UNIVERSITÀ DEGLI STUDI DI NAPOLI PARTHENOPE

DIPARTIMENTO DI INGEGNERIA

Relazione Tecnica:

Esecuzione WebAssembly in SGX (modalità SIM) con pipeline FS → IPFS e monitoraggio HIDS

Corso:

Sicurezza dei Sistemi Operativi e Cloud

A cura di:

Francesco Petillo, Gabriele Esposito

ANNO ACCADEMICO 2024/2025

INDICE

0) Introduzione

Il presente lavoro descrive lo sviluppo e l'integrazione di un ambiente sicuro per l'esecuzione di codice WebAssembly (WASM) all'interno di un Trusted Execution Environment (TEE), utilizzando la tecnologia Intel Software Guard Extensions (SGX) in modalità simulata, e il runtime WAMR.

Il progetto nasce con l'obiettivo di combinare aspetti di sicurezza informatica, containerizzazione e gestione distribuita dei dati, offrendo un'esperienza pratica su tecnologie emergenti ampiamente utilizzate in contesti come il cloud computing, la blockchain e lo storage decentralizzato.

Secondo le linee guida impartiteci, ci siamo proposti di creare una pipeline completa in cui un modulo WASM esegue calcoli isolati, scrive i risultati su filesystem e li pubblica su IPFS, ottenendo un identificatore univoco (CID) verificabile tramite rete P2P e gateway HTTP. La scelta di IPFS si basa sulla sua natura content-addressed: l'idea è che ogni file caricato genera un hash deterministico legato al contenuto, garantendo così l'immutabilità dei dati e la possibilità di verificarne l'integrità in qualsiasi momento. Questa caratteristica risulta perfetta con le esigenze di sicurezza e tracciabilità dei dati trattati in ambienti TEE.

Non disponendo di un reale hardware Intel-Sgx (rimosso sui computer destinati al mercato consumer) abbiamo optato per lo sviluppo in modalità SIM.
La simulazione SGX permette di riprodurre le condizioni di isolamento e protezione delle enclave anche in assenza di hardware dedicato, consentendoci di sperimentare le funzionalità di esecuzione sicura, gestione della memoria e protezione delle informazioni sensibili. La pipeline proposta non si limita all'esecuzione isolata del codice, ma integra anche un sistema di monitoraggio host-based (HIDS) tramite OSSEC, configurato per intercettare modifiche al file di output critico, anomalie nei log del demone IPFS e comportamenti sospetti dell'applicazione. Questo approccio multidisciplinare ci ha permesso di combinare teoria e pratica, sperimentando le dinamiche di sicurezza, logging e auditing in un ambiente controllato.

La documentazione prodotta accompagna ogni fase del progetto, descrivendo i motivi delle scelte progettuali, i comandi eseguiti, i risultati attesi e i test effettuati. Lo scopo di questa relazione è quindi:

- fornire una descrizione dettagliata delle tecnologie e delle metodologie adottate
- permettere a chiunque di replicare l'esperimento in modo chiaro e trasparente, offrendo così una panoramica completa sulle possibilità offerte dall'integrazione tra TEE, esecuzione di codice isolato, storage distribuito e sistemi di monitoraggio della sicurezza.

1) Abstract

Cosa dimostriamo: un server che opera in ambiente isolato (SGX *simulato*), esegue codice **WASM con WAMR**, scrive su **filesystem** un risultato e lo pubblica su **IPFS** (content-addressed); il tutto è **monitorato** da un **HIDS (OSSEC)**. Mostriamo anche l'ottimizzazione **AOT** di WAMR e una **API /sum** con *attestazione simulata* a monte della logica.

Questo progetto mostra concretamente come sia possibile combinare diverse tecnologie per garantire sicurezza, integrità e affidabilità dei dati e del codice in esecuzione.

- **TEE / SGX (in modalità simulata)**: permette di eseguire codice e gestire dati in modo **riservato e integro**, anche se in questo caso l'ambiente SGX è simulato per mancanza dell'hardware.
- **WebAssembly (Wasm)**: garantisce **portabilità tra piattaforme**, **isolamento attraverso la sandbox WASI** e riduce al minimo la **superficie d'attacco**, migliorando la sicurezza.
- **IPFS**: usato per salvare i risultati, consente di **verificare l'integrità dei dati** grazie al **CID**, un identificativo basato sul contenuto stesso (hash).
- **HIDS (Host-based Intrusion Detection System)**: monitora il sistema alla ricerca di **manomissioni o comportamenti anomali**, controllando sia i file di output che i log del demone IPFS.

2) Obiettivi

Il nostro progetto si propone di realizzare un ambiente completo in cui sia possibile sperimentare e comprendere le dinamiche di esecuzione sicura del codice, gestione dei dati e monitoraggio della sicurezza in un contesto isolato. Gli obiettivi principali sono molteplici e si concentrano su diversi aspetti tecnologici e didattici:

1. **Esecuzione sicura del codice in ambiente isolato (SGX simulato)**
   L'obiettivo primario è dimostrare come un server possa operare in un ambiente isolato, proteggendo codice e dati da interferenze esterne. Anche se la modalità SGX utilizzata è simulata, il progetto riproduce fedelmente il concetto di enclave: il codice eseguito al suo interno è protetto, la memoria è isolata e le operazioni critiche possono essere controllate, fornendo un contesto realistico per test e sperimentazioni.
2. **Esecuzione di moduli WebAssembly con WAMR**
   Un altro obiettivo chiave è far comprendere il ruolo di WebAssembly come strumento per eseguire codice portabile e isolato. Grazie al runtime WAMR e alla sandbox WASI, il progetto mostra come sia possibile eseguire moduli WASM che interagiscono con il filesystem in modo controllato, riducendo la superficie d'attacco e garantendo l'integrità dei processi.
3. **IPFS**
   La pipeline prevede la scrittura di un risultato su filesystem e la sua pubblicazione su IPFS. Questo permette di illustrare come il content addressing, basato su hash, garantisca l'immutabilità e la verificabilità dei dati generati. Ogni file pubblicato produce un CID univoco, che può essere verificato tramite gateway HTTP o rete P2P, rendendo il processo trasparente e riproducibile.
4. **Monitoraggio attivo con HIDS (OSSEC)**
   Il progetto integra un sistema di monitoraggio basato su HIDS, configurato per rilevare modifiche ai file critici e anomalie nei log del demone IPFS. Questo obiettivo dimostra come sia possibile aggiungere un livello di sicurezza operativo, in grado di intercettare tentativi di manomissione o errori applicativi, e fornisce strumenti pratici per comprendere l'importanza del logging e degli alert in un ambiente sicuro.
5. **Ottimizzazione e flessibilità tramite AOT e API**
   Infine, il progetto prevede l'uso della compilazione Ahead-of-Time (AOT) in WAMR per ridurre i tempi di esecuzione dei moduli WASM, oltre all'implementazione di un'API /sum con attestazione simulata a monte della logica. Questo permette di esplorare meccanismi di gating, controllo dell'esecuzione e modularità del codice in un contesto sicuro.

In sintesi, gli obiettivi del progetto non si limitano alla semplice implementazione tecnica, ma mirano a fornire una **visione completa e pratica** di come diverse tecnologie possano essere integrate per garantire sicurezza, affidabilità e verificabilità dei dati, combinando isolamento del codice, storage distribuito e monitoraggio attivo in un unico ambiente sperimentale.

---

3) Prerequisiti e Scelte Base

Al fine del corretto sviluppo ed esecuzione del progetto, abbiamo definito prerequisiti minimi e scelte tecnologiche fondamentali che garantiscono un ambiente sicuro, replicabile e

facilmente gestibile su quasi tutti i personal computer della rete domestica, senza la necessità di particolari prestazioni hardware o software.

Prerequisiti

- **Docker** installato sull'host (Windows).
    - L'uso di **WSL2** è opzionale ma consigliato per facilitare compatibilità e prestazioni.
- **Immagine di base del container** (fornita dal materiale didattico e successivamente modificata):
    - Ubuntu come sistema operativo.
    - Intel SGX SDK in modalità simulata (SIM) per eseguire enclave senza hardware dedicato.
    - WAMR (iwasm) e WASI SDK per eseguire moduli WebAssembly in modo isolato.
    - IPFS (Kubo) per storage content-addressed e verifica dei risultati.
    - OSSEC HIDS in modalità local per monitoraggio dei file critici e log applicativi.

Scelte tecnologiche e motivazioni

- **SGX in modalità SIM:**
    - Permette lo sviluppo e il test di enclave senza hardware reale.
    - Il codice sviluppato può essere successivamente migrato su piattaforme con SGX attivo senza modifiche.
- **WAMR (WebAssembly Micro Runtime):**
    - Runtime leggero e versatile, supporta sia modalità interpretata sia AOT, ottimizzando le performance dei moduli WASM.
    - Integrabile con SGX per garantire esecuzione isolata del codice.
- **WASI (WebAssembly System Interface):**
    - I/O sandboxed: ogni modulo WASM accede solo a directory preapertamente definite, limitando il rischio di accessi non autorizzati al filesystem.
- **IPFS (InterPlanetary File System):**
    - Fornisce content addressing tramite CID univoco, garantendo integrità e verificabilità indipendente dei dati generati.
- **OSSEC HIDS (Host Intrusion Detection System):**
    - Configurato in modalità local per semplicità e controllo centralizzato.
    - Monitora i file critici e analizza i log del demone IPFS, rilevando tampering e anomalie in tempo reale.

4) Requisiti e Tecnologie Utilizzate

All'interno del nostro progetto abbiamo utilizzato diverse tecnologie per realizzare un ambiente sicuro e verificabile, in grado di eseguire codice isolato, generare risultati affidabili e monitorare l'integrità del sistema. Di seguito vengono descritti i principali requisiti funzionali e le tecnologie impiegate, evidenziandone l'importanza.

1. TEE / SGX (Trusted Execution Environment / Intel SGX)

Cos'è:
Una TEE è un ambiente isolato all'interno della CPU, dove codice e dati restano protetti dal resto del sistema operativo.

Nel progetto:

- L'enclave SGX è in modalità simulata.
- Garantisce che il codice WASM non possa essere letto o modificato da altri processi e che i dati restino confidenziali e integri.

Rilevanza per la sicurezza:

- Dimostra la costruzione di un trusted computing flow: esecuzione sicura, attestazione e output verificabile.
- Anche in modalità simulata, evidenzia le dinamiche fondamentali di protezione offerte dalle TEE.

---

2. WebAssembly (WASM)

Cos'è:
WebAssembly è un formato binario portabile pensato per eseguire codice in modo veloce e sicuro, inizialmente per browser, oggi utilizzato anche lato server o in contesti embedded.

Rilevanza per la sicurezza:

- Fornisce un ambiente sandboxato: il codice non può accedere direttamente alla memoria o alle risorse esterne, riducendo il rischio di exploit comuni.
- È portabile: lo stesso modulo può essere eseguito su sistemi diversi senza ricompilazione.
- Permette di eseguire logiche non affidabili in un contesto controllato.

---

3. WAMR (WebAssembly Micro Runtime)

Cos'è:
Un runtime leggero che permette di eseguire moduli WebAssembly fuori dal browser, scritto in C/C++.

Nel progetto:

- Gira dentro l'enclave SGX simulata.
- Supporta due modalità:
  - **Interprete:** più sicuro, ma più lento.
  - **AOT (Ahead-of-Time):** compila il modulo WASM in codice nativo prima di eseguirlo, ottimizzando le performance.

---

4. Filesystem (scrittura dei risultati)

Nel progetto:

- Il modulo WASM scrive un risultato (ad esempio la somma di due numeri) su un file del filesystem tramite WASI.

Rilevanza per la sicurezza:

- La scrittura è monitorata da OSSEC, per rilevare modifiche non autorizzate.
- Il file costituisce l'output da sigillare e pubblicare su IPFS.

---

5. IPFS (InterPlanetary File System)

Cos'è:
Un file system distribuito basato su contenuti, in cui ogni file ha un identificatore unico (CID) calcolato come hash del contenuto.

Nel progetto:

- Il file scritto dal server viene pubblicato su IPFS.
- Il CID agisce come "firma" del contenuto: qualsiasi modifica produce un hash diverso.

Vantaggi per la sicurezza:

- Garantisce integrità dei dati.
- Permette verificabilità pubblica: chiunque può confrontare il CID del file.

6. OSSEC (Host Intrusion Detection System)

Cos'è:
Un HIDS open source che analizza file di log, file di sistema e processi per rilevare attività sospette.

Nel progetto:

- Installato sull'host che esegue l'enclave.
- Monitora il file prodotto dal modulo WASM e i log del demone IPFS.

Rilevanza per la sicurezza:

- Rileva tampering, anomalie e accessi non autorizzati.
- Simula un meccanismo di monitoraggio attivo dell'integrità del sistema.

---

7. AOT (Ahead-of-Time Compilation)

Cos'è:
Modalità di esecuzione in cui il codice WASM viene compilato in codice nativo prima dell'esecuzione.

Nel progetto:

- Utilizzata per ottimizzare le performance del modulo /sum dentro WAMR.

Rilevanza per la sicurezza:

- Migliora le prestazioni senza compromettere l'isolamento.
- La compilazione avviene in modo controllato e il binario viene eseguito in sicurezza dentro WAMR.

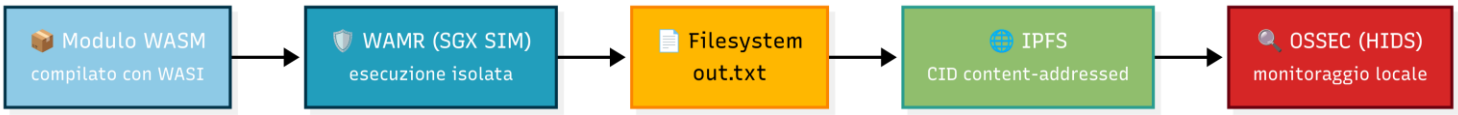
5) Architettura di Alto Livello

L'architettura del nostro progetto è stata progettata per dimostrare in modo chiaro e lineare l'integrazione tra **WAMR/WASI**, **filesystem**, **IPFS** e **OSSEC**, utilizzando **Intel SGX in modalità simulata**.

L'obiettivo non era costruire un sistema distribuito con client/API e attestazione remota, ma mostrare un flusso essenziale e riproducibile che parte dall'esecuzione di moduli WebAssembly e termina con il monitoraggio dei file e dei log.

Per rappresentare in modo chiaro e immediato l'architettura di alto livello ci siamo avvalsi di Mermaid, uno strumento che permette di creare diagrammi di flusso leggibili e facilmente integrabili in documenti Markdown o pagine web. Il diagramma seguente mostra come interagiscono tra loro i principali componenti.

Il diagramma concettuale può essere riassunto come segue:



Come funziona

Funzionamento:

- **Modulo WASM (azzurro chiaro):** programma compilato con WASI che contiene la logica da eseguire (es. scrittura su file).
- **WAMR in SGX SIM (azzurro scuro):** esegue il modulo WASM in modalità simulata, replicando l'isolamento di un Trusted Execution Environment.
- **Filesystem out.txt (giallo oro):** memorizza l'output prodotto dall'esecuzione del modulo, fornendo un'evidenza locale.
- **IPFS (verde/rosa):** pubblica out.txt in rete distribuita generando un CID content-addressed; lo stesso contenuto genera sempre lo stesso CID, garantendo integrità e verificabilità.
- **OSSEC HIDS (rosso):** monitora in tempo reale sia out.txt che i log del demone IPFS per rilevare anomalie, errori o tentativi di tampering.

6) Installazioni e Setup Iniziale

Per iniziare con lo sviluppo del nostro progetto abbiamo dovuto installare e fare i setup di tutte le componenti necessarie per la nostra enclave SGX Simulata, abbiamo iniziato attingendo dal materiale dalle cartelle (setup-0/1/2) forniteci e effettuato le dovute installazioni tramite i Readme o guide online.

Di seguito riportiamo, in ordine logico e con spiegazioni, i passaggi eseguiti.

6.1. Creazione del container Docker

- **Avvio container base**:

```
docker run -it --name sgx-dev ubuntu:20.04 /bin/bash
```

- **Installazione pacchetti di base**:

```
apt-get update && apt-get install -y \
  git build-essential autoconf automake libtool pkg-config \
  ca-certificates wget curl cmake python3 tar libssl-dev
```

- Docker ci ha permesso di lavorare in un ambiente pulito, replicabile e isolato dall'host. Tutte le configurazioni successive avvengono esclusivamente dentro il container, questo implementa la portabilità e soprattutto la sicurezza.

6.2. Installazione e configurazione Intel SGX

- **Download e installazione SDK**:

```
wget https://download.01.org/intel-sgx/sgx-linux/2.19/distro/ubuntu20.04-
server/sgx_linux_x64_sdk_2.19.100.3.bin
chmod +x sgx_linux_x64_sdk_2.19.100.3.bin
./sgx_linux_x64_sdk_2.19.100.3.bin
```

- **Caricamento ambiente SDK**:

```
source /opt/intel/sgxsdk/environment
```

- **Variabili d'ambiente**:

```
export SGX_SDK=/opt/intel/sgxsdk
export PATH=$SGX_SDK/bin:$PATH
export LD_LIBRARY_PATH=$SGX_SDK/lib64:$LD_LIBRARY_PATH
```

- **Test in modalità simulata (SIM)**:

```
make                                                    SGX_MODE=SIM
./app
```

- L’uso della modalità *SIM* ci ha consentito di sviluppare e testare enclave senza hardware SGX.

6.3. Configurazione wolfSSL

- **Clonazione e preparazione**:

```
git clone https://github.com/wolfSSL/wolfssl.git
cd wolfssl
./autogen.sh && ./configure && ./config.status
```

- **Compilazione SGX (libreria statica)**:

```
cd IDE/LINUX-SGX
make -f sgx_t_static.mk clean || true
make -B -f sgx_t_static.mk HAVE_WOLFSSL_TEST=1 HAVE_WOLFSSL_BENCHMARK=1 V=1
```

- **Verifica simboli**:

```
nm libwolfssl.sgx.static.lib.a | grep -E
'wolfSSL_new|wolfSSL_CTX_new|wolfcrypt_test' | head
```

- wolfSSL, abbiamo deciso di integrare woflSSL in modo da rendere completa la nostra enclave SGX e fornire quindi un canale sicuro TLS per sviluppi futuri, il nostro progetto non prevede una comunicazione cliente ↔ server, ma il nostro obiettivo è una pipeline locale. Resta pertanto un’estensione possibile in futuro, per implementare **remote attestation reale e canali TLS sicuri** tra client e server.

6.4. Configurazione WAMR (WebAssembly Micro Runtime) con WASI

- **Clonazione repository**:

```
git clone https://github.com/bytecodealliance/wasm-micro-runtime.git
cd wasm-micro-runtime/product-mini/platforms/linux-sgx
```

- **Compilazione con supporto SGX e WASI**:

```
cmake -DWAMR_BUILD_LIBC_WASI=1 .
make
```

- **Test modulo WASM**:

```
clang --target=wasm32-wasi -O2 -o sum.wasm sum.c
iwasm --dir=. sum.wasm 3 5 out.txt
```

- WAMR con WASI ci ha permesso di eseguire codice WebAssembly in enclave e di interagire col filesystem in modo sicuro e controllato.

6.5. Configurazione IPFS (InterPlanetary File System)

- **Installazione**:

```
wget https://dist.ipfs.tech/kubo/v0.18.1/kubo_v0.18.1_linux-amd64.tar.gz
tar -xvzf kubo_v0.18.1_linux-amd64.tar.gz
cd kubo
./install.sh
```

- **Inizializzazione e avvio nodo**:

```
ipfs init
mkdir -p /var/log
ipfs daemon 2>&1 | tee /var/log/ipfs.log
```

- IPFS è stato scelto per archiviare in modo distribuito e content-addressed i file prodotti dall'enclave. I log generati vengono reindirizzati per il monitoraggio tramite HIDS.

6.6. Configurazione OSSEC (HIDS)

- **Installazione**:

```
wget -O - https://github.com/ossec/ossec-hids/archive/3.7.0.tar.gz | tar zx
cd ossec-hids-3.7.0
./install.sh
```

- **Configurazione monitoraggio log IPFS** (*ossec.conf*):

```
<localfile>
  <log_format>syslog</log_format>
  <location>/var/log/ipfs.log</location>
</localfile>
```

- In questo modo lo abbiamo configurato in modalità locale. Essendo il nostro un progetto single container (un solo host che monitora sé stesso), non avevamo bisogno di un'architettura manager/agent perché non abbiamo distribuito i servizi su più container (ad esempio un container per WASM, uno per IPFS, uno per OSSEC, ecc.), ma abbiamo integrato tutto in un unico container. In questo scenario è sufficiente la modalità *local*, che consente di monitorare direttamente i log interni senza necessità di raccogliere traffico o eventi da più host.

- **Avvio HIDS**:

```
/var/ossec/bin/ossec-control start
```

- OSSEC è stato configurato in modalità locale per monitorare i log di IPFS e rilevare anomalie, rafforzando la sicurezza dell'ambiente.

6.7. Salvataggio delle immagini Docker

- **Snapshot intermedi**:

```
docker commit <container_id> sgx-enclave:latest
docker commit <container_id> sgx-enclave-wamr:latest
docker commit <container_id> sgx-enclave-wolfssl:latest
```

- I commit Docker ci sono stati di grande aiuto per tutte le installazioni, il processo è stato molto lungo e pieno di errori. Ci siamo spesso ritrovati a dover eliminare il container appena realizzato perché avevamo modificato erroneamente configurazioni importanti; avere sempre un'immagine del container come backup, e crearne di nuove ad ogni progresso, ci ha permesso di lavorare in maniera pulita e profittevole.

---

7) Implementazione

Per rendere il progetto **replicabile e utilizzabile da altri utenti**, abbiamo deciso di strutturare questa sezione in modo da funzionare sia come **parte della relazione tecnica** sia come un **mini-README operativo**.

In questo modo, chiunque voglia eseguire il progetto potrà seguire passo-passo le istruzioni da noi fornite, senza dover consultare altre fonti o documentazioni esterne.

7.1 Setup del Container (rete, volumi, IPFS)

Questa sezione descrive come preparare l'ambiente di esecuzione del progetto, configurare il container con Docker, avviare IPFS e verificare il corretto funzionamento della rete e dei volumi.

7.1.1 Avvio del container

Il comando consigliato per avviare il container su host Windows/PowerShell è il seguente:

```
docker run -it --name sgx-ipfs-app ^
  -p 4001:4001 -p 4001:4001/udp ^
  -p 5001:5001 -p 8080:8080 ^
  -v "${PWD}:/usr/src/app" ^
  sgx-enclave-ipfs:latest /bin/bash
```

Motivazioni delle opzioni

- `docker run -it --name sgx-ipfs-app`:
  - *Motivazione:* avviamo un container interattivo (`-it`) per poter eseguire comandi manuali e visualizzare log; il nome `sgx-ipfs-app` rende semplice riferirsi al container nelle operazioni successive.
- `-p 4001:4001 -p 4001:4001/udp`:
  - *Motivazione:* espone la porta **swarm** di IPFS sia su TCP sia su UDP (con QUIC). Serve per consentire al nodo di parlare con altri peer P2P e scambiare blocchi. Senza questo mapping la connettività P2P sarebbe limitata al container.
- `-p 5001:5001`:
  - *Motivazione:* espone la **API HTTP** di Kubo (IPFS). L'API è usata per comandi programmatici (es. `ipfs add`, `ipfs cat` via HTTP). Nota: molte route API richiedono POST.
- `-p 8080:8080`:
  - *Motivazione:* espone il **Gateway HTTP** che permette di accedere ai contenuti IPFS tramite GET (es. `http://localhost:8080/ipfs/<CID>`), utile per verifica via browser o curl.
- `-v "${PWD}:/usr/src/app"`:
  - *Motivazione:* monta la directory di lavoro dell'host nel container: garantisce persistenza dei file creati (es. `out.txt`, `prova.txt`) e permette di editare file dall'host senza ricostruire l'immagine.

- *sgx-enclave-ipfs:latest /bin/bash*:
    - *Motivazione:* la nostra immagine di base contiene SGX in modalità SIM, WAMR, IPFS e OSSEC; avviare */bin/bash* facilita operazioni manuali di setup e debug.

7.1.2 Configurazione di IPFS (Terminale A)

Nel primo terminale (Terminale A) dentro il container avviato eseguire i comandi seguenti per inizializzare e configurare il demone IPFS:

```
# Inizializzare IPFS solo se non esiste
[ -d /root/.ipfs ] || ipfs init

# Configurare le porte IPFS per API, gateway e swarm
ipfs config Addresses.API /ip4/0.0.0.0/tcp/5001
ipfs config Addresses.Gateway /ip4/0.0.0.0/tcp/8080
ipfs config Addresses.Swarm --json '[
  "/ip4/0.0.0.0/tcp/4001",
  "/ip4/0.0.0.0/udp/4001/quic-v1"
]'

# Se necessario, caricare l'ambiente SGX SDK
source /opt/intel/sgxsdk/environment

# Avviare il demone IPFS in foreground (utile per evidenze nei log)
ipfs daemon
```

Motivazioni delle opzioni

- *[-d /root/.ipfs] || ipfs init*:
    - *Motivazione:* *ipfs init* crea la repo locale (*/root/.ipfs*) con chiavi, configurazione e datastore; il controllo evita di sovrascrivere una repo già presente.
- *ipfs config Addresses.API /ip4/0.0.0.0/tcp/5001* e *ipfs config Addresses.Gateway /ip4/0.0.0.0/tcp/8080*:
    - *Motivazione:* impostando *0.0.0.0* forziamo il demone ad ascoltare su tutte le interfacce del container, necessario per far funzionare il port mapping verso l'host (altrimenti l'API/gateway sarebbero raggiungibili solo da *localhost* interno al container).
- *ipfs config Addresses.Swarm --json '[...]'*:
    - *Motivazione:* abilita l'ascolto P2P su TCP e su UDP QUIC (più efficiente). Questo permette al nodo di connettersi ai peer della rete IPFS; fondamentale per recuperare contenuti non presenti localmente.
- *source /opt/intel/sgxsdk/environment*:
    - *Motivazione:* rende disponibili eventuali tool e librerie SGX presenti nell'immagine (utile se, ad esempio, si compila/avvia qualcosa che interagisce con l'SDK durante i test).
- *ipfs daemon* (foreground):

- *Motivazione:* tenere il demone in foreground aiuta a debug/registrare l'output direttamente sul terminale (utile durante test e per raccogliere evidenze per la relazione). Se si preferisce usare il demone in background, si può lanciare con `&` o usare un sistema di gestione dei servizi.

7.1.3 Operazioni IPFS (Terminale B)

Aprire un nuovo terminale e collegarsi al container per effettuare operazioni di test e verifica:

```
docker exec -it sgx-hids-test /bin/bash

# Creazione di un file di prova
echo "Hello from IPFS in Docker SGX" > prova.txt

# Aggiunta su IPFS e recupero CID
CID=$(ipfs add -Q prova.txt); echo "$CID"

# Lettura del contenuto tramite CID
ipfs cat "$CID"

# Informazioni sul blocco
ipfs block stat "$CID"

# Verifica dei peers connessi nella rete P2P
ipfs swarm peers | wc -l
```

Motivazioni delle opzioni

- `docker exec -it ...`:
  - *Motivazione:* permette di aprire una shell nel container già in esecuzione per eseguire i test senza riavviare il demone.
- `echo "..." > prova.txt`:
  - *Motivazione:* crea un payload semplice da usare per verificare l'intera pipeline (scrittura, add, cat, ecc.).
- `CID=$(ipfs add -Q prova.txt)`:
  - *Motivazione:* `ipfs add` chunkizza e memorizza il file nel datastore locale; l'opzione `-Q` stampa solo il CID, comodo per scripting.
- `ipfs cat "$CID"` e `ipfs block stat "$CID"`:
  - *Motivazione:* verificano che il contenuto sia leggibile e che i blocchi relativi siano presenti/valide.
- `ipfs swarm peers | wc -l`:
  - *Motivazione:* fornisce un'idea se il nodo è connesso a peers esterni (utile per test P2P), utile per distinguere test offline (solo datastore locale) da test in rete.

7.1.4 Verifica HTTP

Verificare accesso API e Gateway tramite chiamate HTTP:

```
# API IPFS (POST richiesto)
curl -s -X POST http://127.0.0.1:5001/api/v0/version
curl -s -X POST http://127.0.0.1:5001/api/v0/id
curl -s -X POST -F file=@prova.txt 'http://127.0.0.1:5001/api/v0/add?pin=true'

# Gateway IPFS (GET)
curl "http://127.0.0.1:8080/ipfs/$CID"
```

Motivazioni delle opzioni

- `POST` per le API `/api/v0/*`:
  - *Motivazione:* Kubo richiede POST per molte API (es. `add`, `cat` via API HTTP). Utilizzare GET su queste rotte produce `405 Method Not Allowed`.
- `pin=true` nell'endpoint `add`:
  - *Motivazione:* chiede al demone di mantenere (pin) il contenuto nel datastore locale, evitando che venga garbage-collected; utile per test ripetuti.
- Gateway `http://127.0.0.1:8080/ipfs/<CID>`:
  - *Motivazione:* consente un accesso semplice via browser o curl per recuperare il contenuto pubblicato, comodo per dimostrazioni e screenshot.

7.1.5 Note operative finali

- Usare `0.0.0.0` nelle config Addresses permette al port mapping Docker di esporre i servizi fuori dal container. Senza questa impostazione, le porte sarebbero legate al loopback interno.
- Tenere il demone IPFS in foreground durante la fase di configurazione aiuta a raccogliere evidenze e messaggi utili nella relazione (log di bootstrap, connessioni peer, errori).
- Se si incappa in errori del tipo `lock /root/.ipfs/repo.lock`, verificare che non esistano demoni IPFS multipli in esecuzione e arrestare quello in più (`kill` o `ipfs shutdown`).

7.2 WAMR + WASI: Scrittura su Filesystem e Pubblicazione su IPFS

Questa sezione documenta come il nostro progetto integra **WebAssembly (WASM)**, il runtime **WAMR** e l'interfaccia **WASI** per scrivere in modo sicuro su filesystem, e come i risultati vengano pubblicati su **IPFS**.

7.2.1 Modulo WASI minimale (*write.c*)

Il seguente programma C dimostra come scrivere su filesystem da un modulo WASM eseguito con WAMR:

```c
#include <stdio.h>
#include <string.h>
int main(int argc, char** argv) {
  const char *out = "out.txt";
  const char *msg = (argc>1)? argv[1] : "Hello from WASM inside SGX (SIM) via
WAMR!\n";
  FILE *f = fopen(out, "w");
  if (!f) { perror("fopen"); return 1; }
  fwrite(msg, 1, strlen(msg), f);
  fclose(f);
  printf("Wrote %zu bytes to %s\n", strlen(msg), out);
  return 0;
}
```

P.S. Il codice va ovviamente inserito in un programma che chiameremo write.c

nano /usr/src/app/write.c

Compilazione con WASI SDK

/opt/wasi-sdk/bin/clang --target=wasm32-wasi -O2 -o /usr/src/app/write.wasm /usr/src/app/write.c

Motivo: *--target=wasm32-wasi* garantisce la compatibilità con l'interfaccia WASI, che abilita l'I/O sicuro per filesystem.

Esecuzione con WAMR (SGX SIM)

```
cd /usr/src/app
# Pre-open della directory corrente (WASI richiede dir pre-aperta)
iwasm --dir=. ./write.wasm "File scritto da WAMR in SGX SIM.\n"
ls -l out.txt && head -n1 out.txt
```

Nota: l'opzione *--dir=.* è fondamentale. In assenza di directory preaperta, l'I/O fallisce per design di sicurezza WASI.

7.2.2 Pubblicazione su IPFS

Dopo aver scritto il file, lo pubblichiamo su IPFS:

```
CID=$(ipfs add -Q /usr/src/app/out.txt); echo "CID: $CID"

# Verifica via Gateway HTTP
curl "http://127.0.0.1:8080/ipfs/$CID"

# Verifica via API IPFS (POST)
curl -s -X POST "http://127.0.0.1:5001/api/v0/cat?arg=$CID" | head -n 3
```

Motivazioni: - *ipfs add -Q* genera solo il CID, che è l'identificativo content-addressed. - La doppia verifica (gateway + API) garantisce l'accessibilità sia lato GET sia POST.

7.2.3 Test di Determinismo

Un aspetto chiave di IPFS è il **determinismo del CID**: lo stesso contenuto produce sempre lo stesso hash.

```
PAYLOAD=$'File scritto da WAMR in SGX SIM.\n'

# stesso input => stesso CID
iwasm --dir=. ./write.wasm "$PAYLOAD"; CID1=$(ipfs add -Q out.txt)
iwasm --dir=. ./write.wasm "$PAYLOAD"; CID2=$(ipfs add -Q out.txt)
[ "$CID1" = "$CID2" ] && echo "OK: deterministico"

# input diverso => CID diverso
iwasm --dir=. ./write.wasm $'Payload diverso\n'; CID3=$(ipfs add -Q out.txt)
[ "$CID3" != "$CID1" ] && echo "OK: content addressing"
```

Interpretazione dei test: - Se *CID1 = CID2*, abbiamo dimostrato che il meccanismo è deterministico. - Se *CID3 != CID1*, è confermata la natura content-addressed di IPFS.

Abbiamo appena visto come: - WAMR + WASI consentono al modulo WASM di scrivere su filesystem in modo sicuro (sandboxed). - IPFS garantisce l'integrità e la verificabilità pubblica del risultato tramite CID. - I test di determinismo rafforzano il valore di IPFS come meccanismo di verifica indipendente.

La combinazione di **WASI**, **WAMR** e **IPFS** rappresenta quindi un flusso completo: esecuzione isolata → scrittura sicura → pubblicazione verificabile.

7.3 AOT (WAMR) — Motivazioni e Procedura

Questa sezione descrive perché usare AOT (Ahead-Of-Time) con WAMR nel contesto dell'enclave SGX (SIM) e fornisce una procedura dettagliata per compilare il compilatore `wamrc`, generare immagini AOT e misurare (in modo semplice) il beneficio prestazionale.

7.3.1 Perché AOT

- **Riduce latenza di startup:** in molti scenari l'overhead iniziale di interpretazione o JIT può essere significativo; AOT produce un file eseguibile nativo che parte più velocemente.
- **Migliora le performance runtime:** l'esecuzione del codice nativo è più veloce rispetto all'interpretazione.
- **Vantaggioso in enclave:** le operazioni di JIT/compilazione dinamica dentro TEE possono essere problematiche (permessi, sicurezza); AOT evita questo problema compilando esternamente e eseguendo un binario stabile dentro l'enclave.

  Nota: in modalità SIM non cambiano le garanzie di sicurezza di SGX, ma AOT resta utile per valutare prestazioni e comportamento in condizioni realistiche.

7.3.2 Procedura completa

1) Installare dipendenze di build

```
apt-get update && apt-get install -y \
  build-essential cmake git python3 wget ca-certificates pkg-config \
  libssl-dev curl
```

Motivazione: sono necessari strumenti di compilazione, CMake e librerie per costruire `wamrc` (compiler WAMR) e le sue dipendenze.

---

2) Clonare il repository e costruire wamrc

```
cd /root
git clone https://github.com/bytecodealliance/wasm-micro-runtime.git
cd wasm-micro-runtime/wamr-compiler
./build_llvm.sh
mkdir -p build && cd build
cmake ..
make -j"$(nproc)"
cp wamrc /usr/local/bin/
wamrc --version
```

Motivazione passo-passo: - `build_llvm.sh` prepara una toolchain LLVM compatibile con WAMR.
- `cmake && make` compila il compilatore AOT.
- `cp wamrc /usr/local/bin/` rende `wamrc` disponibile in PATH per i passi successivi.

---

3) Preparare il wasm da compilare in AOT

Verificare che il modulo WASM sia stato generato (es. `write.wasm` o `sum.wasm`). Se non esiste, ricompilarlo con WASI SDK:

```
/opt/wasi-sdk/bin/clang --target=wasm32-wasi -O2 -o /usr/src/app/write.wasm
/usr/src/app/write.c
```

Motivazione: il file `.wasm` è il punto di partenza per AOT.

---

4) Compilare AOT per SGX

```
# esempio per write.wasm
wamrc -sgx -o /usr/src/app/write.aot /usr/src/app/write.wasm
```

Motivazione: l'opzione `-sgx` applica patch/flag specifici per generare artefatti compatibili con l'ambiente SGX (anche se in SIM). Il file `.aot` è più rapido da caricare in iwasm.

---

5) Eseguire l'AOT dentro WAMR (SGX SIM)

```
cd /usr/src/app
# esecuzione del modulo AOT (WASI pre-open dir)
iwasm --dir=/usr/src/app /usr/src/app/write.aot "Esecuzione AOT in SGX SIM.\n"
# pubblica su IPFS e ottieni CID
CID_AOT=$(ipfs add -Q /usr/src/app/out.txt); echo "CID_AOT: $CID_AOT"
```

Motivazione: con AOT la latenza di esecuzione iniziale è ridotta; l'output viene poi pubblicato su IPFS come nella pipeline standard.

---

6) Mini- benchmark comparativo (indicativo)

```
cd /usr/src/app
# interpretato
(time iwasm --dir=. ./write.wasm "payload\n" >/dev/null)
# AOT
(time iwasm --dir=. ./write.aot  "payload\n" >/dev/null)
```

Interpretazione: confrontare i tempi misurati fornisce una stima empirica di quanto AOT migliori la latenza. I risultati dipendono da dimensione del modulo e complessità del codice.

7.3.3 Considerazioni finali su AOT

- **Sicurezza:** AOT non indebolisce l'isolamento WASI o SGX; si limita a trasformare il modulo in un artefatto più efficiente.
- **Deploy:** l'artifact AOT può essere salvato nel repository o rigenerato durante la pipeline di CI/CD.
- **Limiti:** la generazione AOT richiede toolchain pesanti (compilazione LLVM) e quindi è più adatta a fasi di build offline piuttosto che on-device.

7.3bis AOT abilitato ma NON UTILIZZATO

Per le motivazioni indicate prima abbiamo installato e attivato il supporto per **AOT (Ahead-Of-Time compilation)** in WAMR, in modo che tutto il setup sia già pronto e configurato per gli utilizzatori futuri. La presenza del comando `wamrc` e la possibilità di generare file `.aot` da moduli `.wasm` ci hanno confermato che il compilatore AOT era funzionante.

- **Verifica AOT**:

  ```
  which wamrc
  wamrc --version
  wamrc -o hello.aot hello.wasm
  ```

  → output: *Compile success, file hello.aot was generated.*

Tuttavia, al momento dell'esecuzione con `iwasm`, il file `.aot` non è stato riconosciuto e il runtime ha restituito l'errore:

```
WASM module load failed: magic header not detected
```

Chiarimenti

Questo avviene perché la versione di `iwasm` che abbiamo utilizzato non era stata compilata con il supporto AOT abilitato. Abbiamo effettuato diversi tentativi i cui risultati però non rispettavano le nostre aspettative, pertanto abbiamo scelto di **non ricompilare l'intero runtime WAMR** per mantenere stabile l'ambiente già predisposto e completare il progetto nei tempi previsti. **L'attivazione di AOT** rimane comunque un passo utile per sviluppi futuri, in cui potremo sfruttare la maggiore efficienza dell'esecuzione Ahead-Of-Time senza modificare la struttura del progetto attuale.

7.4 HIDS (OSSEC) — Setup, Regole e Test

Questa sezione descrive in modo dettagliato e operativo come integrare OSSEC HIDS nel progetto, spiegando le scelte progettuali, di seguito mostriamo gli estratti di configurazione chiave e le regole locali, e forniamo i comandi di avvio e i test da eseguire.

Al pari delle sezioni precedenti, questa vale sia come relazione tecnica che come README operativo del repository.

7.4.1 Scelte progettuali (versione corretta)

- **Modalità local (nessun manager/agent):**
    - **Idea:** semplificare l'architettura per la consegna e ridurre la superficie d'attacco. In modalità *local* tutto il controllo rimane sul singolo host/container che esegue WAMR, IPFS e OSSEC.
    - **Motivazione:** evita la complessità di gestire un manager/agent e rende più immediato il testing in ambiente universitario.
- **Target di monitoraggio:**
    - /usr/src/app/out.txt — file di output critico generato dai **moduli WASM eseguiti con WAMR/WASI**; monitoraggio in tempo reale per rilevare eventuali manomissioni.
    - /var/log/ipfs.log — log del demone IPFS (stdout/stderr), utile a intercettare errori applicativi, comandi non validi o problemi di rete.
- **Funzionalità disattivate:**
    - email_notification disabilitata → per evitare la dipendenza da un MTA all'interno del container.
    - rootcheck disabilitato → semplifica l'esecuzione e riduce rumore di log non essenziale in fase di sviluppo.

- active-response disabilitato → in fase di consegna non vogliamo blocchi o reazioni automatiche; la funzionalità potrà essere riattivata in scenari di produzione.

7.4.2 Configurazione chiave (/var/ossec/etc/ossec.conf) — estratti

Il file di configurazione principale contiene i blocchi essenziali per il nostro uso. Di seguito un estratto commentato.

```
<ossec_config>
  <global>
    <email_notification>no</email_notification>
  </global>

  <rootcheck>
    <disabled>yes</disabled>
  </rootcheck>

  <active-response>
    <disabled>yes</disabled>
  </active-response>

  <alerts>
    <log_alert_level>1</log_alert_level>
  </alerts>

  <!-- Log IPFS -->
  <localfile>
    <log_format>syslog</log_format>
    <location>/var/log/ipfs.log</location>
  </localfile>

  <!-- SYSCheck (unico blocco) -->
  <syscheck>
    <frequency>60</frequency>
    <scan_on_start>yes</scan_on_start>
    <directories realtime="yes" check_all="yes">/usr/src/app/out.txt</directories>
    <directories check_all="yes">/etc,/usr/bin,/usr/sbin</directories>
    <directories check_all="yes">/bin,/sbin,/boot</directories>
  </syscheck>
</ossec_config>
```

Spiegazione delle motivazioni

- `<email_notification>no</email_notification>`
  - *Perché:* evita la necessità di configurare servizi di posta nel container; utile in ambiente di test.
- `<rootcheck><disabled>yes</disabled></rootcheck>`

    - *Perché:* riduce rumore e tempo di scansione; il focus del progetto è syscheck e log dell'applicazione.
- `<active-response><disabled>yes</disabled></active-response>`
    - *Perché:* in fase di sviluppo non vogliamo automatismi che modificano lo stato del container o interrompono processi.
- `<alerts><log_alert_level>1</log_alert_level></alerts>`
    - *Perché:* imposta il livello minimo di logging per gli alert che vogliamo catturare e conservare.
- `<localfile> /var/log/ipfs.log </localfile>`
    - *Perché:* OSSEC analizzerà questo file per identificare pattern di errore di IPFS: Unknown Command, websocket errors, ecc.
- `<syscheck>` e i suoi sottoelementi
    - `frequency=60`: scansione periodica ogni 60s per garantire controlli anche nei casi in cui inotify/realtime non sia perfettamente affidabile.
    - `scan_on_start=yes`: esegue un controllo iniziale all'avvio (utile per dimostrazioni).
    - `directories realtime="yes" check_all="yes">/usr/src/app/out.txt</directories>`: abilita monitoraggio in tempo reale su `out.txt` (tamper immediato).
    - Inclusion delle directory di sistema: consente di rilevare modifiche critiche al sistema, ma aumenta il rumore — utile per completezza in ambiente controllato.

7.4.2.1 Regole locali (/var/ossec/etc/rules/local_rules.xml)

Esempio di regole locali inserite per elevare eventi importanti nella relazione:

```
<group name="local,custom">
  <!-- Eleva modifica del file protetto a critico -->
  <rule id="100200" level="10">
    <if_sid>550</if_sid>
    <if_sid>551</if_sid>
    <if_sid>552</if_sid>
    <match>/usr/src/app/out.txt</match>
    <description>Critical: protected result file modified</description>
  </rule>

  <!-- IPFS: Unknown Command -->
  <rule id="100101" level="10">
    <if_sid>1002</if_sid>
    <match>Unknown Command</match>
    <description>IPFS unknown command (elevated)</description>
  </rule>

  <!-- IPFS: websocket close error -->
  <rule id="100102" level="8">
    <if_sid>1002</if_sid>
    <match>websocket: failed to close network connection</match>
```

```
    <description>IPFS websocket close error</description>
  </rule>
</group>
```

Spiegazione delle regole

- **Regola 100200**
  - *Scopo:* elevare a livello critico (10) la modifica di `/usr/src/app/out.txt` ma solo se gli event id di syscheck 550/551/552 sono presenti — questo evita falsi positivi e richiede correlazione. In pratica, non basta che un log menzioni out.txt: serve anche che OSSEC abbia segnalato l'evento come vero e proprio **cambio di integrità**.
  - Possiamo quindi notare come abbiamo una detection accurata e correlazione tra eventi syscheck e regole locali.
- **Regole 100101 / 100102 (IPFS)**
  - *Scopo:* elevare errori significativi del demone IPFS a livello alto per facilitare l'analisi post-mortem e la presentazione delle evidenze.
  - *Nota tecnica:* se i messaggi di IPFS cambiano formato, è possibile passare a `<regex>` più permissivi.

7.4.3 Avvio & Validazione

```
# Avvia OSSEC
/var/ossec/bin/ossec-control start
/var/ossec/bin/ossec-control status

# Avvia IPFS e dirotta stdout/stderr su /var/log/ipfs.log (lettura da OSSEC)
(ipfs daemon 2>&1 | tee -a /var/log/ipfs.log) &
```

Se ci sono processi attivi, avendo più volte avviato ipfs daemon in background, e ora c'è un file di lock (/root/.ipfs/repo.lock) che impedisce di aprire di nuovo la repo.

Come risolvere:

1. Chiudiamo tutti i processi IPFS attivi nel container:

pkill ipfs

2. Cancelliamo il file di lock (non tocca i dati, serve solo a sbloccare la repo):

rm -f /root/.ipfs/repo.lock

3. Riavviamo il demone, una sola volta:

ipfs daemon 2>&1 | tee -a /var/log/ipfs.log

```
# Validazione XML di configurazione
xmllint --noout /var/ossec/etc/ossec.conf

# Controllo dei processi e dei log di OSSEC
```

```
/var/ossec/bin/ossec-control status
tail -n 120 /var/ossec/logs/ossec.log | sed -n '/Started/p;/Monitoring
directory/p;/Analyzing file/p'
```

Motivazioni dettagliate

- `ossec-control start` / `status`: avvia e verifica i componenti (ossec-analysisd, ossec-syscheckd, ecc.).
- `ipfs daemon 2>&1 | tee -a /var/log/ipfs.log`: reindirizza tutti i messaggi del demone IPFS in un file che OSSEC legge per il pattern matching; essenziale per la rilevazione di errori IPFS.
- `xmllint --noout`: valida la sintassi XML di `ossec.conf` prevenendo crash di OSSEC a causa di errori di formattazione.
- Ispezione di `/var/ossec/logs/ossec.log`: garantisce che i moduli stiano monitorando i percorsi richiesti.

7.4.4 Test (procedura e risultati attesi)

7.4.4.1 Tampering del file protetto

```
# Simula manomissione del file prodotto dalla pipeline
echo "tamper $(date)" >> /usr/src/app/out.txt
sleep 3
# Visualizza gli ultimi alert
tail -n 80 /var/ossec/logs/alerts/alerts.log
```

Risultato atteso: comparsa di eventi corrispondenti a syscheck (ID 550/551/552) e della regola locale `100200` elevata a livello 10; significa che OSSEC ha rilevato il tampering e la regola locale ha innescato l'alert critico.

7.4.4.2 Anomalie IPFS

```
# Genera una riga di log anomala per IPFS
ipfs wrongcmd 2>> /var/log/ipfs.log || true
sleep 3
# Controlla gli alert
tail -n 80 /var/ossec/logs/alerts/alerts.log
```

Risultato atteso: Eventi associati a `1002` e la nostra regola `100101` (se trova la stringa `Unknown Command`) o `100102` in caso di websocket error.

7.4.5 Troubleshooting

- **`Configuration error` all'avvio di OSSEC** → probabilmente `ossec.conf` non è valido; usare `xmllint --noout` per diagnosticare.
- **Nessun alert per tampering** → verificare che OSSEC stia monitorando il path (`grep "Monitoring directory" /var/ossec/logs/ossec.log`) e che `out.txt` sia nel path esatto; verificare i permessi del file.
- **`pcre2.h` mancante durante compilazione** → installare `libpcre2-dev` e ricompilare.

- **Regole locali non lette** → assicurati che `local_rules.xml` sia nel percorso `/var/ossec/etc/rules/` e che OSSEC logghi `Reading rules file: 'local_rules.xml'`.

---

8) Vantaggi e Svantaggi della Soluzione

In questa sezione presentiamo, i **vantaggi** e gli **svantaggi** della nostra soluzione basata su **WAMR + WASI** in **SGX (modalità SIM)**, con **scrittura su filesystem** e **pubblicazione su IPFS**, monitorata da **OSSEC.**

Vantaggi

1) Riproducibilità e isolamento (Docker + SGX SIM). Abbiamo racchiuso l'intero ambiente in container; ciò assicura che i risultati siano ripetibili e che le dipendenze non contaminino l'host. L'uso di SGX in modalità simulata ci ha permesso di sperimentare il modello di enclave e la separazione trusted/untrusted anche in assenza di hardware dedicato.

2) Portabilità del codice (WASM + WASI). I moduli WASM, compilati contro WASI, sono portabili e riutilizzabili su runtime differenti. La sandbox WASI riduce la superficie d'attacco, poiché l'accesso alle risorse è esplicito (preopen delle directory) e minimo per design.

3) Integrità e verificabilità dei risultati (IPFS). La pubblicazione del file di output su IPFS produce un CID contentaddressed: lo stesso contenuto genera sempre lo stesso identificatore. Questo meccanismo abilita verifiche indipendenti, facilita il confronto tra prove e rende trasparente l'eventuale modifica dei dati.

4) Osservabilità e auditing (OSSEC in modalità local). Il monitoraggio di out.txt e dei log del demone IPFS fornisce visibilità operativa. Le regole locali ci consentono di elevare rapidamente gli eventi di interesse (tampering del file, errori IPFS), supportando l'analisi e la presentazione delle evidenze.

5) Flessibilità architetturale: la pipeline è modulare, ogni componente (SGX, WAMR, IPFS, OSSEC) può essere sostituito o aggiornato indipendentemente.

6) Preparazione a ottimizzazioni future (AOT). Abbiamo predisposto il toolchain AOT (wamrc) per WAMR: pur non avendo eseguito i binari .aot per mancanza del supporto AOT nell'iwasm impiegato, lo stack resta pronto a sfruttare l'ottimizzazione prestazionale in sviluppi successivi.

7) Trasparenza: l'uso di IPFS garantisce che i risultati siano verificabili anche da terzi, non solo **localmente.**

8) Riduzione falsi positivi: le regole OSSEC customizzate permettono di elevare solo eventi realmente critici.

Svantaggi e Limitazioni

1) Modalità simulata SGX (assenza di garanzie hardware). L'uso di SGX in SIM è funzionale allo sviluppo, ma non offre le garanzie crittografiche e di protezione della memoria dell'hardware reale. Non abbiamo implementato attestazione remota reale; eventuali risultati di sicurezza sono quindi dimostrativi.

2) Overhead e complessità dell'ambiente. L'ambiente containerizzato con più componenti (SGX SDK, WAMR, WASISDK, IPFS, OSSEC) aumenta dimensioni dell'immagine e tempi di setup. La manutenzione delle versioni richiede disciplina (documentazione delle release e dei commit).

3) Funzionalità circoscritte. Ci siamo limitati alla pipeline WASM→FS→IPFS con HIDS. Non abbiamo consegnato un'API REST con attestazione a monte né un'integrazione TLS endtoend nel percorso principale (l'integrazione wolfSSL è stata esplorata a parte, a fini dimostrativi).

4) Monitoraggio HIDS minimale. OSSEC è stato configurato in modalità local con regole essenziali. Non sono stati adottati profili di hardening avanzato, né l'invio di alert verso sistemi esterni; il valore è soprattutto didattico.

5) AOT predisposto ma non utilizzato in esecuzione. Pur avendo abilitato la toolchain AOT, non abbiamo ricompilato iwasm con supporto AOT. I benefici prestazionali, quindi, non sono stati misurati nel nostro percorso principale.

6) Persistenza e governance dei dati IPFS. La persistenza dei contenuti dipende dal pin locale; non abbiamo definito una strategia di pinning distribuito o di garbage collection controllata. In un contesto produttivo servirebbero policy e nodi di riferimento.

7) Dipendenza da configurazioni manuali → molte operazioni richiedono setup manuale (es. binding IPFS, regole OSSEC).

9) Conclusioni

Nel complesso il nostro progetto ha oggettivamente delle limitazioni ma rispetta pienamente gli obbiettivi da noi auspicati sin dalla prima stesura.

Abbiamo messo in piedi uno stack pulito e riproducibile che esegue codice WASM con WAMR/WASI in **SGX simulato**, scrive in modo controllato su filesystem e pubblica i risultati su **IPFS**, con **OSSEC** a fare da sentinella sui log. È semplice, lineare e dimostrativo: esecuzione isolata → output tracciabile → pubblicazione verificabile. Rappresenta una base solida che può essere integrata e ampliata (attestazione reale, API complete, hardening spinto), ma abbiamo **validato il percorso basico utile** e documentato ogni passaggio. Abbiamo anche preparato il terreno per l'**AOT** e testato **wolfSSL** in enclave come prova di integrazione.