtualizzazione che consente di inserire le applicazioni all'interno di unità separate chiamate *container*. Un'unità contiene una singola applicazione e tutte le sue dipendenze software in modo tale da poter eseguire e spostare velocemente l'applicazione da un ambiente ad un altro. Il *Docker engine* è poi in grado di eseguire più *container* contemporaneamente. In Figura 7.3 è possibile notare la differenza dell'architettura Docker al confronto delle macchine virtuali. Come evidenziato, i *container* Docker non contengono l'intero sistema operativo come le macchine virtuali. In questo modo l'architettura di Docker risulta vantaggiosa in termine di risorse.



(a) Docker `container`. (b) Macchine virtuali.

Figura 7.3: Architettura Docker al confronto con la virtualizzazione delle macchine.

Il meccanismo `SCONE` [29] utilizza l'esecuzione sicura offerta da Intel SGX per proteggere i processi dei *container* Docker dagli attacchi esterni. Questo meccanismo offre delle librerie di interfaccia che comunicano sia con il *container* Docker che con i driver SGX di Intel. Questa strategia consente di proteggere i contenitori Docker con le proprietà offerte dall'architettura Intel SGX, senza dover modificare il codice dell'applicazione da proteggere. Inoltre, con l'utilizzo dei contenitori `SCONE` le esecuzioni del codice hanno una variazione delle prestazioni che oscilla tra 0,6x e 1,2x, rispetto il container standard. Di conseguenza, l'esecuzione di tipo `SCONE` può anche migliorare le prestazioni oltre ad inserire le protezioni citate.

[12] https://www.docker.com/

7.3.3 Secure Network Function Virtualization

Il Network Function Virtualization (NFV) [30] è una strategia per fornire i servizi di rete. Questi servizi, come firewall[13], Intrusion Detection System (IDS)[14] e load balancer[15], sono funzioni di rete che vengono eseguite su appositi nodi fisici. Con il NFV, questi servizi vengono spostati ed eseguiti in macchine virtuali gestite da server, riducendo quindi i nodi fisici sulla rete. Questo approccio mira a migliorare la capacità e la flessibilità della rete. In particolare, i servizi di rete non vengono più direttamente associati ad un hardware specifico, consentendo un mantenimento delle due parti (software e hardware) più agile ed economico. Inoltre, le funzionalità di rete possono essere gestite da singole o multiple macchine aumentando la scalabilità della rete.

Intel SGX può essere utilizzato per migliorare la sicurezza di queste funzionalità di rete. Infatti, le applicazioni NFV contengono informazioni sensibili, come dati cache dei Web Server o informazioni sugli stati di un IDS. Questi ultimi possono essere protetti attraverso l'uso di Intel SGX con un nuovo schema di protezione chiamato Securing-NFV (S-NFV) [31]. Con quest'approccio, la logica ed i dati sensibili della funzionalità di rete vengono eseguiti all'interno dell'enclavi SGX, mentre nella parte untrusted dell'applicazione si mantengono le funzionalità di gestione dell'enclave (come il caricamento e l'interfaccia alle chiamate sicure) e le funzionalità direttamente connesse all'interfaccia di rete.

7.3.4 Secure cloud microservices

Con l'utilizzo dei `cloud microservices` [32] le applicazioni distribuite seguono un nuovo approccio che consiste nel dividere le singole funzioni base di un'applicazione in microservizi indipendenti. In questo modo, sistemi di grandi dimensioni suddividono la complessità ed il carico di lavoro in microservizi che possono essere gestiti da macchine separate, al fine di migliorarne la scalabilità, l'affidabilità ed il mantenimento.

Tuttavia, per mantenere una elevata scalabilità, questi microservizi sono associati all'esecuzione in ambienti cloud dove non c'è il diretto controllo fisico della macchina. Per questo motivo, non si pone fiducia nelle macchine in cloud e di conseguenza si ricercano strumenti per proteggere i microservizi di un'applicazione.

Intel SGX può essere utilizzato per migliorare la sicurezza del codice eseguito in microservizi [33] basati su Eclipse Vert.x[16], uno strumento utilizzato per la scrittura di applicazioni compatibili con tale approccio. Tuttavia, la tecnologia Vert.x

[13] Un firewall è un dispositivo divisorio tra più reti che fornisce funzioni di sicurezza analizzando (ed eventualmente bloccando) il traffico.

[14] Un IDS è un dispositivo di rete che monitora la rete locale al fine di rilevare anomalie e accessi non autorizzati.

[15] Il load balancer è un componente che gestisce i flussi di rete al fine di distribuire il carico su più server in base alle loro capacità.

[16] https://vertx.io/

utilizza il linguaggio di programmazione Java e di conseguenza non è direttamente compatibile con l'uso dell'enclavi di Intel SGX, dove viene richiesto un codice scritto in linguaggio C o C++. Per questo motivo si utilizza il Java Native Interface (JNI), un framework di programmazione che consente di connettere il codice nativo (in particolare codice C e C++) con il codice Java. Con l'utilizzo di JNI il codice scritto in Java può richiamare il codice C/C++ untrusted che fornisce un collegamento nativo all'enclavi di Intel SGX. In questo modo le applicazioni di grandi dimensioni vengono divise in piccole componenti sicure (protette con l'enclavi SGX) e componenti non protette (implementate in Java), aumentando la fiducia nell'applicazione con un impatto medio sulla perdita di prestazioni limitato circa al 9%.

7.4 Partizionamento automatico delle applicazioni per Intel SGX

Glamdring[34] è un framework sviluppato con lo scopo di proteggere le applicazioni scritte in linguaggio C attraverso l'utilizzo di Intel SGX. L'obiettivo del programma è automatizzare la modifica del codice di un'applicazione al fine di renderla compatibile con Intel SGX. Come discusso in Sezione 2.1.3, il codice dei programmi protetti con SGX deve essere diviso in due parti fondamentali, chiamate trusted ed untrusted. La prima parte è formata dall'insieme di funzioni e dati che necessitano di protezione poiché gestiscono o rappresentano i segreti dell'applicazione. La parte untrusted invece è l'insieme di codice che non necessita di protezione e che quindi non tratta o non interagisce direttamente con i dati sensibili.

Al fine di utilizzare il framework, lo sviluppatore annota il codice dell'applicazione specificando quali dati contengono informazioni sensibili. Poi, viene eseguito Glamdring, il quale analizza il codice sorgente annotato e lo partiziona nelle sezioni trusted ed untrusted, come richiesto da Intel SGX. A questo scopo lo strumento utilizza l'analisi statica del codice per decidere quali regioni siano da proteggere. In particolare viene eseguita l'analisi del flusso dell'applicazione per identificare le funzioni che potrebbero leggere i dati sensibili. Inoltre, lo strumento utilizza il *backward slicing*[17] per determinare le funzioni che accedono alle informazioni da proteggere.

Tutte le funzioni che interagiscono con i dati sensibili dell'applicazione vengono automaticamente spostate nell'enclave, al fine di supportarle con le proprietà di sicurezza offerte da Intel SGX. Inoltre, Glamdring aggiunge del nuovo codice all'applicazione, implementando operazioni crittografiche agli estremi dell'enclave

[17]Lo slicing è una tecnica di semplificazione del sorgente che rimuove parti di codice. In particolare il backward slicing mantiene solo il codice che interagisce con un insieme di variabili. Dopo l'applicazione della tecnica, il codice è eseguibile e preserva il risultato del programma originale sui dati scelti.

e controlli eseguiti a runtime al fine di proteggere l'applicazione da attacchi *Enclave call ordering*[18], *Iago*[19] e *Replay*[20]. Le modifiche svolte dal framework unite all'esecuzione con Intel SGX, garantiscono la confidenzialità e l'integrità dei dati dell'applicazione, anche nel caso in cui l'attaccante abbia il diretto e completo controllo della macchina.

Le prestazioni delle applicazioni modificate da Glamdring sono state confrontate con l'esecuzione nativa e con l'esecuzione attraverso il meccanismo SCONE, descritto in Sezione 7.3.2. Lo studio delle performance ne suggerisce un deterioramento rispetto alle due alternative citate. Tuttavia, la diminuzione delle prestazioni dei programmi protetti con Intel SGX è molto influenzata dalla frequenza con cui avviene il cambio di contesto tra parte trusted ed untrusted. Con particolare riferimento all'esecuzione di Memcached[21], le prestazioni native raggiungono una velocità di risposta alle richieste di circa 3.5 volte più veloce rispetto all'eseguibile modificato da Glamdring. Invece, il meccanismo SCONE confrontato con l'esecuzione protetta da Glamdring risulta più veloce di circa 1.8 volte. La maggiore velocità di SCONE è dovuta ad una minore frequenza di cambio di contesto tra l'ambiente trusted ed untrusted, con lo svantaggio di una maggiore allocazione della memoria protetta[34].

La soluzione Glamdring può essere confrontata con il framework sviluppato con questo lavoro di tesi. Infatti, lo scopo dei due programmi è automatizzare le modifiche necessarie al codice sorgente al fine di poterlo proteggere con la tecnologia Intel SGX. Inoltre, entrambi i framework sono abili nel generare il file sorgente e il file di definizione dell'enclave, facendosi carico di implementare il codice per gestire gli errori comuni delle chiamate ECall ed OCall, e di gestire l'allocazione dell'enclave. Tuttavia, per effettuare le operazioni della conversione, è necessario effettuare delle analisi su codice sorgente. L'analisi statica del codice viene quindi effettuata da entrambi i framework che la utilizzano per individuare le parti di codice da proteggere. La principale differenza tra i due programmi è l'approccio con cui si identificano le regioni da proteggere, in particolare è possibile definire due diverse strategie: data-driven e function-driven. La prima strategia, utilizzata da Glamdring, individua le funzioni da spostare nell'enclave attraverso la loro dipendenza dai dati sensibili. Infatti, lo sviluppatore annota le variabili da proteggere ed il framework identificherà come funzioni da spostare nell'enclave tutte quelle che gestiscono tali dati.

Con la strategia function-driven, quella utilizzata dal framework oggetto di questa tesi, lo sviluppatore annota direttamente le funzioni da proteggere, dando così più controllo al programmatore sulle funzioni effettivamente spostate nell'enclave.

[18]Con l'attacco *Enclave call ordering* l'attaccante cambia l'ordine con cui vengono eseguite le chiamate ECall al fine di alterare i dati presenti all'interno dell'enclave.

[19]L'attacco *Iago* consiste nel modificare il valore di ritorno delle *system call* effettuate dalle funzioni protette al fine di alterarne il comportamento.

[20]L'attaccante che svolge un attacco di tipo *replay* intercetta e scambia i dati forniti alle chiamate protette con vecchi messaggi inviati precedentemente.

[21]Memcached è un programma utilizzato per migliorare la velocità dei sistemi distribuiti che memorizza i risultati delle interrogazioni ai database e riduce così il numero di richieste ai server. https://memcached.org/

L'analisi effettuata dal framework studia il flusso di esecuzione dell'applicazione e la dipendenza tra le funzioni, individuando quali altre regioni di codice siano necessarie alle funzioni da proteggere in modo da spostarle a loro volta nell'enclave. Questo approccio consente di applicare la protezione alle funzioni strettamente necessarie alla computazione sicura, anche quando i dati in input non possono essere protetti per loro natura dai processi privilegiati (e.g. input da tastiera, da webcam o da microfono).

Un'altra differenza tra i due programmi è la dipendenza dagli strumenti esterni utilizzati per effettuare l'analisi del codice. Il framework di questa tesi utilizza solo software libero come gli analizzatori di codice come Ctags[22] e Frama-C[23], mentre Glamdring utilizza il software proprietario CodeSurfer[24].

[22] https://ctags.io/

[23] https://frama-c.com/download_boron.html

[24] https://www.grammatech.com/codesurfer-binaries

Capitolo 8

Conclusioni

In questo lavoro di tesi, si è descritta una nuova soluzione di protezione automatizzata del codice, relativa alla tecnologia Intel SGX. Come descritto nel Capitolo 2, questa tecnologia definisce un altro tipo di esecuzione protetta del codice, basata su nuove istruzioni macchina dei processori Intel. L'esecuzione protetta è consentita attraverso l'uso di regioni di memoria chiamate enclavi, le quali garantiscono le proprietà di integrità e di riservatezza ai dati in esse contenuti. Quindi, il codice e le variabili memorizzati nelle enclavi non possono essere letti né modificati dagli altri processi, indipendentemente dal loro livello di privilegio.

Tuttavia, l'utilizzo della tecnologia SGX comporta alcune limitazioni ed adempimenti. Il limite più evidente è la necessità di eseguire le applicazioni protette con i processori Intel compatibili, di conseguenza le applicazioni progettate e compilate per questa tecnologia potranno essere eseguite solo con macchine compatibili. Poi, il codice protetto nell'enclavi deve essere scritto nei linguaggi C e C++. Inoltre, il codice protetto non può importare librerie esterne (come le librerie standard dei linguaggi C/C++) ad eccezione delle librerie standard ridefinite dalla tecnologia SGX. Infine, il codice dell'applicazione deve essere diviso in due parti chiamate trusted ed untrusted. La prima consiste nelle sezioni di codice e nei dati protetti nelle enclavi, mentre la seconda è la parte non sensibile dell'applicazione che non necessita della protezione SGX. Questa divisione è fondamentale nell'architettura ed un'errata ripartizione del codice può annullare le proprietà di sicurezza offerte e danneggiare le prestazioni.

L'adempimento più importante ed oneroso è la modifica dei file sorgente dell'applicazione. Infatti, per poter garantire le proprietà di sicurezza citate, il codice ed i dati sensibili devono essere spostati nel codice dell'enclave. Per definizione, il codice dell'enclave (la parte trusted dell'applicazione) non è accessibile dall'esterno (la parte untrusted). Le due parti dell'applicazione possono comunicare solo attraverso delle funzioni di interfaccia chiamate ECall ed OCall. Queste ultime devono essere definite nel codice protetto e devono essere inserite nel file di dichiarazione dell'enclave. Quindi, un secondo adempimento oneroso è l'implementazione (nel sorgente), la dichiarazione (nel file di definizione dell'enclave) e la gestione degli errori delle chiamate di interfaccia, ECall ed OCall. Infine, un ultimo adempimento è l'allocazione e la gestione degli errori dell'enclave. Infatti, il codice untrusted deve prima generare l'enclave per poi potersi interfacciare con essa. Inoltre, il codice non

protetto deve gestire eventuali errori di allocazione o di manomissione dell'enclave, per esempio avvisando l'utente e terminando il programma.

Come discusso nel Capitolo 5, queste modifiche da apportare all'applicazione sono molto complesse ed onerose, soprattutto al crescere della dimensione dell'applicazione. Per questo motivo si è sviluppato un framework di conversione del codice che permette di automatizzare queste operazioni di protezione. Lo sviluppatore annota il codice definendo quali dati e funzioni siano connesse alle risorse sensibili, specifica al framework i file sorgente da convertire ed avvia la conversione. Il framework analizza i file, ne comprende le entità (e.g. funzioni, strutture dati, definizioni, unioni, enumerazioni, librerie importate) e genera un modello di dati in grado di rappresentare il codice. Poi, il framework avvia un'analisi del flusso del codice, individuando le dipendenze tra le funzioni ed utilizza le annotazioni dell'utente per individuare la parte di codice da inserire all'interno dell'enclave. Genera quindi le funzioni destinazione delle chiamate ECall ed OCall, comprendendo la gestione degli errori e gestendo la comunicazione dei dati cifrati tra le chiamate. Successivamente, il framework elabora il file di definizione dell'enclave scritto in formato EDL, dove inserire le chiamate ECall/OCall, le strutture dati (comprese enumerazioni ed unioni), le definizioni utente e l'importazione delle librerie necessarie. Infine, lo strumento gestisce intelligentemente l'allocazione dell'enclave, generando la struttura dati protetta seguendo le necessità dell'applicazione. In sintesi, il framework analizza il codice annotato dallo sviluppatore e genera un pacchetto pronto alla compilazione con Intel SGX.

Successivamente, si sono analizzate le prestazioni sia durante la conversione dell'applicazione che durante l'esecuzione del programma protetto. Come descrive il Capitolo 6, si è proceduto a convertire pacchetti Debian e codici sorgenti di applicazioni di dimensione e complessità crescente generati appositamente. Si è quindi analizzato il tempo e le risorse necessarie al framework per convertire i programmi e si è approfondito l'impatto prestazionale dell'uso della tecnologia SGX sulle applicazioni convertite.

Infine, nel Capitolo 7 ci si è soffermati sulle altre tecnologie disponibili per l'applicazione automatica delle tecniche di protezione del codice sorgente. Nel capitolo si descrivono quindi le tecniche utilizzate per contrastare il reverse engineering, la manomissione del codice ed il processo di debug. Poi, si definiscono alcuni strumenti di attuazione delle tecniche come Diablo, Tigress ed ESP utilizzati per le protezioni inserite nel codice binario e nel codice sorgente. In ultimo, il Capitolo 7 descrive gli utilizzi pratici di Intel SGX, valutando la tecnologia applicata a diversi campi, come la protezione dei contenitori Docker e delle architetture a microservizi.

Il framework oggetto di questa tesi, ha come obbiettivo l'automazione della conversione di un'applicazione per l'uso della tecnologia SGX. Tuttavia, lo strumento possiede ancora alcune limitazioni, come la definizione di una singola enclave per il programma, la compatibilità con il solo tipo interno per le chiamate sicure e la gestione delle variabili globali sensibili. Come valore lavoro, si prevede un miglioramento del framework tramite l'aggiunta di opzioni che permettano di dividere i dati e le funzioni sicure in diversi enclavi, separandoli per contesti diversi. Inoltre, si presume la compatibilità con diversi tipi di dato per il valore di ritorno delle

funzioni sicure, prevedendo una gestione degli errori adattiva al tipo di dato utilizzato. Infine, si considera una più avanzata implementazione di analisi e modifica del codice che permetta di amministrare l'uso delle variabili globali nelle sezioni di codice protetto, generando del codice di gestione che ne permetta un accesso controllato.

Appendice A

Manuale utente

In questa appendice si descrivono le modalità di funzionamento del framework con particolare attenzione al flusso di lavoro da svolgere per configurarlo. In dettaglio, in Appendice A.1 si definiscono gli utilizzi del framework a partire dalla sua installazione fino al suo utilizzo pratico. Vengono descritte quindi le configurazioni, le annotazioni e le modalità di esecuzione del framework. Inoltre, vengono definiti i codici di errore generati dal framework in caso di errori o di limitazioni non rispettate dal codice sorgente.

Il framework è in grado di analizzare un pacchetto di codice sorgente scritto in linguaggio C e convertirlo automaticamente in un codice trusted ed untrusted, attraverso l'utilizzo della tecnologia Intel SGX. Il codice trusted viene eseguito in contenitori virtuali chiamati enclavi, i quali consentono la protezione del codice con le proprietà di integrità e riservatezza. L'utente annota il codice sorgente originale e definisce quali funzioni devono essere eseguite nelle regioni protette. Poi, viene eseguito il framework che modifica il codice originale ed aggiunge i file necessari alla creazione dell'enclave. Terminata la conversione il codice viene compilato seguendo le informazioni riportate nelle successive sezioni.

Infatti, in Appendice A.2 si definisce come abilitare Intel SGX sulla macchina e si descrive l'esecuzione della compilazione del codice convertito, in Appendice A.3 per sistemi operativi Linux ed in Appendice A.4 per sistemi operativi Windows.

A.1 Utilizzo del framework

In questa appendice si definiscono nel dettaglio le modalità di utilizzo del framework. In particolare, in Appendice A.1.1 si descrivono le operazioni e le dipendenze software necessarie all'installazione del framework di conversione sulla macchina. Poi, in Appendice A.1.2 si definiscono le modifiche preliminari del codice necessarie al framework per poter attuare la conversione del sorgente. In questa fase l'utente annota il codice per definire le funzioni destinazione delle chiamate ECall ed OCall, aggiunge le direttive di copia e spostamento per abilitare l'uso delle costanti globali nelle funzioni trusted, e modifica il codice per rispettare i limiti del framework.

In Appendice A.1.3 si definisce in modo pratico come configurarle ed eseguire il framework sul pacchetto di codice. In Appendice A.1.5 si descrivono i codici di errore generati dal framework in caso di errori o limiti non rispettati e si fornisce una possibile modalità per la loro gestione. Infine, vengono descritti i tipi dei dati ammessi nelle chiamate ECall ed OCall in Appendice A.1.6.

A.1.1 Installazione del framework

Al fine di utilizzare il framework è necessario installare una versione di Python compatibile. Ad oggi il framework utilizza la versione di Python 3.8.1[1]. L'installazione dipende dal sistema operativo in uso e non è quindi approfondita. Una volta eseguita l'installazione dell'interprete, è necessario installare i moduli Python utilizzati dal framework, con i comandi citati:

- `pip install pydot`
- `pip install regex`
- `pip install tqdm`
- `pip install logging3`

Poi, il framework necessita degli strumenti esterni Frama-C[2], Ctags[3] e GCC[4]. Questi strumenti devono poter essere raggiungibili dal framework senza che questo ne conosca il percorso completo di installazione. Si necessita quindi un aggiornamento della variabile d'ambiente `$PATH` con l'aggiunta dei tre percorsi assoluti. Infine, il codice del framework viene fornito all'interno del file zip "`framework.zip`" insieme al file di configurazione "`frameworkConfig.py`", descritto in Sezione A.1.3. Quest'ultimo è configurato per convertire un codice sorgente di esempio incluso con il sorgente del framework. Al fine di poter utilizzare il framework è necessario estrarre il contenuto della cartella compressa e salvarlo su disco.

A.1.2 Preparazione del codice da convertire

Prima di poter convertire il codice sorgente per proteggerlo con Intel SGX, è necessario riconoscere le parti sensibili del programma che necessitano della protezione ed individuare quali funzioni debbano essere inserite nell'enclave. Allo stesso modo, occorre individuare le funzioni untrusted chiamate dall'interno dell'enclave. Entrambi questi tipi di funzioni, trusted ed untrusted, devono essere annotate con le macro di definizione delle chiamate ECall ed OCall, mostrate in Figura A.1. Le annotazioni macro scritte nel file sorgente hanno sintassi `#define sgx_[ecall|ocall]_nomeFunzione` e comunicano al framework le funzioni che devono essere definite come chiamate ECall ed OCall. Inoltre, la notazione della macro

[1] https://www.python.org/downloads/release/python-381/

[2] https://frama-c.com/download_boron.html

[3] https://ctags.io/

[4] https://www.cygwin.com/

è seguita da una lista di argomenti, inseriti tra una coppia di parentesi tonde, che informano lo strumento sul tipo di copia degli argomenti desiderata dall'utente:

- `i`, copia dell'argomento in ingresso alla chiamata;
- `o`, copia dell'argomento in uscita alla chiamata;
- `b`, copia dell'argomento in ingresso ed in uscita alla chiamata;
- `u`: copia non effettuata.

```
#define sgx_ecall_trusted ([myBuf, u])
int trusted(void *myBuf, int len){
    ...
}

#define sgx_ocall_untrusted ([text, i, text_size])
int untrusted(char* text, int text_size){
    ...
}
```

Figura A.1: Esempio di definizioni di macro per chiamate ECall ed OCall.

Una volta annotato il codice con le definizioni di chiamate ECall ed OCall, è necessario individuare le parti trusted di codice che utilizzano costanti e variabili globali. Il framework di conversione non è ancora in grado di analizzare la dipendenza software delle funzioni alle costanti globali. Per questo motivo, in caso di necessità, l'utente può utilizzare le direttive di copia e spostamento di parti di codice sorgente. In Figura A.2a è rappresentata una definizione d'esempio per la direttiva di spostamento, mentre in Figura A.2b è definita una direttiva di copia del codice.

```
#pragma move_start
static const char
       *const digit[] = {
       ----.",
       --..",
};
#pragma move_end
```

(a) Direttiva di spostamento.

```
#pragma copy_start
static const char
       *const digit[] = {
       "----.",
       "--..",
};
#pragma copy_end
```

(b) Direttiva di copia.

Figura A.2: Esempio di definizione delle direttive `move` e `copy`.

Nel dettaglio le due direttive vengono definite attraverso dei comandi di precompilazione `#pragma` e non vengono utilizzate le direttive di definizione `#define` poiché l'utilizzo di più direttive di copia e spostamento nello stesso file genererebbe degli errori di ridefinizione.

La direttiva di copia è composta da due definizioni `#pragma` inserite agli estremi di una sezione di codice. In questo modo il framework seleziona la parte di codice contenuta e la copia all'interno del codice dell'enclave. Allo stesso modo, la direttiva di spostamento definisce altre due parole chiave per definire una regione di codice da spostare all'interno della regione dell'enclave. In questo modo, le costanti globali possono essere copiate o spostate anche all'interno del codice dell'enclave.

A.1.3 Esecuzione del framework

L'utilizzo del framework può avere luogo solo dopo l'installazione di tutte le dipendenze software necessarie, spiegate in Appendice A.1.1, ovvero:

- interprete Python 3.8.1 o successivo;
- moduli Python `pydot`, `regex`, `tqdm` e `logging3`;
- strumenti esterni Frama-C, Ctags e GCC.

Poi, l'utente deve provvedere alla modifica del programma in modo da rispettare i limiti del framework. Successivamente, l'utente individua le funzioni che devono essere protette nell'enclave e che ne devono essere i punti di uscita. Poi, vengono annotate queste funzioni direttamente nei file sorgente del programma attraverso le macro del framework, come definito in Appendice A.1.2.

```
files = ["file1.c", "file1.h", "file2.c", "file2.h"]
rootFolder = "source-code"
```

Figura A.3: Configurazione del framework per l'indicazione dei file.

Eseguite queste operazioni, l'utente configura il framework impostando la lista di file e la cartella radice del codice sorgente. Questo è possibile farlo attraverso i due valori di configurazione presenti nel file "`frameworkConfig.py`" della cartella principale del progetto, come mostrato in Figura A.3. La lista `files` contiene i percorsi relativi dei singoli file mentre la stringa `rootFolder` contiene il percorso alla cartella radice del codice sorgente.

A questo punto, è possibile eseguire il file "`framework.py`" con doppio click del mouse o attraverso l'uso dell'interprete Python invocato sulla console con il comando `python framework.py`. Il framework inizia il flusso di esecuzione, dove viene analizzato e convertito il codice. Quindi, termina generando nel percorso di lavoro

una cartella nominata "`generated_trusted`". Al suo interno vengono inserite due cartelle ed il file di log. All'interno della prima cartella viene memorizzato il codice sorgente modificato, mentre la seconda cartella contiene il codice dell'enclave. In Figura A.4 è riassunta la struttura del codice protetto in uscita dal framework. Inoltre, nel file di log sono contenute le informazioni relative ai dati analizzati e alle operazioni effettuate durante la conversione del codice.

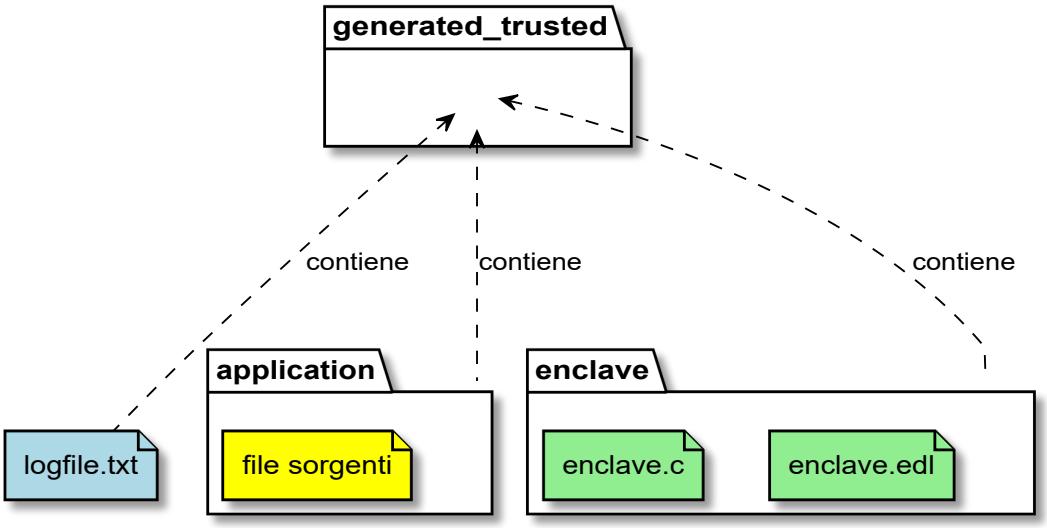


Figura A.4: Struttura del codice contenuto nelle cartelle in uscita.

Dopo la conversione è possibile compilare il codice in base al sistema operativo in uso, come spiegato in Appendice A.4 per la compilazione su sistemi operativi Windows e come descritto in Appendice A.3 per i sistemi operativi Linux.

A.1.4 Esempio di configurazione del Framework

In questa appendice si presenta un esempio concreto di utilizzo del framework. Al fine si effettuare le operazioni svolte in questa appendice è necessario eseguire prima l'installazione delle dipendenze del framework, descritte in Appendice A.1.1.

Il codice del framework è fornito nel file zip "`framework.zip`" insieme ad un'applicazione d'esempio e la configurazione necessaria alla sua conversione. Dopo aver scaricato il file compresso è necessario decomprimere il suo contenuto e salvarlo su disco. A questo punto è possibile convertire l'applicazione `banner` contenuta nell'omonima cartella. L'applicazione è formata da tre file di nome: "`banner.c`", "`XGetopt.h`" e "`XGetopt.c`".

Al fine di convertire l'applicazione è necessario configurare il framework per selezionarne i file sorgente. Aprire quindi il file di configurazione del framework "`frameworkConfig.py`" e modificarne il contenuto in modo da selezionare la cartella radice dei sorgenti e i singoli nomi dei file, come visibile in Figura A.5.

```
files = ["banner.c", "XGetopt.h", "XGetopt.h"]
rootFolder = "banner"
```

Figura A.5: Esempio di configurazione del framework.

Successivamente è necessario definire le funzioni che necessitano delle protezioni offerte da Intel SGX (ECall) e le funzioni che saranno l'output dell'enclave (OCall). Aprire quindi i file sorgente dell'applicazione ed individuare le funzioni sensibili. Nel caso d'esempio si è scelta la funzione `compute_message` come ECall e la funzione `putchar_ocall` come chiamata OCall. Modificare quindi il file "`banner.c`" per annotare le due funzioni, come visibile in Figura A.6. Nella notazione della chiamata ECall `compute_message` è necessario inserire la notazione `[message, i]` per definire il metodo di copia dell'argomento puntatore, in particolare viene imposta la copia in input per l'argomento `message`.

```
#define sgx_ocall_putchar_ocall ()
int putchar_ocall(char c){
        ...

#define sgx_ecall_compute_message ([message, i])
int compute_message(char message[MAXMSG], int nchars, int width){
        ...
```

Figura A.6: Esempio di annotazione per le chiamate ECall ed OCall.

Poiché la funzione `compute_message` utilizza le costanti `asc_ptr` e `data_table` e queste ultime non sono utilizzate da altre funzioni, è possibile inserire l'annotazione di movimento per spostare i dati da proteggere all'interno dell'enclave. Modificare quindi il file "`banner.c`" per annotare i due valori costanti, come visibile in Figura A.7.

```
#pragma move_start
        const int asc_ptr[NCHARS] = {
                ...
        const char data_table[NBYTES] = {
                ...
#pragma move_end
```

Figura A.7: Esempio di annotazione per la macro di movimento.

Completati questi passaggi è possibile eseguire il framework per effettuare la conversione. Aprire un terminale nella cartella radice del framework, ovvero dove è possibile trovare il file "`frameworkConfig.py`" ed eseguire il comando `python framework.py`. Sul terminale verrà mostrata una barra di progresso che indica lo stato della conversione. Terminato il processo, è possibile individuare una nuova cartella nominata "`generated_trusted`", dove si trovano due ulteriori directory "`application`" ed "`enclave`" oltre al file di log "`logfile.txt`" da consultare quando la conversione non termina correttamente. Le due cartelle contengono rispettivamente il codice sorgente modificato dell'applicazione e il codice dell'enclave unito al suo file di definizione. Il codice così convertito può essere compilato ed eseguito con Intel SGX seguendo l'Appendice A.3 per la compilazione su sistemi Linux e l'Appendice A.4 per la compilazione su sistemi Windows .

A.1.5 Codici di errori

In questa appendice si definiscono gli errori generati dal framework in caso di conflitti con l'architettura SGX e di problemi ricorrenti. Si fornisce quindi, una lista di codici e di spiegazioni in modo che l'utente possa gestire i problemi del proprio codice.

#A001 Trovate più definizioni di macro per la stessa funzione: rimuovere le macro in eccesso per la funzione riportata.

#A002 Trovata una definizione macro per una funzione non esistente: controllare che il nome della funzione inserito nella macro sia corretto o definire la funzione.

#A003 Funzione <function> non trovata nel codice: definirla nei file sorgente.

#A004 Struttura <struct> non trovata nel codice: definirla nei file sorgente.

#A005 Enumerazione <enum> non trovata nel codice: definirla nei file sorgente.

#A006 Unione <union> non trovata nel codice: definirla nei file sorgente.

#B001 Non ci annotazioni macro ECall nel codice: definire almeno una chiamata ECall da proteggere.

#B002 Il codice sorgente è di sintassi non compatibile: cambiare la sintassi del codice nello standard ANSI C.

#B003 Il codice untrusted e il codice trusted usano la stessa funzione <function>: definire la funzione come ECall o OCall.

#B004 La struttura dati di tipo <type> e nome <name> non è trovata nei file ma viene utilizzata per le chiamate ECall ed OCall. Utilizzare solo i tipi utente o tipi semplici, descritti in Appendice A.1.6.

#B005 Una chiamata OCall ed una chiamata ECall hanno lo stesso nome <name>: cambiare il nome o definire la funzione singolarmente ECall o OCall.

#B006 Il file <file> non esiste: controllare il nome del file ed il percorso specificato.

#C001 Lo strumento CTAGS non è disponibile: installare CTAGS, come descritto in Appendice A.1.1.

#C002 Trovata ridefinizione di nome <name> nei file <file>: il nome delle strutture dati, enumerazioni, unioni e funzioni deve essere univoco. Cambiare il nome nel codice.

#D001 Lo strumento Frama-C non è disponibile: installare Frama-C, come descritto in Appendice A.1.1.

#D002 La funzione <function> non esiste nel file <file>. definire la funzione nel file.

#D003 Frama-C ha trovato errori nel file <file>: correggere gli errori del codice sorgente nel file.

#E001 La funzione ECall <function> è già presente nel file EDL: contattare lo sviluppatore.

#E002 La funzione OCall <function> è già presente nel file EDL: contattare lo sviluppatore.

#F001 Funzione <function>, ECall o OCall, definita `static`: rimuovere l'attributo `static`.

#F002 Il valore di ritorno della funzione ECall o OCall non è di tipo `int`: cambiare la funzione in modo che abbia valore di ritorno intero.

#F003 Le chiamate ECall e OCall non possono utilizzare i vettori dinamici: utilizzare i puntatori (* invece che []), come spiegato in Appendice A.1.6.

#F004 Le chiamate ECall e OCall possono avere come attributi costanti solo puntatori: rimuovere l'attributo `const`, come descritto in Appendice A.1.6.

#F005 Mancano degli attributi nella macro <macro>: definire il tipo di copia e la grandezza del valore riferito per tutti gli argomenti puntatore, come spiegato in Appendice A.1.2.

#F006 L'argomento <arg> definito nella macro <macro> non è coerente con la definizione della funzione: correggere l'argomento seguendo le guide definite in Appendice A.1.2.

#F007 L'argomento <arg>, utilizzato come dimensione da copiare, non esiste nella funzione <function>: definire l'argomento o controllare il nome.

#F008 L'argomento <arg>, utilizzato come dimensione da copiare, deve essere di tipo intero: cambiare il tipo dell'argomento.

#F009 L'argomento <arg>, utilizzato come dimensione da copiare non può essere un puntatore: cambiare il tipo ad intero semplice.

#F010 L'argomento <arg> non esiste nella funzione <function>: controllare il nome o definire l'argomento.

#F011 La dimensione di copia va specificata solo per gli argomenti da copiare: utilizzare un metodo di copia o rimuovere la dimensione per l'argomento <arg> in funzione <function>.

#F012 La dimensione di copia deve essere specificata per l'argomento <arg> da copiare: definire la dimensione o utilizzare l'opzione `u` descritta in Appendice A.1.2.

#F013 I puntatori multipli possono essere utilizzati solo l'opzione `u` descritta in Appendice A.1.2.

#F014 La dimensione <size> non deve essere specificata per il parametro <arg> nella funzione <function>: rimuovere la dimensione specificata.

#F015 L'opzione <option> non esiste tra quelle disponibili: utilizzare le opzioni valide scritte in Appendice A.1.2.

#F016 Il tipo <type> in funzione <function> non è supportato dal framework: utilizzare i tipi supportati descritti in Appendice A.1.6.

#G001 Lo strumento GCC non è disponibile: installare GCC, come descritto in Appendice A.1.1.

#G002 La configurazione delle librerie SGX è corrotta: controllare la configurazione in <file> ed eseguire nuovamente il framework.

#G003 Il file <file> contiene errori: correggere gli errori di sintassi.

#G004 La funzione <function> non è definita nelle librerie SGX: definire una nuova funzione OCall per poterla utilizzare dall'interno dell'enclave.

#G005 Il file di configurazione <file> non esiste: eseguire nuovamente il framework per generarlo.

A.1.6 Tipi ammessi negli argomenti delle chiamate ECall ed OCall

Gli argomenti delle chiamate ECall ed OCall possono essere solo dei seguenti tipi:

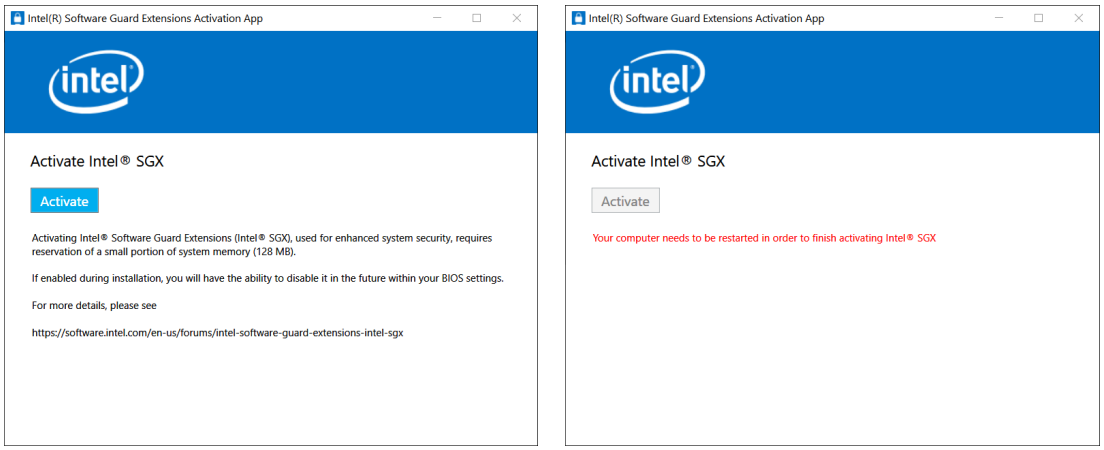
- `[const] struct nome_tipo_struct var`
- `[const] enum nome_tipo_enum var`
- `[const] union nome_tipo_union var`
- `[const] void *var`
- `[const] [unsigned] int var`
- `[const] [unsigned] short var`
- `[const] [unsigned] char var`
- `[const] [unsigned] float var`
- `[const] [unsigned] double var`

Inoltre, ogni argomento può essere di tipo puntatore e/o vettore statico. I vettori a dimensione dinamica non sono supportati dalla sintassi EDL [12], mentre la direttiva `const` è valida solo per i puntatori.

Infine, le direttive di ridefinizione di nome `typedef` non sono supportate dal framework, di conseguenza è necessario inserire il nome completo delle strutture dati.

A.2 Abilitare Intel SGX

La tecnologia Intel SGX è supportata dalle CPU Intel di sesta generazione di Intel Core Skylake. Al fine di abilitare il supporto a Intel SGX, è necessario avviare la macchina ed accedere alle configurazioni del BIOS. Poi, attraverso le impostazioni avanzate di sicurezza è possibile abilitare il supporto a SGX. Nel caso il proprio BIOS non supporti tali opzioni, non è possibile eseguire sulla macchina applicazioni sviluppate e compilate per l'esecuzione sicura in enclave. Tuttavia, per i sistemi Windows 10 esiste un ulteriore metodo per abilitare il supporto all'architettura.



(a) Prima. (b) Dopo.

Figura A.8: Attivazione di Intel SGX attraverso Intel(R) Software Guard Extensions Activation App.

Attraverso l'installazione e l'utilizzo dell'applicazione Intel(R) Software Guard Extensions Activation App[5] è possibile abilitare il supporto ad SGX. In caso di utilizzo dell'applicazione, dopo l'attivazione del supporto, sarà necessario riavviare il sistema operativo. In Figura A.8 è rappresentata l'applicazione prima e dopo l'attivazione di Intel SGX

A.3 Compilazione su sistemi operativi Linux

Intel SGX SDK è compatibile con le seguenti distribuzioni Linux[6]:

- Ubuntu 16.04/18.04 LTS 64-bit[7];
- Red Hat Enterprise Linux Server release 7.4/8.0 64 bit[8];
- CentOS 7.5 64 bit[9];
- Fedora 27 Server 64 bit[10];
- SUSE Linux Enterprise Server 12 64 bit[11].

[5] https://www.microsoft.com/store/productId/9N4TV0SXLNMT

[6] https://download.01.org/intel-sgx/sgx-linux/2.9.1/docs/Intel_SGX_Installation_Guide_Linux_2.9.1_Open_Source.pdf#page=6

[7] https://www.ubuntu-it.org/download

[8] https://www.redhat.com/it/resources/red-hat-enterprise-linux-server

[9] https://www.centos.org/download/

[10] https://getfedora.org/it/

[11] https://www.suse.com/it-it/products/server/

I comandi da eseguire per installare Intel SGX SDK dipendono dalla distribuzione Linux utilizzata. Si rimanda quindi alla guida[12] della distribuzione in uso per ulteriori informazioni a riguardo.

La compilazione per Intel SGX SDK è effettuata, su sistemi operativi Linux, con lo strumento GNU make[13]. Questo strumento utilizza 3 diversi file per compilare i sorgenti. Infatti, il primo viene utilizzato per compilare il codice untrusted, il secondo per compilare l'enclave ed il terzo per le configurazioni comuni.

Quest'ultimo contiene le configurazioni che devono essere impostate dall'utente per poter eseguire la compilazione. Di queste, le più importanti sono:

1. il percorso di installazione di Intel SGX SDK;
2. il percorso di output;
3. il tipo di modalità di compilazione (debug, pre-release, release, simulation).

Inoltre, l'utente può modificare il file di configurazione dell'enclave dove sono specificati gli attributi come la dimensione della regione Stack e della regione Heap. Infine, l'utente deve fornire la coppia di chiavi RSA per firmare l'enclave. Per la modalità di debug, è possibile utilizzare la chiave privata RSA a 3072 bit, fornita con la compilazione di esempio del framework, per firmare l'enclave. Il framework viene fornito con i file di configurazione e compilazione predefiniti utili alla compilazione dei sorgenti generati da framework, sui sistemi Linux.

Durante la compilazione della libreria dell'enclave sono necessari due importanti strumenti: `sgx_edger8r`, `sgx_sign`. Il primo serve per creare le librerie trusted e untrusted (a partire dal file EDL dell'enclave generato dal framework) utilizzate dal codice dell'applicazione per connettere la parte protetta e non protetta. Il secondo invece, firma il file della libreria dinamica con la chiave privata. Questi due strumenti sono presenti dentro alle cartelle degli SDK ed il loro percorso è specificato nel file di configurazione di GNU make fornito con il framework.

L'utente quindi, configura i file di compilazione del progetto untrusted dove è necessario inserire il percorso ed il nome di tutti i file. Vengono quindi inseriti i comandi per generare i file oggetto per ogni sorgente e in ultimo il file binario eseguibile dell'applicazione.

Una volta completate queste operazioni, l'utente esegue il comando GNU make con percorso alla cartella principale del progetto. L'interprete eseguirà i file GNU make presenti in tutte le sottocartelle (compresa quella relativa all'enclave) e genererà il file eseguibile e la libreria dell'enclave nella cartella di output. I file modificati dal framework vengono compilati allo stesso modo dei progetti utente protetti con SGX. Si rimanda quindi alla guida completa per la compilazione su sistemi Linux[12].

[12] https://download.01.org/intel-sgx/sgx-linux/2.9.1/docs/Intel_SGX_Installation_Guide_Linux_2.9.1_Open_Source.pdf#page=7

[13] https://www.gnu.org/software/make/

A.4 Compilazione su sistemi operativi Windows

Prima di procedere alla compilazione delle applicazioni sviluppate per Intel SGX, come quelle convertite con il framework, è necessario installare una versione di Microsoft Visual Studio 2015 o superiore[14], abilitare Intel SGX sul sistema (come descritto in Appendice A.2) ed installare Intel SGX SDK [15]. L'ordine di queste operazioni è fondamentale e non può essere variato. Infatti, l'installazione di Intel SGX SDK configura le impostazioni di Visual Studio aggiungendo un plugin che consente di compilare il codice con SGX. Nel caso di un'installazione fuori ordine è necessario disinstallare Intel SGX SDK e reinstallarlo nuovamente[16]. Per la compilazione d'esempio si utilizza Visual Studio Professional 2017 (il nome di alcuni tasti potrebbe variare a seconda della versione utilizzata). Dopo la conversione eseguita dal framework, spiegata in Appendice A.1.3, il framework genera una cartella contenente il codice sorgente modificato. Al suo interno vengono inserite due cartelle, "enclave" e "application", contenenti rispettivamente il codice dell'enclave ed i file sorgenti modificati. Queste ultime devono essere importate in Visual Studio attraverso l'uso di due diverse soluzioni appartenenti allo stesso progetto. Si descrivono quindi i passaggi minimi necessari per compilare ed eseguire un programma convertito:

[14] https://visualstudio.microsoft.com/it/vs/

[15] https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions/sdk.html

[16] https://software.intel.com/content/www/us/en/develop/articles/getting-started-with-sgx-sdk-for-windows.html

1. aprire Visual Studio e creare un nuovo progetto, Figura A.9;

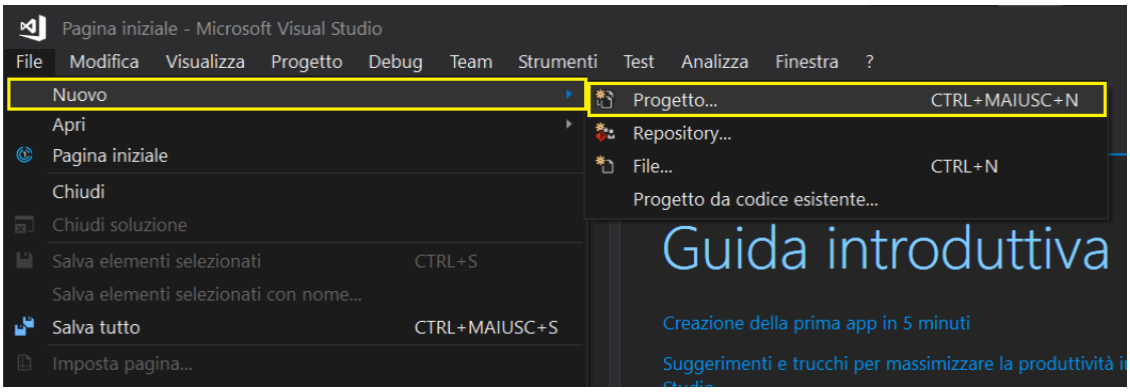


Figura A.9: Creazione di un nuovo progetto con Visual Studio.

2. creare il progetto con tipo “App console” e impostare il nome “application”, Figura A.10;

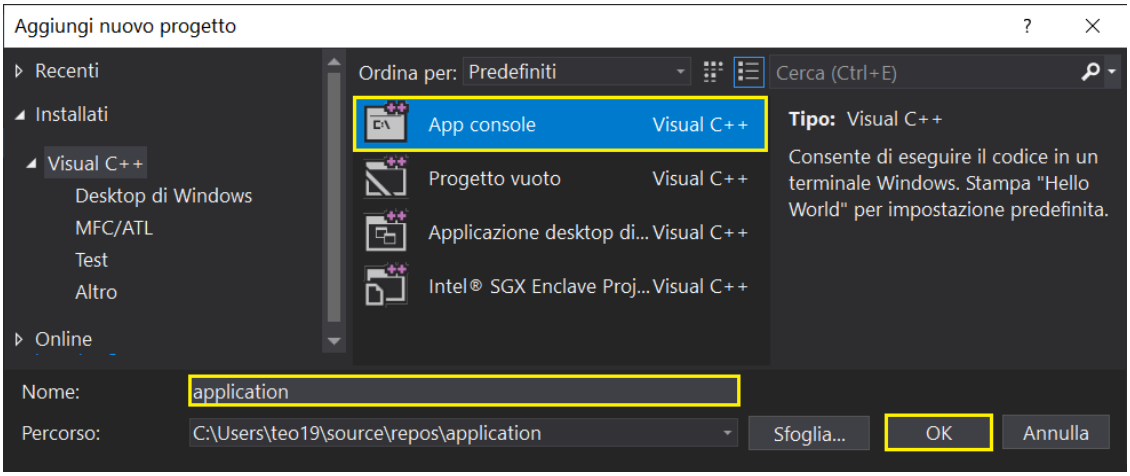


Figura A.10: Creazione progetto untrusted.

3. cliccare con il tasto destro su “Soluzione application”, scegliere “Aggiungi” e successivamente “Nuovo progetto...”, Figura A.11;

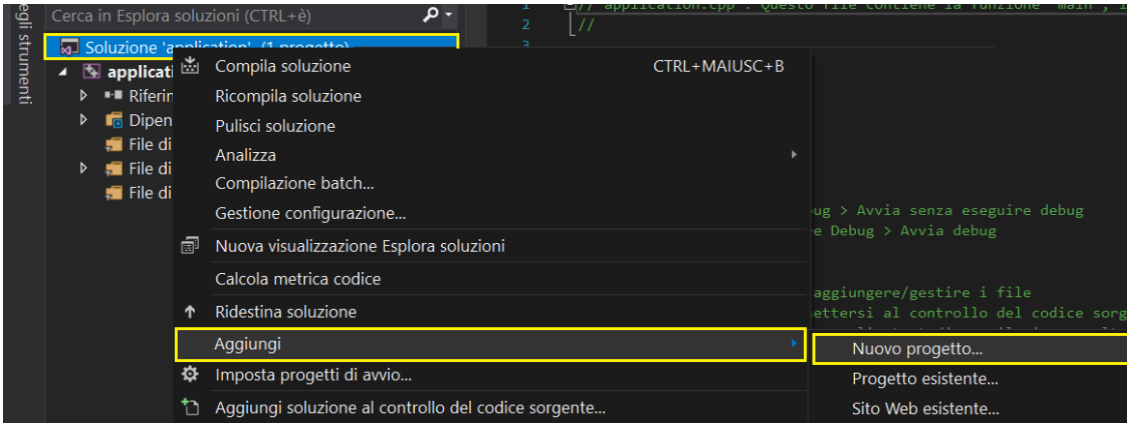


Figura A.11: Aggiunta di una nuova soluzione ad un progetto esistente.

4. creare il progetto con tipo “Intel SGX Enclave Project” e impostare il nome “enclave”, Figura A.12;

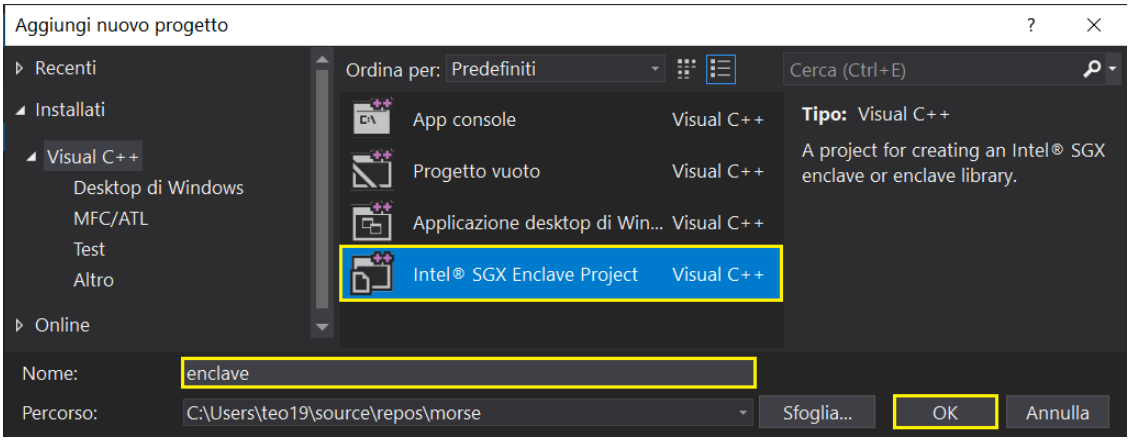


Figura A.12: Creazione progetto trusted.

5. cliccare “finish” nella procedura guidata, Figura A.13;

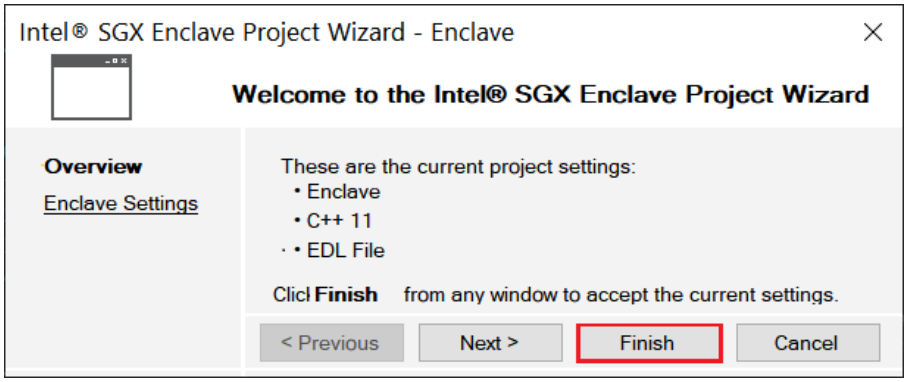


Figura A.13: Procedura guidata per la creazione del progetto trusted.

6. cliccare con il tasto destro su “application”, quindi “proprietà” e modificare il set di caratteri impostandolo come “Non impostato”, Figura A.14;

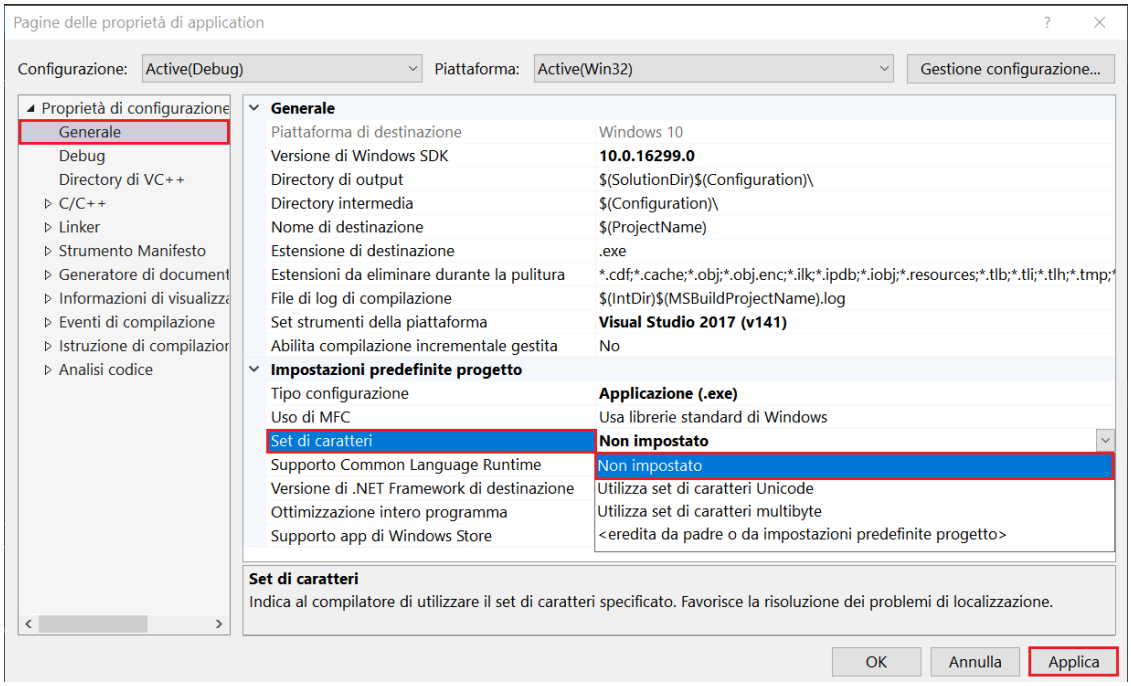


Figura A.14: Set di caratteri del progetto trusted.

7. spostarsi su “Debug”, quindi cambiare la directory di lavoro impostandola “`$(OutDir)`”, Figura A.15;

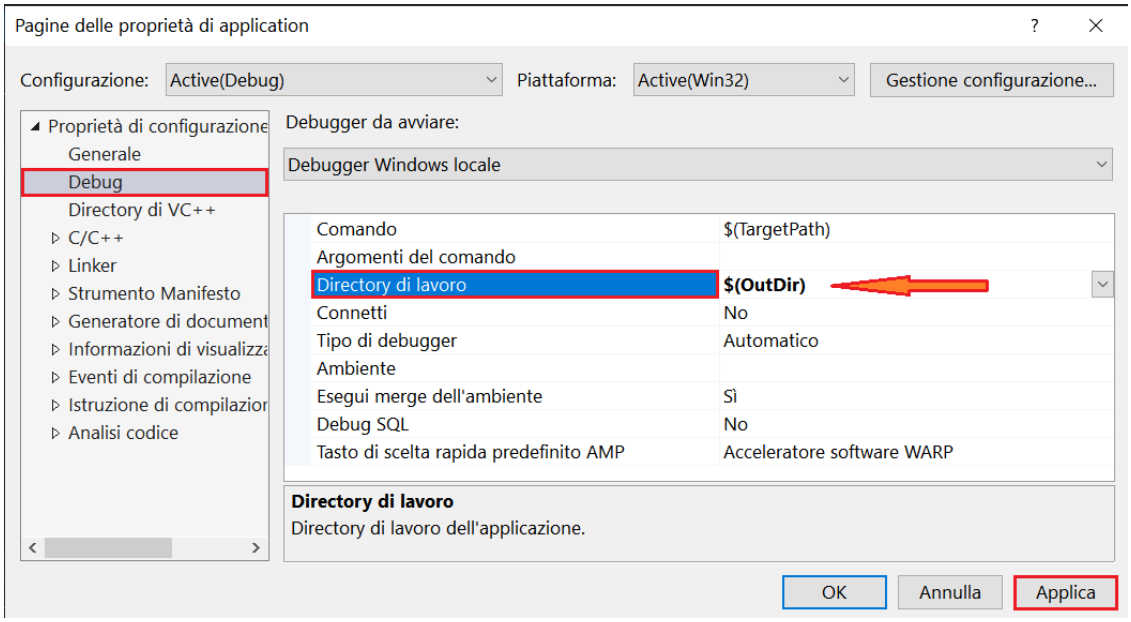


Figura A.15: Directory di lavoro del progetto trusted.

8. cliccare con il tasto destro su “enclave”, quindi “proprietà” e modificare la directory di lavoro impostandola “`$(OutDir)`”, Figura A.16;

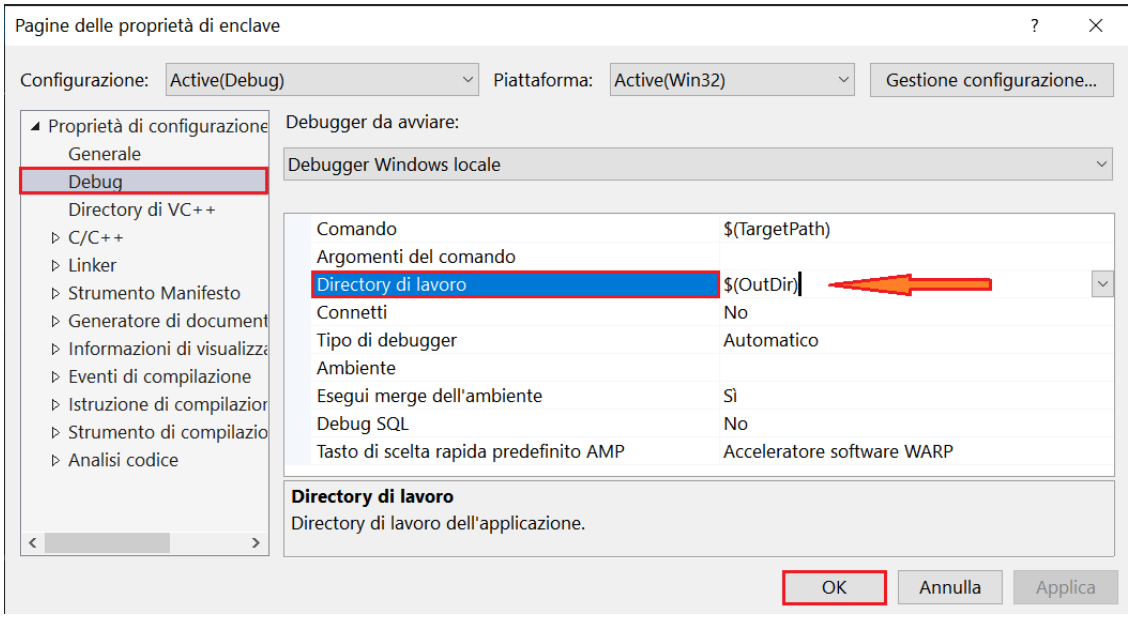


Figura A.16: Directory di lavoro del progetto untrusted.

9. copiare le due cartelle generate dal framework nella cartella del progetto di Visual Studio sovrascrivendo il file in conflitto (“`enclave.edl`” generato dal framework), Figura A.17;

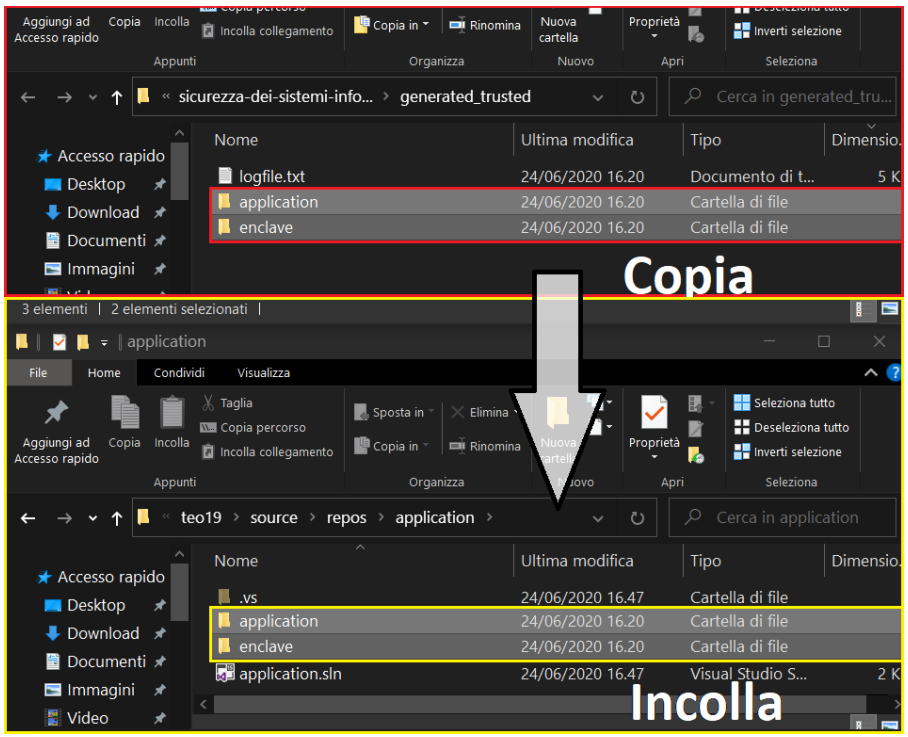


Figura A.17: File convertiti importati nelle cartelle di lavoro.

10. eliminare il file "`enclave.cpp`" dai sorgenti del progetto enclave;
11. importare il file sorgente dell'enclave (generato dal framework) nel progetto corrispondente: enclave → Source Files → Aggiungi → Elemento Esistente... → enclave.c;
12. eliminare il file "`application.cpp`" dai sorgenti del progetto application;
13. importare i file sorgenti utente (modificati dal framework) nel progetto application: application → Aggiungi → Elemento Esistente... → tutti i file utente + enclave_manager.h;
14. importare l'enclave, generata dal framework, nel progetto application attraverso la configurazione del plugin SGX, Figura A.18;

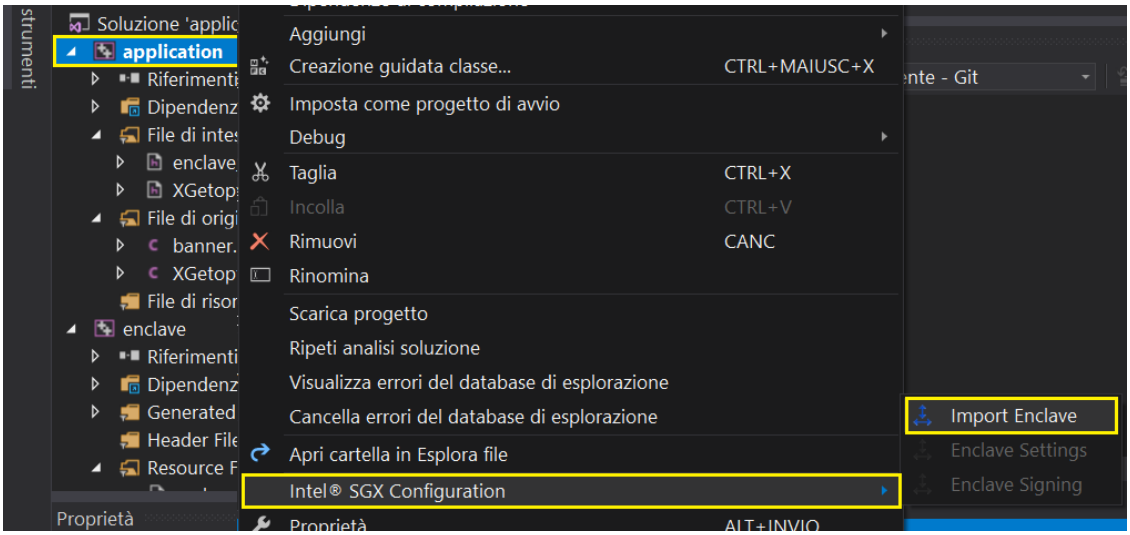


Figura A.18: Plugin Intel SGX per importare l'enclave nel progetto untrusted.

15. selezionare la soluzione dell'enclave da importare, applicare e chiudere la finestra, Figura A.19;

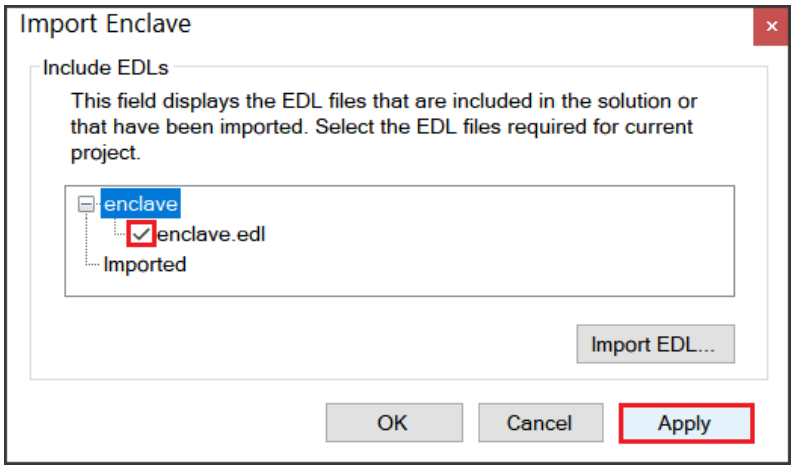


Figura A.19: Procedura per selezionare ed importare l'enclave.

16. selezionare la soluzione, compilare i progetti e verificare l'output generato, Figura A.20;

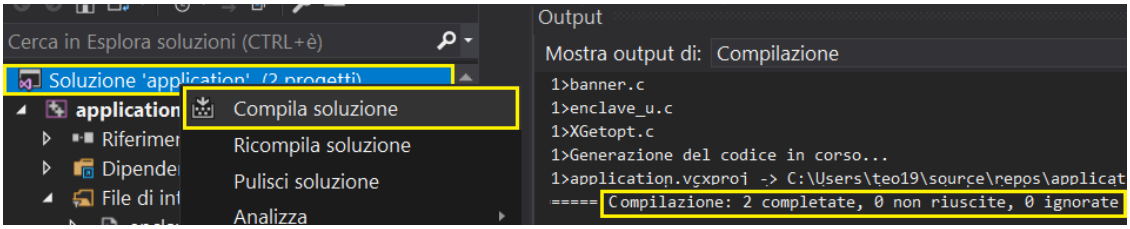


Figura A.20: Compilazione della soluzione con Visual Studio.

17. eseguire il programma convertito dal framework con Intel SGX, Figura A.21.

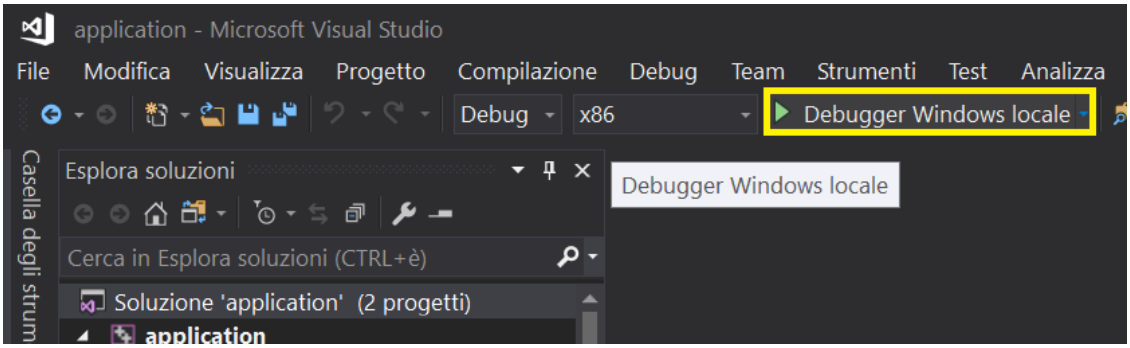


Figura A.21: Esecuzione del programma convertito.

Appendice B

Manuale sviluppatore

In questa appendice si descrive il funzionamento del framework a livello di codice implementativo. In particolare, l'Appendice B.1 descrive la struttura e l'elaborazione del framework, con particolare interesse alle funzionalità dei singoli moduli interni ed alle loro interazioni. Poi, si descrivono le dipendenze dai moduli esterni al framework, analizzando per ognuno le funzionalità importate. Infine, si descrivono gli strumenti esterni utilizzati dal framework, soffermandosi sui metodi, sulle opzioni e sulle funzionalità impiegate. Il codice del framework è disponibile nel file compresso "framework.zip". In particolare, i moduli approfonditi nelle prossime sezioni sono memorizzati nella cartella "python/src" a sua volta contenuta nel file compresso citato. Per proseguire con l'esecuzione o lo sviluppo del framework è necessario decomprimere la cartella compressa e memorizzarla su disco.

B.1 Struttura ed elaborazione del framework

Il framework è sviluppato con una struttura gerarchica centralizzata. Il codice è diviso in sette moduli diversi, ognuno appartenente all'omonimo file (.py). Tre di questi moduli (`ctagsInterface`, `framaInterface` e `gccInterface`) sono l'interfaccia Python di basso livello ai comandi esterni descritti in Appendice B.3 e sono responsabili della lettura dell'output generato e della gestione degli errori. Questi forniscono le funzionalità base dell'analisi statica e della ripulitura del codice. In particolare `ctagsInterface` fornisce una interfaccia per identificare tutti i nomi definiti nel codice, al fine di controllarne l'univocità ed asserirne il corretto riconoscimento da parte del framework. Infatti, i nomi delle definizioni vengono anche calcolati dal framework ed i risultati dei due strumenti vengono confrontati, in modo da riconoscere se il codice sorgente segue lo standard ANSI C (unica sintassi compatibile con il framework). Il modulo `framaInterface` invece, viene utilizzato per analizzare il codice semanticamente al fine di calcolare il grafo delle chiamate a partire da ogni funzione. In questo modo il framework recupera i dati ed utilizza queste dipendenze software per dividere il codice nelle due parti trusted ed untrusted. Infine, il modulo `gccInterface` viene utilizzato per ripulire ogni file sorgente dai commenti dell'utente al fine di ridurre la complessità delle espressioni regolari,

utilizzate nel riconoscimento delle entità. Inoltre, il modulo viene utilizzato per calcolare quale libreria standard ridefinita da SGX definisca una funzione data, come spiegato in Appendice B.3.3.

Ad un più alto livello di astrazione si posiziona il modulo `Function`, il quale modella una funzione nelle sue componenti base (tipo di ritorno, nome, argomenti e codice implementativo), fornisce delle operazioni per modellare le chiamate ECall ed OCall, e implementa i metodi per generare i codici di gestione delle chiamate citate.

I moduli `cFileReaderWriter` ed `edlFileWriter` si posizionano al livello di astrazione dei file sorgente. Il primo si occupa di aprire, e leggere un singolo file sorgente dell'utente, riconoscere le entità presenti nel codice ed identificare le annotazioni definite dall'utente. Inoltre, `cFileReaderWriter` implementa i metodi per modificare il codice sorgente, come l'aggiunta, la modifica o la rimozione di un'entità. Il modulo `edlFileWriter` invece, gestisce il file di descrizione `EDL` dell'enclave. Questo implementa i metodi per aggiungere le definizione delle strutture dati e per definire le chiamate ECall ed OCall. Infine, il modulo più astratto del framework è `codeAnalyser`, il quale gestisce l'intero flusso di conversione e definisce funzioni di alto livello per l'analisi e la modifica dei sorgenti.

In Figura B.1 sono rappresentati i moduli del framework e le loro relazioni. Inoltre, nella figura sono compresi i nomi delle variabili di classe e i nomi dei metodi di ogni modulo. Oltre ai moduli riportati, il framework definisce due ulteriori moduli, `framework` e `frameworkConfig`, utilizzati rispettivamente per il richiamo della funzione `main` (presente nel modulo `codeAnalyser`) e per la scrittura della configurazione del framework (l'insieme dei percorsi dei file da analizzare e la cartella principale del sorgente).

L'elaborazione del framework inizia con il modulo `framework` che legge la configurazione dal modulo `frameworkConfig` e richiama la funzione `main` con i parametri corretti. Quest'ultima esegue dei controlli iniziali richiamando le funzioni del modulo `codeAnalyser` che verificano l'installazione degli strumenti esterni (Ctags, Frama-C e GCC), controllano la configurazione delle librerie di SGX (in `gccInterface`), verificano l'esistenza dei file e generano un backup dell'applicazione da convertire.

Poi, viene eseguito il modulo `ctagsInterface` su tutti i file sorgente dell'utente, al fine di controllare l'univocità dei nomi e verificare la compatibilità del framework con la versione del linguaggio C utilizzata. Vengono quindi aperti i file sorgente con il modulo `cFileReaderWriter` che richiama al suo interno `gccInterface` per eseguire la ripulitura dei codici sorgenti dai commenti. Poi, vengono riconosciute tutte le entità del codice attraverso le espressioni regolari contenute nel modulo `cFileReaderWriter`. In questo momento si riconoscono anche le funzioni destinazione delle chiamate ECall ed OCall. Quindi, la funzione `main` controlla l'esistenza delle definizioni dei tipi degli argomenti delle chiamate ECall ed OCall, esegue la divisione del sorgente nei due contesti (trusted ed untrusted) e calcola le librerie di SGX da importare nel codice dell'enclave. Al termine dell'esecuzione della divisione si analizza la separazione dei due contesti controllando l'assenza di dipendenze software reciproche.

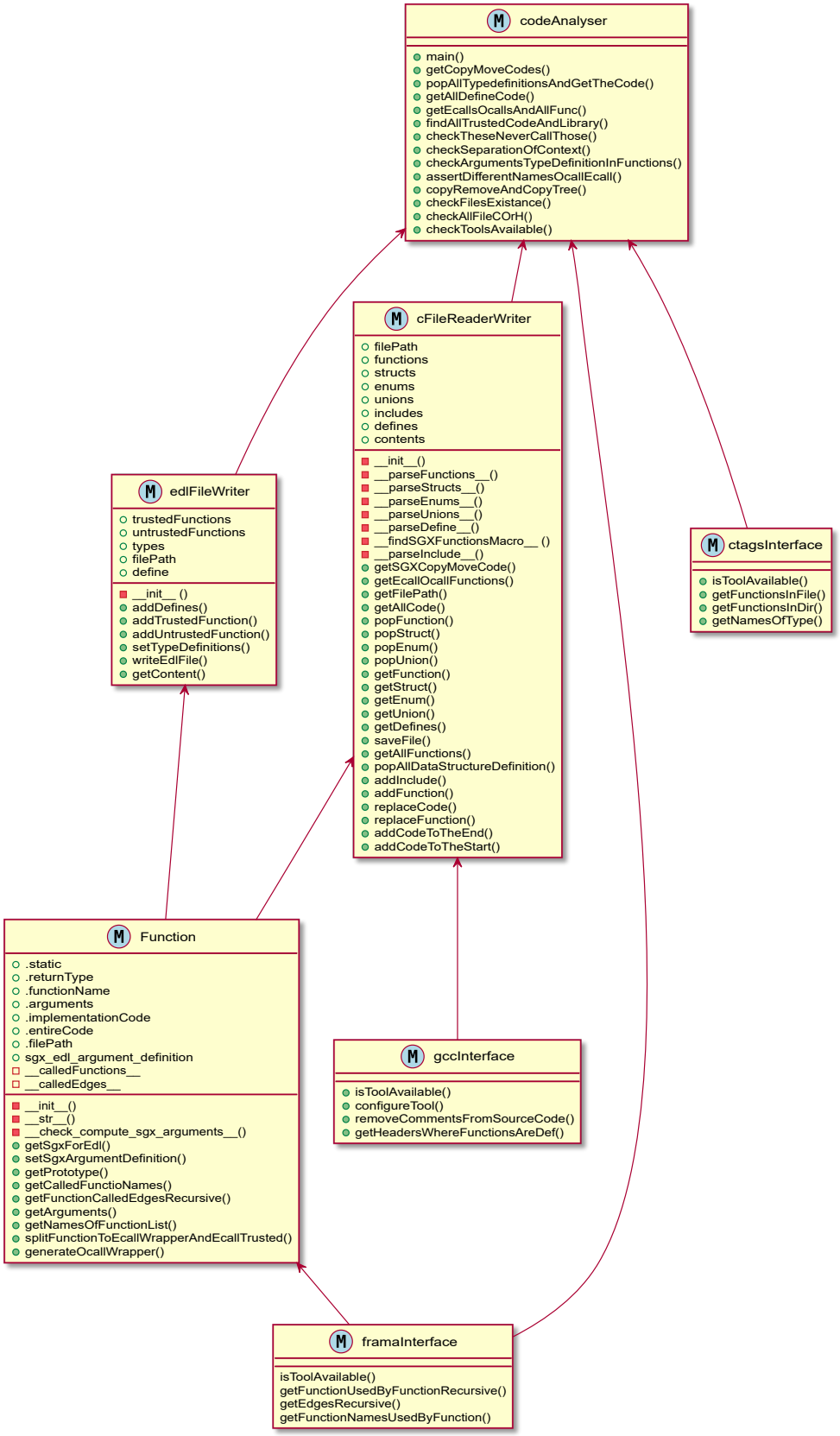


Figura B.1: Moduli interni del framework di conversione.

La funzione `main` esegue quindi le modifiche sul codice sorgente dell'utente. Inizialmente, vengono generate le funzioni di gestione delle chiamate OCall, inserite

nel codice dell'enclave. Poi, vengono spostate le funzioni considerate sicure e le chiamate ECall all'interno dell'enclave. Quindi, si genera il codice untrusted utilizzato esternamente per la gestione degli errori delle chiamate ECall. Vengono quindi inserite nel codice dell'enclave le costanti globali precedentemente annotate dall'utente con le direttive di copia e spostamento e viene effettuato il salvataggio del sorgente dell'enclave.

Infine, attraverso l'uso del modulo `edlFileWriter`, viene generato il codice in linguaggio `EDL` per la definizione dell'enclave. Inizialmente, vengono esportate al suo interno tutte le strutture dati definite dall'utente. Tutti i file sorgente untrusted vengono quindi aggiornati per includere le definizioni appena rimosse, attraverso l'inclusione della libreria `enclave_manager.h`, necessaria anche per l'allocazione e la gestione dell'enclave. Poi, attraverso il modulo `Function`, si generano i costrutti nel formato `EDL` utilizzati per definire le chiamate ECall ed OCall nel file di descrizione dell'enclave. Al termine dell'operazione vengono aggiunte nel file `EDL` le definizioni delle direttive `#define` dell'utente.

In conclusione si esegue il salvataggio di tutti i file sorgente nella cartella `application` ed i file dell'enclave nella `enclave`. Queste due cartelle vengono entrambe memorizzate all'interno del direttorio `generated_trusted`, situato nella cartella radice del framework.

B.2 Dipendenze Python del framework

In questa sezione si analizzano i moduli Python esterni utilizzati dal framework.

subprocess[1] consente di generare nuovi processi a partire da comandi eseguibili sulla console, leggerne il risultato dell'esecuzione sia dallo standard output che dallo standard error. Questo modulo può connettere più processi in modalità pipeline ottenendo lo stesso comportamento della linea di comando.

pydot[2] può aprire e parsificare un file testuale scritto in linguaggio `dot`[3]. Questa tipologia di file rappresenta modelli a grafo e viene utilizzata dal framework per leggere i dati del call-graph generato da Frama-C, spiegato in Appendice B.3.2.

regex[4] implementa le funzionalità delle espressioni regolari ed è un'alternativa al modulo `re`. Viene utilizzato nel progetto poiché il modulo `re` non permette le espressioni regolari ricorsive, ovvero le espressioni che rappresentano una regola annidata come l'apertura e la chiusura dello stesso numero di parentesi. Questo modulo è particolarmente utilizzato quando si parsificano i file sorgente per individuare gli elementi di interesse, come funzioni, strutture dati e definizioni.

[1] https://docs.python.org/3/library/subprocess.html

[2] https://pypi.org/project/pydot/

[3] https://www.graphviz.org/pdf/dotguide.pdf

[4] https://pypi.org/project/regex/

re[5] è il modulo già integrato per eseguire le espressioni regolari su un testo.

tqdm[6] crea sullo standard output una barra di progresso in modo che l'utente possa verificare l'avanzamento dell'esecuzione del framework. Questo modulo fornisce graficamente la percentuale di avanzamento, la stima del tempo rimanente e il tempo passato durante le operazioni che utilizzano la barra.

logging3[7] è un modulo capace di gestire i messaggi di log che può essere configurato per scriverli su un file o stamparli in console. Questo consente di filtrare i messaggi secondo dei livelli in modo da personalizzare quali tipi di informazioni visualizzare in uscita. I livelli dei messaggi utilizzati dal framework sono: debug, info, warning ed error.

sys[8] è l'interfaccia dell'ambiente dell'interprete che viene utilizzata per accedere agli standard di console, stdout e stderr, utilizzati da `logging3`.

shutil[9] è il modulo che fornisce i comandi per accedere, copiare ed eliminare cartelle e file. Nel framework viene utilizzato per generare un backup dei file in modo da modificare i file copiati senza alterare il codice sorgente originale. Questo modulo viene utilizzato anche per ripulire ed eliminare i file generati dagli strumenti esterni (descritti in Appendice B.3) come GCC e Frama-C durante i calcoli necessari al framework.

time[10] fornisce l'accesso all'orologio di sistema in modo da poter calcolare il tempo di esecuzione del framework, utilizzato nelle analisi descritte in Sezione 6.4.

os[11] fornisce l'accesso alle istruzioni per verificare l'esistenza dei file, come i file sorgente utente ed il file di configurazione del framework analizzato in Appendice B.3.3.

Alcuni di questi moduli sono già presenti nelle librerie standard di Python mentre i rimanenti devono essere installati singolarmente attraverso i comandi del package manager `pip`[12].

B.3 Dipendenze esterne del framework

In questa sezione si definiscono gli strumenti esterni utilizzati dal framework. Poi, si approfondisce per ogni strumento la sua modalità di impiego.

[5] https://docs.python.org/3/library/re.html

[6] https://github.com/tqdm/tqdm

[7] https://pypi.org/project/logging3/

[8] https://docs.python.org/3/library/sys.html

[9] https://docs.python.org/3/library/shutil.html

[10] https://docs.python.org/3/library/time.html

[11] https://docs.python.org/3/library/os.html

[12] https://pip.pypa.io/en/stable/

Il framework necessita che gli strumenti esterni siano disponibili nel percorso corrente. Quindi, una volta proceduto all'installazione degli stessi, è necessario aggiornare la variabile di ambiente del sistema operativo (generalmente nominata `$PATH`) ed aggiungere ad essa i percorsi assoluti a singoli file eseguibili degli strumenti.

Nelle sezioni successive si descrivono gli utilizzi e le versioni degli strumenti utilizzati dal framework: Ctags, Frama-C e GCC.

B.3.1 Universal Ctags

Il framework utilizza Universal Ctags[13] per analizzare il codice sorgente e ricevere il nome di funzioni, strutture dati, unioni ed enumerazioni. Il framework controlla univocità dei nomi ricevuti dallo strumento e ferma l'esecuzione dell'analisi in caso di ridefinizioni, per le motivazioni spiegate in Sezione 5.3.1. Poi, il framework apre i file sorgente e crea un modello di dati delle entità da utilizzare nelle analisi successive descritte in Sezione 5.3.2. Da questo modello si ricava anche lo stesso tipo di informazioni appena generate da Ctags. In questo modo è possibile verificare che i nomi riconosciuti dal framework siano identici in sintassi e numero a quelli individuati da Ctags. Questa operazione è necessaria poiché il framework riconosce solo le sintassi dello standard `ANSI-C`, mentre Ctags è i grado di analizzare codice meno recente come quello nello standard `K&R`. Quindi, nel caso in cui le due analisi portino a risultati diversi, si riconosce l'utilizzo di una sintassi C non compatibile con il framework.

Il comando di Ctags viene eseguito dal framework in modalità pipeline[14] con il comando `printf`, in modo da comunicare simultaneamente tutti i percorsi dei file e analizzarli quindi con un unico comando di Ctags. In Figura B.2 è riportato il comando principale da eseguire sulla console di Python, equivalente a quello utilizzato dal framework.

```
printf file_1 file_2 ... file_n |
ctags -f - --c-types=dfgsu --fields=a-f-i-k-K-l-m-n-s-S-z-t -L -
```

Figura B.2: Comandi per l'utilizzo di Ctags.

L'opzione `'-f -'` viene utilizzata per specificare che il risultato deve essere stampato sulla console, piuttosto che scritto su un file. L'impostazione `'-L -'` viene utilizzata per specificare che i percorsi ai file da analizzare sono comunicati tramite pipeline. L'opzione `'--fields=a...-t'` disabilita tutte le informazioni disponibili nei risultati tranne il nome dell'entità, il percorso del file e la definizione. Infine, l'impostazione `'--c-types=dfgsu'` viene utilizzate per filtrare i tipi degli

[13] https://ctags.io/

[14] Con modalità pipeline si intende l'esecuzione in serie di due comandi in cui l'output del primo comando corrisponde all'input del secondo.

elementi e selezionare solo quelli desiderati: `'d'` per le direttive `#define`, `'f'` per le funzioni, `'g'` per le enumerazioni, `'s'` per le strutture dati e `'u'` per le unioni di dati.

B.3.2 Frama-C

Il grafo delle chiamate è un modello in cui i nodi rappresentano le funzioni eseguite e gli archi suggeriscono la direzione delle chiamate a funzione. Questo tipo di grafo può essere costruito dinamicamente a partire da una registrazione dell'esecuzione di un programma oppure può essere costruito staticamente a partire da codice sorgente. Questa ultima soluzione, utilizzata da Frama-C, è destinata a rappresentare una qualsiasi esecuzione del programma, che nel caso dipenda da un input imprevedibile, potrà solo approssimare in eccesso tutte le possibili chiamate dipendenti dalle incognite.

Il framework utilizza lo strumento esterno Frama-C[15] al fine di eseguire l'analisi statica sul codice sorgente. Infatti, durante la conversione si necessita di calcolare tutte le funzioni chiamate dall'interno di un'altra funzione, come descritto in Sezione 5.3.2. Il risultato di quest'analisi deve prevedere sia l'uso di puntatori a funzione[16] sia chiamate a funzioni utilizzando l'approccio standard. Il codice che fa uso dei puntatori a funzione rende più difficile individuare quali siano tutte le funzioni invocate poiché il vero nome della funzione invocata non è visibile ed è necessario analizzare il flusso dei valori assunti dalla memoria puntata.

Frama-C espone diverse opzioni di analisi che potrebbero essere impiegate per identificare le funzioni usate da una funzione: `users`, `syntactic callgraph` e `semantic callgraph`. Tuttavia, l'opzione utilizzata dal framework è l'analisi semantica, poiché è l'unica in grado di costruire un grafo delle chiamate che comprenda anche l'analisi dei puntatori a funzione.

```
frama-c -scg tmp.dot -lib-entry -main functionName filepath
```

Figura B.3: Comando di Frama-C per l'esecuzione dell'analisi semantica.

In Figura B.3 è riportato il comando principale equivalente a quello utilizzato dal framework per eseguire l'analisi semantica del codice. L'opzione `semantic callgraph` costruisce un file in formato `dot` che viene successivamente analizzato dal framework. Questo formato è utilizzato per rappresentare testualmente un grafo generico[17].

In Figura B.4 è riportato il grafo delle chiamate costruito da Frama-C a partire dal codice sorgente. Il grafo ha come nodo radice la funzione il cui nome è specificato

[15] https://frama-c.com/download_boron.html

[16] Un puntatore a funzione è un indirizzo di memoria che quando viene referenziato, invoca una funzione passandole zero o più argomenti specificati come una chiamata a funzione standard.

[17] https://www.graphviz.org/pdf/dotguide.pdf

dall'opzione `-main`, mentre l'opzione `-lib-entry` specifica di eseguire l'analisi senza presumere i valori iniziali delle variabili globali.

Come è possibile vedere in Figura B.4b, il grafo in uscita contiene tutte le funzioni raggiungibili a partire dalla funzione radice `add`. L'analisi semantica comprende il valore dei puntatori a funzione e ricostruisce correttamente il grafo delle chiamate del programma. Tuttavia, il grafo è processato con la direzione delle frecce nel verso opposto rispetto all'approccio standard ed è quindi necessario interpretare come il punto di arrivo del grafo la funzione di partenza `add`.

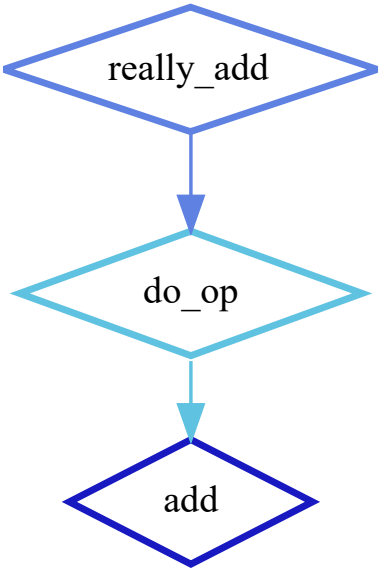
```
enum op { ADD, MULT };

int really_add(int u, int v){
    return u + v;
}

int do_op(enum op op, int u, int v){
    if (op == ADD)
        return really_add(u, v);
    else if (op == MULT)
        return really_mult(u, v);
    else
        return -1;
}

int add(int x, int y){
  int (*f)(enum op, int, int) = &do_op;
  return f(ADD, x, y);
}
```

(a) Codice di esempio.



(b) Grafo in formato `dot`.

Figura B.4: Esempio di utilizzo dell'opzione `semantic callgraph`.

L'analisi semantica delle chiamate a funzione produce dei dati sia su standard output che su file. Nel caso in cui la funzione di partenza non sia presente nel file sorgente, il programma termina con valore di ritorno 0 e non verrà creato nessun file rappresentante il grafo. Un altro caso di errore è la mancanza del file specificato o l'errato percorso fornito che termina con il valore di ritorno 1. Nel caso ci siano funzioni ricorsive, la ricorsione viene rilevata e viene rappresentata graficamente con una freccia che parte e termina sullo stesso nodo. In ultimo, è possibile che il comando ritorni il codice 5, se la funzione specificata non ha funzioni chiamate al suo interno e anche in questo caso non viene generato il file del grafo.

B.3.3 GCC

Il framework utilizza il compilatore GCC[18] per due principali operazioni: rimuovere i commenti dal codice sorgente e calcolare il file di configurazione per le librerie di Intel SGX SDK. Il primo utilizzo serve a semplificare le espressioni regolari utilizzate dal framework per identificare le entità del codice (come funzioni, strutture dati, enumerazioni e unioni), come descritto in Sezione 5.3.1. In Figura B.5 è riportato il comando eseguito dal framework per processare un file sorgente e rimuovere i commenti del codice. L'opzione `-fpreprocessed`[19] viene utilizzata per processare il file al fine di rimuovere i commenti, mentre le opzioni rimanenti servono ad impedire l'esecuzione di altre operazioni di preprocessore, come l'espansione delle macro.

```
gcc -fpreprocessed -dD -E filepath
```

Figura B.5: Comando di `GCC` per rimuovere i commenti dal codice sorgente.

In Figura B.6 è definito il comando per generare il file di configurazione del framework per le librerie di SGX. Questo file contiene la lista di funzioni standard ridefinite da SGX ed associa ad ognuna di esse il percorso alla propria libreria. Questo file è utilizzato per verificare le funzioni chiamate all'interno della regione protetta e per la generazione delle librerie da includere nell'enclave, come spiegato in Sezione 5.3.2. Il comando definisce i percorsi delle cartelle contenenti le librerie di SGX, durante l'esecuzione del processo, con l'opzione `-I` e impedisce l'importazione delle librerie standard attraverso le opzioni `-nostdlib` e `-nostartfiles`. Inoltre, importa il file di configurazione `__convert_tool_configure.c` dove sono definite le librerie di SGX da importare e specifica l'opzione `--aux-info` per generare il file di configurazione del framework.

```
gcc -nostdlib -nostartfiles -I./include/tlibc -I./include/
   -I./include/tlibc/sys -aux-info sgx_configuration_file
   ./include/__convert_tool_configure.c
```

Figura B.6: Comando per generare il file di configurazione del framework.

[18] https://www.cygwin.com/

[19] https://linux.die.net/man/1/gcc

Bibliografia

[1] L. Regano, "An Expert System for Automatic Software Protection". PhD thesis, Politecnico di Torino, July 2019. https://iris.polito.it/retrieve/handle/11583/2751495/270821/Regano_PhD_Thesis.pdf

[2] Joint Task Force Transformation Initiative, "Managing Information Security Risk: Organization, Mission, and Information System View." NIST SP800-39, March 2011, DOI 10.6028/nist.sp.800-39

[3] V. Costan and S. Devadas, "Intel SGX Explained", IACR Cryptology ePrint Archive, vol. 2016, January 2016. https://eprint.iacr.org/2016/086.pdf

[4] J. Daemen and V. Rijmen, "The Design of Rijndael: AES - The Advanced Encryption Standard", Information Security and Cryptography, Springer Berlin Heidelberg, November 2002, ISBN: 978-3-540-42580-9

[5] M. J. Dworkin, "Recommendation for block cipher modes of operation." NIST SP 800-38A, December 2001, DOI 10.6028/nist.sp.800-38a

[6] S. Gueron, "A Memory Encryption Engine Suitable for General Purpose Processors", IACR Cryptology ePrint Archive, vol. 2016, February 2016. https://eprint.iacr.org/2016/204

[7] E. Barker, L. Chen, A. Roginsky, A. Vassilev, and R. Davis, "Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography." NIST SP 800-56A, April 2018, DOI 10.6028/nist.sp.800-56ar3

[8] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware Guard Extension: Using SGX to Conceal Cache Attacks", DIMVA 2017: 14th Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment, Bonn (Germany), July 6–7, 2017, pp. 3–24, DOI 10.1007/978-3-319-60876-1_1

[9] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution", EuroS&P 2019: IEEE European Symposium on Security and Privacy, Stockholm (Sweden), June 17–19, 2019, pp. 142–157, DOI 10.1109/eurosp.2019.00020

[10] "Intel(R) Software Guard Extensions (SGX) SW Development Guidance for Potential Bounds Check Bypass (CVE-2017-5753) Side Channel Exploits." Intel Corporation, February 2018, https://software.intel.com/sites/default/files/180204_SGX_SDK_Developer_Guidance_v1.0.pdf

[11] M. Schwarz, S. Weiser, and D. Gruss, "Practical Enclave Malware with Intel SGX", DIMVA 2019: 16th Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment, Gothenburg (Sweden), June 19–20, 2019, pp. 177–196, DOI 10.1007/978-3-030-22038-9_9

[12] "Intel(R) Software Guard Extensions Developer Reference for Linux* OS." Intel Corporation, April 2020, https://download.01.org/intel-sgx/sgx-

linux/2.9.1/docs/Intel_SGX_Developer_Reference_Linux_2.9.1_Open_Source.pdf

[13] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption." Advanced Micro Devices, April 2016, https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf

[14] W. Futral and J. Greene, "Fundamental Principles of Intel(R) TXT", Intel(R) Trusted Execution Technology for Server Platforms: A Guide to More Secure Datacenters (P. Jeffrey and H. Patrick, eds.), pp. 15–36, Apress, September 2013, DOI 10.1007/978-1-4302-6149-0_2

[15] W. Arthur, D. Challener, and K. Goldman, "Platform Security Technologies That Use TPM 2.0", A Practical Guide to TPM 2.0: Using the New Trusted Platform Module in the New Age of Security (W. Steve and H. Patrick, eds.), pp. 331–348, Apress, January 2015, DOI 10.1007/978-1-4302-6584-9_22

[16] "TPM Main Part 1 Design Principles." Trusted Computing Group, March 2011, https://trustedcomputinggroup.org/wp-content/uploads/TPM-Main-Part-1-Design-Principles_v1.2_rev116_01032011.pdf

[17] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-hashing for message authentication." RFC-2104, February 1997, DOI 10.17487/rfc2104

[18] M. L. Nelson, "A Survey of Reverse Engineering and Program Comprehension", Tech. Rep. cs.SE/0503068, European Organization for Nuclear Research (CERN), March 2005. http://cds.cern.ch/record/829443

[19] T. László and A. Kiss, "Obfuscating C++ Programs via Control Flow Flattening", Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae. Sectio Computatorica, vol. 30, June 2007. http://ac.inf.elte.hu/Vol_030_2009/003.pdf

[20] C. Collberg, C. Thomborson, and D. Low, "A Taxonomy of Obfuscating Transformations", Computer Science Technical Reports 148, Department of Computer Science, The University of Auckland (New Zealand), July 1997. https://researchspace.auckland.ac.nz/handle/2292/3491

[21] M. L. Fredman, J. Komlos, and E. Szemeredi, "Storing a sparse table with O(1) worst case access time", SFCS 1982: 23rd Annual Symposium on Foundations of Computer Science, Chicago (IL, USA), November 3–5, 1982, pp. 165–169, DOI 10.1109/sfcs.1982.39

[22] H. Fang, Y. Wu, S. Wang, and Y. Huang, "Multi-stage Binary Code Obfuscation Using Improved Virtual Machine", Information Security (X. Lai, J. Zhou, and H. Li, eds.), pp. 168–181, Berlin (Germany), Springer Berlin Heidelberg, October 2011, DOI 10.1007/978-3-642-24861-0_12

[23] B. D. Sutter and B. Coppens, "Anti-callback Stack Checks", ASPIRE Project Deliverable 2.08: Offline Code Protection Report, November 2015. https://aspire-fp7.eu/sites/default/files/D2.08-ASPIRE-Offline-Code-Protection-Report.pdf

[24] A. Viticchié, C. Basile, A. Avancini, M. Ceccato, B. Abrath, and B. Coppens, "Reactive Attestation: Automatic Detection and Reaction to Software Tampering Attacks", SPRO 2016: ACM Workshop on Software PROtection, New York (NY, USA), October 24–28, 2016, pp. 73–84, DOI 10.1145/2995306.2995315

[25] A. Cabutto, P. Falcarin, B. Abrath, B. Coppens, and B. De Sutter, "Software Protection with Code Mobility", MTD 2015: 2nd ACM Workshop on Moving

Target Defense, Denver (CO, USA), October 12–16, 2015, pp. 95–103, DOI 10.1145/2808475.2808481

[26] B. Abrath, B. Coppens, S. Volckaert, J. Wijnant, and B. De Sutter, "Tightly-Coupled Self-Debugging Software Protection", SSPREW 2016: 6th Workshop on Software Security, Protection, and Reverse Engineering, Los Angeles (CA, USA), December 5–6, 2016, pp. 1–10, DOI 10.1145/3015135.3015142

[27] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs", Journal of the ACM, vol. 59, May 2012, DOI 10.1145/2160158.2160159

[28] S. Brenner, C. Wulf, D. Goltzsche et al., "SecureKeeper: Confidential ZooKeeper Using Intel SGX", Middleware 2016: 17th International Middleware Conference, Trento (Italy), December 12–16, 2016, pp. 1–13, DOI 10.1145/2988336.2988350

[29] Chen, Guoxing and Chen, Sanchuan and Xiao, Yuan and Zhang, Yinqian and Lin, Zhiqiang and Lai, Ten H., "SCONE: Secure Linux Containers with Intel SGX", OSDI 2016: 12th USENIX Conference on Operating Systems Design and Implementation, Savannah (GA, USA), November 2–4, 2016, pp. 689–703. https://pdfs.semanticscholar.org/3d14/235977cd26ed5adb87d65cf0eb65d120fa4a.pdf

[30] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. D. Turck, and R. Boutaba, "Network Function Virtualization: State-of-the-Art and Research Challenges", IEEE Communications Surveys & Tutorials, vol. 18, September 2015, pp. 236–262, DOI 10.1109/comst.2015.2477041

[31] M. Shih, M. Kumar, T. Kim, and A. Gavrilovska, "S-NFV: Securing NFV States by Using SGX", SDN-NFV 2016: ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization, New Orleans (LA, USA), March 9–11, 2016, pp. 45–48, DOI 10.1145/2876019.2876032

[32] J. Thones, "Microservices", IEEE Software, vol. 32, January 2015, pp. 116–116, DOI 10.1109/ms.2015.11

[33] S. Brenner, T. Hundt, G. Mazzeo, and R. Kapitza, "Secure Cloud Micro Services Using Intel SGX", Distributed Applications and Interoperable Systems (L. Y. Chen and H. P. Reiser, eds.), pp. 177–191, Springer International, May 2017, DOI 10.1007/978-3-319-59665-5_13

[34] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P. L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer, and P. Pietzuch, "Glamdring: Automatic Application Partitioning for Intel SGX", USENIX ATC 2017: USENIX Annual Technical Conference, Santa Clara (CA, USA), July 12–14, 2017, pp. 285–298. https://www.usenix.org/system/files/conference/atc17/atc17-lind.pdf