Alma Mater Studiorum – Università di Bologna

DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA (DISI)

Corso di Laurea in Ingegneria Informatica

Tesi di laurea
in
Tecnologie Web T

Analisi e Valutazione delle Performance di WebAssembly System Interface

Candidato:
Francesca Conti

Relatore:
Chiar.mo Prof. Ing. Paolo Bellavista

Correlatore:
Ing. Andrea Garbugli

Anno Accademico 2022-2023
Sessione V

Oggi, 19 marzo 2024, vorrei ringraziare mio papà per avermi detto, a maggio 2023, "non ce la farai mai a laurearti a marzo".
Inoltre ci tengo a ringraziare la mia famiglia, mia sorella per avermi aiutato, la mia mamma e il mio papà per avermi supportato (e sopportato) sempre durante questo viaggio, l'Irene per essere stata al mio fianco e Luca, senza il quale non sarei qui, per avermi mostrato cosa significa essere amati.
Un ringraziamento va anche a Gino e Rosi, per avermi insegnato cosa significa amare incondizionatamente.

Abstract

WebAssembly, abbreviato in Wasm, è un linguaggio di programmazione nato nel 2015 che consente di eseguire codice originariamente scritto in diversi linguaggi di programmazione, come ad esempio C, C++, Rust e Python, all'interno di un ambiente isolato, chiamato runtime, in modo sicuro ed efficiente, originariamente pensato per essere eseguito all'interno del browser. Successivamente è stato creato WASI [1] ovvero WebAssembly System Interface, che amplia la possibilità di utilizzo di Wasm al di fuori del solo browser, permettendo in aggiunta la comunicazione con le risorse di sistema. L'obiettivo di questa tesi è valutare le performance di diversi runtime WebAssembly e delle loro implementazioni di WASI, confrontandoli in base a parametri e scenari d'uso differenti, grazie all'impiego della libreria Libsodium [3] come caso studio. Per far ciò sono stati analizzati diversi aspetti dei vari runtime Wasm riguardo alla velocità di esecuzione, velocità di lettura e scrittura file, integrazione con le socket e conformità a WASI. Dai risultati di questi test risulta come WebAssembly sia una tecnologia estremamente promettente, soprattutto in termini di portabilità e velocità di esecuzione, sebbene sia ancora in fase di sviluppo, e contenga, di conseguenza, ancora svariate criticità e difficoltà d'uso.

Indice

Elenco delle figure

Introduzione

Nel 2015 è stato introdotto WebAssembly (Wasm), un linguaggio di programmazione che offre un metodo efficiente e sicuro per eseguire codice all'interno del browser web. La sua caratteristica principale è la portabilità: infatti il codice, scritto in linguaggi diversi come C, Rust e Python, può essere compilato in Wasm e funzionare in modo omogeneo su qualsiasi browser compatibile.
Nel 2019 è stato introdotto WASI (WebAssembly System Interface), un'interfaccia standard che consente alle applicazioni Wasm di interagire con le risorse di sistema. Questo rappresenta un ampliamento nell'utilizzo di Wasm, in quanto ne aumenta l'utilizzo oltre il browser web, aprendo la strada a nuove possibilità per applicazioni più complesse, ma comunque performanti anche al di fuori del browser. Nonostante i notevoli progressi, WebAssembly è ancora in fase di sviluppo e presenta alcune criticità. Tra queste, la mancanza di librerie per diverse tipologie di applicazioni e i problemi di compatibilità tra i diversi runtime di Wasm. Non tutti i runtime implementano, infatti, tutte le funzionalità di WASI, creando incompatibilità con alcune applicazioni. Questo studio si concentra proprio sull'analisi delle performance dei diversi runtime di WebAssembly e delle loro implementazioni di WASI. L'obiettivo è valutare lo stato attuale di queste implementazioni, confrontando i runtime su parametri e scenari d'uso differenti. Per la valutazione delle performance è stata utilizzata Libsodium, una libreria crittografica di riferimento. Sono stati esaminati vari aspetti dei runtime Wasm, come la velocità di esecuzione, la gestione dei file e delle socket, nonché la conformità a WASI. Lo studio fornisce una panoramica completa delle performance dei runtime di WebAssembly e delle loro implementazioni di WASI. I risultati ottenuti offrono importanti informazioni per

gli sviluppatori che desiderano utilizzare Wasm per le loro applicazioni. Inoltre, l'analisi mette in evidenza le aree di miglioramento e le sfide future per lo sviluppo di WebAssembly e WASI.

Capitolo 1

Esecuzione Efficiente all'interno di Web Browser

Contents

1.1 Obiettivo della Tesi

Questa tesi si propone di analizzare le performance dei diversi runtime di WebAssembly e delle loro implementazioni di WebAssembly System Interface. L'obiettivo è quello di fornire una valutazione dello stato attuale di questa tecnologia emergente, confrontando i runtime su una varietà di parametri e scenari d'uso. La valutazione si basa sull'utilizzo della libreria Libsodium, una libreria che contiene svariate funzionalità crittografiche, come caso studio per la valutazione delle performance, in modo da ottenere risultati realistici e vicini a delle vere applicazioni.

L'analisi include diversi aspetti dei runtime Wasm, come la velocità di esecuzione, la gestione dei file e delle socket, nonché la conformità dell'implementazione di WebAssembly System Interface.

1.2 WebAssembly

WebAssembly, spesso abbreviato come WASM, è un formato di bytecode portatile progettato per l'esecuzione efficiente di codice all'interno di un browser web. È stato annunciato nel 2015 dal World Wide Web Consortium (W3C) con ingegneri provenienti da importanti aziende come Mozilla, Microsoft, Google e Apple e da quel momento si sta espandendo sempre di più fino ad arrivare ad essere supportato dalla maggior parte dei browser.

Rappresenta un significativo passo avanti rispetto alle tecnologie precedenti come JavaScript, in quanto si propone come strumento essenziale per superare alcune limitazioni intrinseche di quest'ultimo, pur senza sostituirlo, ma piuttosto affiancandolo, garantendo prestazioni notevolmente migliori per applicazioni web complesse.

Javascript, infatti, è stato tradizionalmente l'unico linguaggio di programmazione supportato dai browser per l'esecuzione di codice lato client. Tuttavia, un ostacolo che si può trovare è nel processo di esecuzione del codice JavaScript all'interno del browser che coinvolge diverse fasi, tra cui la tokenizzazione, la creazione dell'albero di sintassi astratta e l'interpretazione o compilazione del codice. La fase di tokenizzazione è particolarmente critica, poiché il codice sorgente JavaScript deve essere suddiviso in token, cosa che può risultare inefficiente, poiché richiede tempo e risorse per analizzare il codice sorgente e convertirlo in una serie di token. Inoltre, JavaScript è un linguaggio interpretato, il che significa che il codice viene tradotto in istruzioni macchina durante l'esecuzione, rallentando ulteriormente le prestazioni.

1.2.1 Funzionalità

WebAssembly offre una soluzione a queste limitazioni Javascript. Essendo infatti un formato di bytecode, il codice WebAssembly è già stato compilato in istruzioni macchina, eliminando la necessità di un processo di tokenizzazione e l'interpretazione durante l'esecuzione.
Da questo possiamo notare alcune sue principali funzionalità:

- **Portabilità**: Il bytecode di WebAssembly è progettato per essere eseguito su diverse architetture di CPU, rendendolo ideale per applicazioni che necessitano di funzionare su una varietà di dispositivi, in modo da poterlo scrivere una volta e non doverlo più modificare a seconda delle necessità.
- **Efficienza**: Le istruzioni di WebAssembly sono mappate direttamente a quelle native della CPU, consentendo un'esecuzione molto più rapida rispetto a JavaScript. Questo è un vantaggio fondamentale per applicazioni che richiedono calcoli intensivi, elaborazione di dati o prestazioni grafiche elevate.
- **Sicurezza**: WebAssembly è progettato per essere sicuro e sandboxed, che lo isola dal sistema operativo e da altri programmi, garantendo che il codice in esecuzione non possa accedere ad informazioni esterne o danneggiare il sistema sottostante.

1.2.2 Funzionamento

Il processo di funzionamento di WebAssembly consente di scrivere applicazioni complesse in una vasta gamma di linguaggi di programmazione e di eseguirle, come spiegato sopra, con prestazioni elevate e una maggiore portabilità.
Questo può essere diviso in tre diverse fasi:

1. Scrivere il codice sorgente in uno dei tanti linguaggi di programmazione supportati (`C`,`C++`, `Rust`, `Python`, ecc..).
2. Una volta completato il codice sorgente, si utilizza un compilatore compatibile con WebAssembly, come Emscripten o LLVM, per convertirlo in un

file di bytecode WebAssembly, che avrà estensione *.wasm*. Durante questo processo, il linguaggio originale in cui è scritto il codice sorgente diventa irrilevante. Questo comporta che si può scrivere il programma in un linguaggio di alto livello e poi compilarlo in WebAssembly per eseguirlo nei browser.

In questo modo per i programmatori non c'è differenza tra i vari linguaggi di programmazione supportati da WebAssembly, semplificando così il processo di sviluppo e consentendo di utilizzare il linguaggio più adatto alle proprie esigenze.

3. Il risultato della compilazione è un file con estensione *.wasm* che può essere utilizzato in 2 modi:

   (a) viene incorporato in una pagina web usando JavaScript e, quando la pagina web viene caricata nel browser, quest'ultimo riconosce il codice WebAssembly e utilizza un motore di wasm dedicato per interpretare il bytecode ed eseguire il programma.

   (b) viene utilizzato all'esterno del browser da una Standalone Runtimes VM.

   In questo caso, il file *.wasm* non è destinato all'esecuzione all'interno del browser, ma viene utilizzato in un ambiente esterno. Questo approccio consente al codice WebAssembly di essere eseguito su una macchina virtuale indipendente dal browser, cosa che può essere utile in un contesto in cui è necessario accedere alle risorse di sistema o quando si desidera eseguire il codice in un ambiente diverso da quello del browser. Per consentire l'accesso alle risorse di sistema in questo contesto, si può fare uso di WASI (WebAssembly System Interface) che verrà approfondito in seguito.

1.3 WAT

Il formato di testo WebAssembly (WAT), noto anche come WebAssembly Text Format, è una rappresentazione creabile e leggibile da parte dell'uomo, delle istruzioni

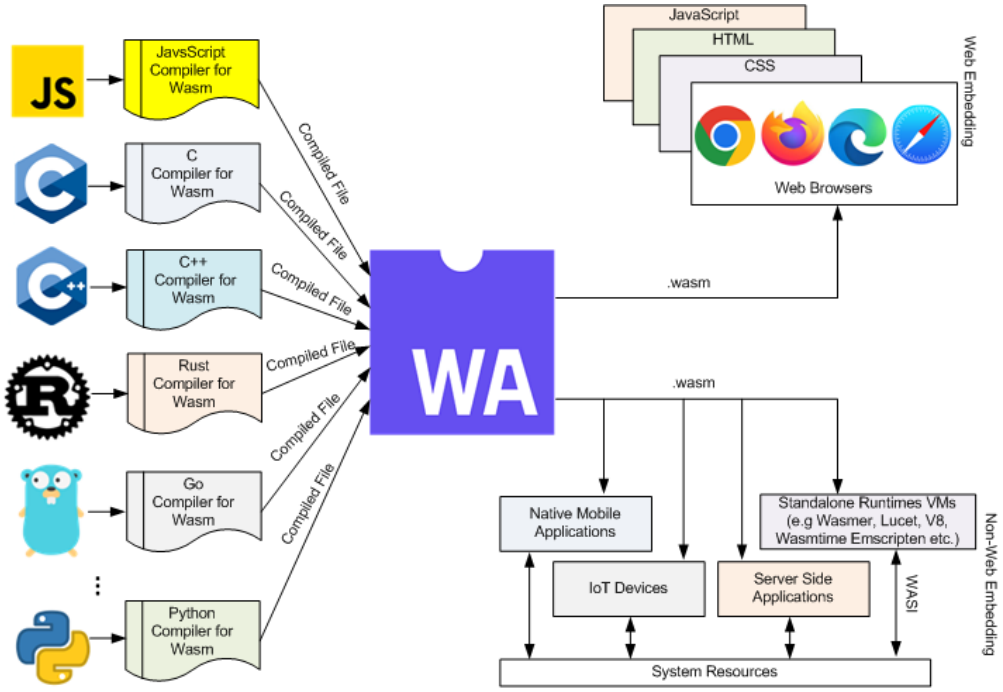


Figura 1.1: Architettura e Workflow di WebAssembly. [7]

che compongono un modulo WebAssembly. Si può considerare simile al linguaggio assembly, ma adattato alla macchina virtuale di WebAssembly. Sebbene il formato binario di WebAssembly sia più efficiente per l'esecuzione, il formato testuale offre numerosi vantaggi per la scrittura, la comprensione e la modifica del codice. Il suo scopo è quello di consentire agli sviluppatori di ispezionare in modo specifico le istruzioni eseguite da WebAssembly, in modo da poter fare debug, modificare istruzioni precise e capirne meglio il funzionamento.

Il WAT è composto da istruzioni, dati e metadati. Le istruzioni controllano il flusso di esecuzione del programma, i dati includono numeri, stringhe e altri tipi, mentre i metadati forniscono informazioni sul programma stesso. La sintassi è simile ai linguaggi assembly tradizionali, rendendola accessibile a chi ha già esperienza in questo campo.

In questo esempio vediamo del codice WAT che definisce una funzione per il calcolo del fattoriale di un numero. Il file `fact.wat` può poi essere facilmente compilato come tale `fact.wasm` con strumenti come WebAssembly Binary Toolkit

```
(module
  (func $factorial (param $n i32) (result i32)
    (if
      (i32.eq (get_local $n) (i32.const 0))         ; Caso base: if n == 0
      (then (i32.const 1))                          ;    return 1
      (else
        (i32.mul                                    ; Moltiplicazione fra:
          (get_local $n)                            ; n *
          (call $factorial                          ; Chiamata ricorsiva
            (i32.sub (get_local $n) (i32.const 1))  ;   con (n - 1)
          )
        )
      )
    )
  )
  (export "factorial" (func $factorial))
)
```

Listing 1: `fact.wat`

(WABT) [8].

1.3.1 Esempio

Per poter capire meglio il funzionamento di WebAssembly, si può introdurre un semplice esempio che dimostra, tramite il processo descritto nel punto precedente, come si possa creare e utilizzare un file `.wasm`. Si inizia dal seguente codice scritto in linguaggio `C` per il calcolo del fattoriale:

Una volta completato il primo step, si compila questo file `C` per ottenere un file `.wasm` utilizzando il compilatore `emscripten` con questo comando:

```
emcc fact.c -o fact.wasm
```

Il risultato di questa compilazione potrà poi essere caricato in un browser web o in un ambiente di runtime WebAssembly.

```c
int fact(int n) {
  if (n <= 1) {
    return 1;
  } else {
    return n * fact(n - 1);
  }
}
```

Listing 2: `fact.c`

```js
// Import del file wasm
const wasm = await WebAssembly.instantiateStreaming(fetch("fact.wasm"));
// Estrazione della funzione scritta originariamente in C
const fact = wasm.instance.exports.fact;
// Chiamata al codice WebAssembly
console.log(fact(5)); // 120
```

Listing 3: `fact.js`

Con questo codice andiamo a importare il file compilato `.wasm` all'interno di JavaScript e dopodiché possiamo chiamare la funzione `fact` definita originariamente all'interno del codice `fact.c` ed eseguirla quindi grazie al runtime WebAssembly all'interno del browser.

Capitolo 2

WASI

Contents

2.1 Che cos'è WASI

Nel 2019 è stato rilasciato WASI, acronimo di WebAssembly System Interface, che rappresenta un nuovo passo avanti nell'utilizzo di WebAssembly. Questo perché gli sviluppatori hanno voluto ampliare l'esecuzione di codice WASM su diversi dispositivi e ambienti, non solo all'interno del browser.

Un limite che inizialmente si incontra, però, è che, quando si sviluppano applicazioni al di fuori del browser, come ad esempio applicazioni desktop, server o altro, è spesso necessario che il codice interagisca con il sistema operativo, o comunque con le risorse di sistema, come il file system. Questa interazione è fondamentale per molte applicazioni, che potrebbero dover leggere o scrivere su file, gestire le connessioni, accedere ad informazioni hardware o effettuare una serie di altre operazioni.

Infatti WebAssembly, che è stato utilizzato principalmente all'interno dei browser, non presenta questa necessità di interazione diretta con il sistema operativo. Di conseguenza, WebAssembly non ha fornito un metodo per questa interazione, il che significa che il codice WASM, eseguito al di fuori del browser, non sa come comunicare con il sistema sottostante. Per questo è nata la necessità di creare un'interfaccia di sistema standardizzata per WebAssembly, che consentisse al codice di accedere in modo sicuro e controllato alle risorse di sistema, cioè proprio WASI.

2.2 Obiettivi principali di WASI: portabilità e sicurezza

Come appena spiegato, un programma software spesso si ritrova ad interagire con il sistema operativo sottostante e, queste operazioni, vengono eseguite attraverso le cosiddette chiamate di sistema, che sono funzioni specifiche fornite dal sistema operativo stesso.

Un problema che sorge è lo sviluppo di applicazioni che devono essere eseguite su diversi sistemi operativi. Ogni sistema operativo, infatti, ha le proprie chiamate di sistema, il che significa, che il codice scritto per un sistema operativo, potrebbe non funzionare correttamente su un altro, senza opportune modifiche. Questo porta a una mancanza di portabilità del codice e richiede un'implementazione specifica per ciascun sistema operativo.

Per risolvere questo problema, si usa il concetto di astrazione. Ovvero, l'astrazione consente di nascondere i dettagli implementativi sottostanti e fornire

un'interfaccia standard, che può essere utilizzata indipendentemente dal sistema operativo specifico. Questo è ciò che viene fatto attraverso le librerie standard dei linguaggi di programmazione, che forniscono un set di funzioni e metodi che possono essere utilizzati senza che il programmatore si debba preoccupare dei dettagli di implementazione. Sarà a carico del compilatore, durante il processo di compilazione del codice sorgente, selezionare l'implementazione appropriata dell'interfaccia standard, in base al sistema operativo a cui è destinato il programma.

Ad esempio, se si sta compilando per Windows, il compilatore utilizzerà le funzioni fornite dalle Windows API, mentre se si sta compilando per Mac o Linux, userà le funzioni POSIX. Il problema con WebAssembly è che, durante la compilazione, non si sa quale sistema operativo sarà di destinazione, poiché il codice WebAssembly potrebbe essere eseguito su diversi dispositivi o ambienti.

Pertanto, è necessario individuare un modo per conservare questa astrazione, in maniera tale da consentire la compilazione del codice una sola volta, e, il suo utilizzo, su una varietà di sistemi operativi diversi, preservando, allo stesso tempo, le due motivazioni principali che hanno portato alla creazione di WASI fin dall'inizio.

I due obiettivi principali di WASI quindi sono:

- **Portabilità:** permettere di sviluppare applicazioni WebAssembly che utilizzano le API di sistema in modo indipendente dal sistema operativo o dall'architettura hardware.

- **Sicurezza:** offrire un ambiente sicuro e isolato per l'esecuzione di applicazioni WebAssembly. Le sue API sono progettate per limitare l'accesso alle risorse di sistema non autorizzato, proteggendo ulteriormente WebAssemby da vulnerabilità.

2.3 Design modulare

In modo diverso dai sistemi operativi che definiscono un vasto insieme di system call da utilizzare, WASI utilizza un approccio più modulare. Invece di una API

monolitica, suddivide le operazioni di sistema in sottogruppi più piccoli che consentono operazioni su file, connessioni di rete e funzionalità temporali. In questa maniera, se un modulo Wasm necessità solamente dell'accesso ai file, il runtime WASI può implementare solamente le funzionalità di file system, senza dover implementare tutte le altre funzioni, come ad esempio la funzionalità di rete, riducendo, quindi, l'uso di risorse. Questo metodo inizia in fase di compilazione di un file in formato `.wasm`, in cui, indipendentemente dal linguaggio di sviluppo utilizzato, il compilatore identifica le API che richiamano librerie specifiche del linguaggio, e le sostituisce con degli import tipici di WASI. Questi import vengono poi interpretati dal runtime WASI e associati alle implementazioni specifiche del sistema operativo prescelto.

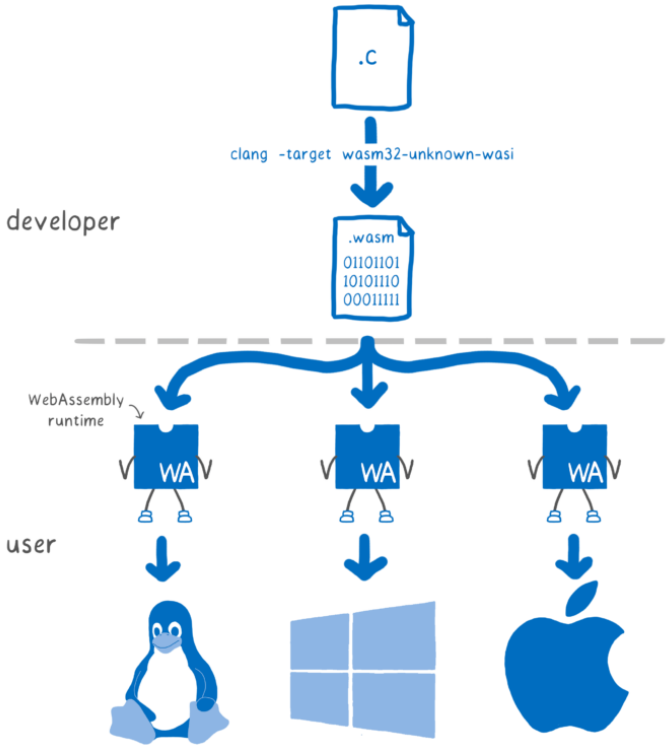


Figura 2.1: Un file sorgente C viene compilato in un singolo file binario. [1]

In termini più tecnici, quindi, WASI si configura come un insieme di API, ciascuna delle quali fornisce un set di funzioni per interfacciarsi con una specifica area del sistema operativo. Le principali API includono:

- *API File*: Permette l'accesso ai file e alle directory del sistema.
- *API Networking*: Permette la comunicazione con la rete tramite l'utilizzo di socket.
- *API Process*: Permette la gestione dei processi e dei thread, mettendo a disposizione varie funzioni per creare nuovi processi, eseguire programmi e gestire la concorrenza.
- *API Memoria*: Permette la gestione della memoria, permettendo di allocare e deallocare la memoria al bisogno.
- *API I/O*: Permette l'accesso alle periferiche di input/output.

L'utilizzo di API modulari rende WASI un'interfaccia flessibile e portabile. Gli sviluppatori possono utilizzare solo le API di cui hanno bisogno, il che rende il codice più snello e sicuro.

Mantenere tutte queste parti separate consente, inoltre, di gestire in modo migliore la sicurezza, in modo tale da fornire al modulo Wasm solamente certi tipi di permessi, come ad esempio l'accesso al file system, senza però concedere il permesso di effettuare operazioni di rete, come l'apertura di una socket. La modularità consente, inoltre, a WASI di svilupparsi in modo armonioso, dato che per supportare nuove funzionalità basta aggiungere nuovi moduli, invece di alterare le API di base.

2.4 Meccanismi di Sicurezza WASI

2.4.1 Sandboxing delle applicazioni WebAssembly

Il sandboxing è un modello di sicurezza che si basa sull'isolamento dove viene quindi creato un ambiente virtuale isolato e controllato. Questo metodo confina l'esecuzione di un modulo WebAssembly in un contesto sicuro, che definisce le azioni consentite e le risorse usufruibili. WASI applica una stretta separazione fra il modulo WebAssembly e il sistema operativo della macchina host e, di default,

non ha accesso diretto al file system, alla rete, ai processi, ad altre applicazioni e ad altre funzionalità importanti del sistema, riducendo in questo modo il rischio di intrusioni e malware, che potrebbero cercare di accedere a risorse sensibili e, migliorando la stabilità e l'affidabilità, in quanto utilizzare un ambiente sandboxed riduce la probabilità di errori e crash.

2.4.2 Gestione delle risorse tramite Capabilities

Un altro modello adottato da WASI per la sicurezza è basato sulle capabilities, ovvero capacità, in linea con il principio del privilegio minimo (POLP), in cui l'accesso alle risorse è controllato da permessi specifici. In questo modo, WebAssembly può accedere solo alle risorse per le quali possiede la capability corrispondente, ottenuta al momento dell'avvio o durante l'esecuzione, e, successivamente, controllata dal runtime WASI prima di autorizzare l'accesso a quella specifica risorsa. Questo comporta due benefici principali, ovvero: maggiore granularità del controllo, dato che permette di definire con precisione i permessi di accesso per ogni modulo, e migliore conformità alle normative, essendo un modello che facilità la conformità agli standard di privacy e sicurezza.

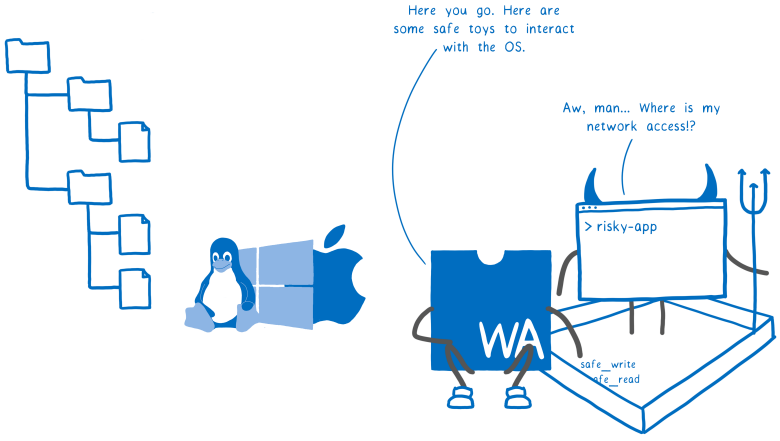


Figura 2.2: Un runtime che inserisce funzioni sicure nella sandbox con un'applicazione. [1]

2.5 Componenti chiave di WASI

- **Operazioni su file** : il codice Wasm non può aprire qualsiasi file. Ha bisogno di un file descriptor che gli permetta l'accesso ad una specifica cartella nel file system. Invece delle tradizionali chiamate `open`, `read` and `write`, WASI offre delle system call specializzate:

  - `path_open`: apre un file all'interno della cartella al quale il runtime Wasm ha accesso.
  - `fd_read`: legge dati da un file.
  - `fd_write`: scrive dati su un file.

- **Connessioni di rete**: qui le funzioni `sock_accept`, `sock_recv`, `sock_send`, `sock_shutdown` si comportano in modo molto simili alle corrispettive funzioni `accept`, `recv`, `send` e `shutdown` in POSIX.

- **Funzionalità temporali**: data la natura sandboxed di WASI, questo non ha accesso diretto al clock di sistema, e definisce, quindi, due funzioni standard per accedere al clock ossia `clock_get_time`, per ottenere il tempo corrente (in modo molto simile a `clock_gettime` in POSIX) e `poll_oneoff`, utilizzata, invece, per sottoscriversi a un evento futuro, come ad esempio il termine di un timer.

- **Numeri casuali**: essendo WASI un'interfaccia, sarà la macchina host a fornire una sorgente di casualità, che potra essere utilizzata tramite la funzione `random_get`.

2.5.1 Differenza compilazione classica vs WASI

Consideriamo uno scenario in cui l'obiettivo del nostro programma sia quello di aprire e leggere il contenuto di un file chiamato `input.txt`.

```c
#include <stdio.h>
int main() {
    FILE *file = fopen("input.txt", "r"); // Apre il file in modalità di lettura
    if (file == NULL) {
        printf("Error opening file!\n");
        return 1;
    }
    // Legge e processa i dati del file...
    fclose(file); // Chiude il file una volta terminato
    return 0;
}
```

Listing 4: File `C` preso come esempio

Andiamo prima di tutto ad analizzare cosa accade durante una compilazione tradizionale (ad esempio su Linux):

1. La funzione `fopen` nella libreria C standard usa internamente la system call `open` fornita dal kernel di Linux.
2. Il kernel di Linux gestisce l'apertura del file, controlla i permessi in base all'utente che ha lanciato il processo e ritorna il file descriptor.
3. La stessa cosa succederà per interazioni successive che utilizzano le system call `read` e `close`.

Ora vediamo cosa succede nel caso di utilizzo di WebAssembly con WASI:

1. Durante la compilazione, invece della libreria standard C, vengono utilizzate delle funzioni specifiche di WASI, che contengono delle implementazioni alternative di varie funzioni, come, ad esempio, la funzione `fopen`.
2. Queste implementazioni contengono delle funzioni (come ad esempio `__wasi_path_open`) che sostituiscono le system call e sono l'unico mezzo che WASI ha per comunicare con le risorse di sistema. Sarà compito del runtime utilizzato implementare queste funzioni WASI, in modo che il codice compilato funzioni correttamente.

```c
// Semplificato a fini di illustrazione
FILE *fopen(const char *filename, const char *mode) {
    int fd = __wasi_path_open(
                0, // File descriptor che rappresenta una cartella
                filename,
                O_RDONLY,   // Flag che indica la modalità di sola lettura
                0           // Nessuna altra flag specifica
             );
    // ... gestione di eventuali errori ...
    return fd; // Ritorna il file descriptor (oppure un errore)
}
```

Listing 5: Possibile implementazione della funzione `fopen` all'interno di WASI

2.6 Confronto con le interfacce di sistema tradizionali come POSIX

Come è stato spiegato, le interfacce di sistema sono componenti fondamentali che permettono al software di interagire con il sistema operativo. È importante comprendere il ruolo che queste interfacce svolgono nell'ambito dello sviluppo del software.

Si può prenderne una molto usata come esempio, ovvero POSIX, che sta per Portable Operating System Interface for Unix, ed è un insieme di standard definiti dall'IEEE, che fornisce un'interfaccia comune per l'interazione tra il software e il sistema operativo Unix-like. POSIX ha garantito una base per lo sviluppo di applicazioni su sistemi Unix-like, fornendo funzionalità e API collaudate.

D'altra parte, WASI, o WebAssembly System Interface, rappresenta una nuova valida opzione nell'evoluzione delle interfacce di sistema. È progettata per la portabilità e la sicurezza e offre un'interfaccia standardizzata per l'accesso alle risorse di sistema da parte del codice WebAssembly, indipendentemente dal sistema operativo sottostante. In questo modo, WASI supera le limitazioni di POSIX, aprendo nuove possibilità per lo sviluppo di applicazioni web ed embedded, in primo luogo,

nella portabilità: mentre POSIX è specifico per sistemi Unix-like, WASI è estremamente portabile e progettato per funzionare su diverse architetture e sistemi operativi. In secondo luogo, nella sicurezza: infatti POSIX non è stata progettata con la sicurezza come priorità e alcune delle sue API possano risultare vulnerabili, mentre WASI è stato creato proprio per poter avere più sicurezza, con API progettate per diminuire i rischi e proteggere i software.

Ovviamente la scelta tra POSIX e WASI dipende dalle esigenze specifiche del progetto: se si necessita di un'interfaccia completa e collaudata per sistemi Unix-like, POSIX potrebbe essere la scelta migliore, in quanto potrebbe risultare leggermente più veloce. Tuttavia, se si prediligono la portabilità e la sicurezza, WASI rappresenta una valida alternativa sicura per l'accesso alle risorse di sistema.

Capitolo 3

Runtime

Contents

3.1 Tipi di runtime WASM

Analogamente al linguaggio assembly estremamente efficiente, ottimizzato per i browser web e altri ambienti di esecuzione, WASM richiede un runtime, ovvero un ambiente software che esegue il codice, interagendo con il sistema sottostante.

Esistono diversi tipi di runtime WASM, ciascuna con caratteristiche e scopi specifici. Questi runtime possono essere suddivisi principalmente in tre macro-categorie:

- **Runtime WASI**: Questi runtime implementano le API WASI (WebAssembly System Interface), fornendo un'interfaccia standardizzata per l'interazione con il sistema operativo. Sono ideali per lo sviluppo di applicazioni cross-platform, ovvero che possono essere eseguite su diversi dispositivi o ambienti, senza la necessità di scrivere il codice specifico per ciascuna piattaforma, e serverless, cioè applicazioni che vengono eseguite senza la necessità di gestire l'infrastruttura sottostante, il server, consentendo comunque al codice di accedere alle risorse di sistema in modo sicuro e portabile.

- **Runtime embedded**: Questi runtime sono progettati per l'esecuzione di codice Wasm in sistemi embedded, caratterizzati da risorse limitate, essendo dispositivi informatici progettati per eseguire specifiche funzioni in un ambiente apposito. Questi runtime ottimizzano l'esecuzione del codice Wasm per garantire prestazioni ottimali, anche su dispositivi con restrizioni hardware.

- **Runtime browser**: Quasi tutti i browser web integrano un runtime Wasm per eseguire applicazioni web complesse direttamente nel browser stesso. Questo permette agli sviluppatori di sfruttare la potenza del codice Wasm per creare esperienze utente avanzate, senza la necessità di plugin esterni o estensioni. ‘

3.2 Funzionamento

Come è stato detto, ad oggi esistono una serie di runtime differenti, che offrono tutti diverse caratteristiche ed implementazioni ma, nonostante questo, tutti seguono dei passaggi fondamentali per funzionare:

1. **Caricamento del file `.wasm`**: Il runtime, come prima cosa, deve caricare un modulo WASM, che sostanzialmente è un file binario con estensione `.wasm`.

2. **Decodifica e Validazione**: Il runtime decodifica il formato binario del file `.wasm` in istruzioni che può eseguire. In questa fase esegue anche una validazione del modulo per assicurarsi che segua le regole di WebAssembly, garantendo sicurezza e predicibilità.

3. **Compilazione**: questo è un passaggio opzionale, anche se molti runtimes compilano il codice Wasm per tradurlo in codice macchina altamente ottimizzato per la specifica architettura hardware sulla quale esegue. Questo passaggio fa si che il codice Wasm possa essere eseguito con performance vicine alle applicazioni native. Questa compilazione può essere fatta principalmente in 2 modi:

   - *Ahead-Of-Time (AOT)*: il codice Wasm viene compilato in codice macchina prima dell'esecuzione, il che comporta tempi di avvio più rapidi ma un maggior utilizzo di memoria.
   - *Just-In-Time (JIT)*: il codice Wasm viene compilato in codice macchina durante l'esecuzione. Ciò consente di ridurre l'utilizzo di memoria e di ottimizzare in base all'ambiente di esecuzione, ma può comportare un overhead maggiore in fase di avvio.

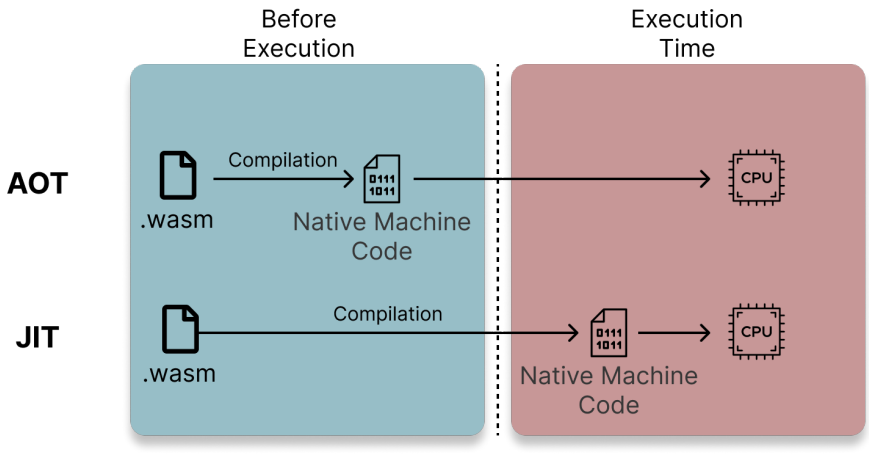


Figura 3.1: AOT vs. JIT

4. **Gestione della memoria**: Come già visto, Wasm funziona con un modello di memoria isolato, in modo da prevenire che il codice Wasm manometta la memoria di sistema. In questo maniera, un modulo non può accedere alla memoria di un altro modulo, né alla memoria del runtime o a quella del sistema operativo sottostante del runtime, a meno che non gli venga dato esplicito accesso. Poichè è compito del runtime Wasm allocare una porzione di memoria lineare dove il modulo Wasm può salvare i proprio dati, le dimensioni di questa sono sempre note, in modo che il runtime possa verificare se un offset di memoria, a cui un modulo sta cercando di accedere, è ancora all'interno dei limiti della sua memoria allocata.

5. **Esecuzione di funzioni**:

   (a) *Risoluzione degli import*: Quando un modulo Wasm deve interagire con l'ambiente esterno, ad esempio per mostrare qualcosa a schermo o per comunicare sulla rete, il runtime fornisce un meccanismo di *import* per consentire a tali interazioni di avvenire.
   Gli *import* sono funzioni che vengono definite nell'ambiente dell'host, cioè l'ambiente in cui il modulo Wasm viene eseguito, come un browser o un altro runtime Wasm. Queste funzioni sono, quindi, rese disponibili al modulo Wasm, consentendogli di chiamare e utilizzare le funzionalità fornite dall'ambiente dell'host per eseguire le operazioni desiderate, come l'interazione con l'interfaccia utente o la comunicazione di rete.

   (b) *Chiamata a funzione Wasm*: Nel momento in cui il codice all'interno dell'environment dell'host, ad esempio JavaScript all'interno di un browser, vuole eseguire una funzione Wasm, chiama una funzione speciale del runtime Wasm, nota come "funzione di esportazione". Questa funzione funge da ponte tra il codice dell'environment, come JavaScript, e il codice Wasm. La funzione di esportazione riceve gli argomenti dalla funzione JavaScript e li converte in un formato compatibile con il codice Wasm. Questo passaggio assicura che i dati vengano trasferiti e interpretati correttamente dal codice Wasm. Una volta che gli argomen-

ti sono pronti, la funzione di esportazione passa il controllo al codice Wasm, specificando la funzione da eseguire e i relativi argomenti.

(c) *Esecuzione del codice Wasm*: Il codice Wasm riceve gli argomenti dalla funzione di esportazione e, utilizzando le informazioni fornitegli, esegue la funzione specificata. Dopo l'esecuzione della funzione Wasm, viene generato un valore di ritorno. Questo valore viene poi restituito alla funzione di esportazione, che si occupa di convertirlo in un formato compatibile con JavaScript. Infine, il valore convertito viene restituito alla funzione JavaScript chiamante, consentendo al codice iniziale di continuare l'esecuzione, utilizzando il valore di ritorno ottenuto dalla funzione Wasm. Questo processo completa la comunicazione tra il codice dell'enviroment dell'host e il codice Wasm, consentendo un flusso di dati.

6. **Interazione con il sistema**: quando un modulo Wasm necessità di interagire con il sistema, ad esempio per leggere file o per fare richieste di rete, esso comunica con l'interfaccia WASI implementata dal runtime, che va a effettuare la system call specificatamente richiesta.

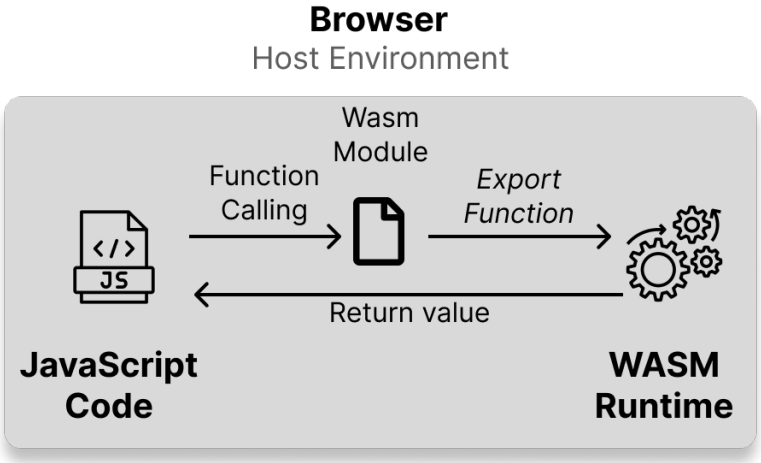


Figura 3.2: Chiamata a funzione WASM

3.3 Runtimes analizzati

Durante la fase di test delle performance di WebAssembly e di WASI, per poter valutare le loro prestazioni, sono stati presi in considerazione diversi runtime, i quali implementano, alcuni completamente, altri parzialmente, le specifiche WASI. Di seguito vengono descritti tutti i runtime utilizzati durante i test.

3.3.1 Wasmer

Wasmer[11], il primo runtime Wasm trattato, si distingue dagli altri runtime principalmente per la sua portabilità. L'obiettivo principale di Wasmer, infatti, è quello di offrire un ambiente in grado di poter eseguire moduli wasm praticamente ovunque, a partire da desktop, server, ambienti cloud, sistemi embedded e nel browser. Grazie a questa caratteristica, Wasmer riesce a supportare una vasta quantità di casi d'uso. Rispetto ad altri runtime, Wasmer offre una serie di compilatori da poter utilizzare a seconda delle proprie esigenze ed i principali sono:

- **LLVM[4]**: Un'infrastruttura di compilazione open source scritta in C++, progettata per l'ottimizzazione di programmi durante la compilazione e l'esecuzione. È compatibile con varie architetture tra cui Intel x86, ARM e RISC-V.
- **Cranelift**: è un compilatore open source scritto in Rust, specificatamente progettato per la generazione di codice macchina. È conosciuto per la sua capacità di produrre codice estremamente veloce ed è generalmente utilizzato laddove il tempo di avvio è una priorità fondamentale.
- **Singlepass**: è il compilatore più semplice e leggero, progettato per massimizzare la velocità di compilazione e generare codice macchina di dimensioni ridotte. Singlepass è utilizzato principalmente in contesti dove la dimensione finale del codice è critica, come nelle applicazioni embedded e dove è prioritaria una compilazione rapida.

Wasmer, inoltre, offre un supporto per diversi backend, tra cui JavascriptCore, che permette di eseguire moduli Wasm direttamente all'interno di un browser web, e

il browser stesso, integrandosi nativamente con Chrome, Firefox, Safari ed altri. Inoltre, Wasmer implementa e supporta interamente WASIX, uno standard che estende le API di sistema WASI con un set di system call aggiuntive.

3.3.2 Wasmtime

Wasmtime[12] è un runtime open source scritto in Rust. È stato progettato per mantenere la massima sicurezza e prestazioni elevate. Questa attenzione deriva dalla natura intrinsecamente non attendibile del codice WebAssembly, che potrebbe provenire da fonti potenzialmente sconosciute. Per questo motivo Wasmtime include diverse funzionalità, per evitare i rischi di vulnerabilità, come il sandboxing, la protezione dello stack e le Capabilities-Based security, ovvero la concessione al codice Wasm solo delle autorizzazioni strettamente necessarie per funzionare, limitando l'accesso alle risorse del sistema e prevenendo attacchi.

Wasmtime utilizza il compilatore Cranelift, spiegato sopra, per offrire quindi ottime prestazioni e tempi di avvio rapidi. Inoltre, implementa completamente la WASI preview 2[9], uno standard che fornisce un'interfaccia API portabile per le applicazioni Wasm, in modo da rendere Wasmtime compatibile con una vasta tipologia di applicazioni esistenti. Tuttavia, essendo ancora in fase di sviluppo, a volte potrebbe risultare instabile e quindi è consigliabile usare WASI Classic.

3.3.3 Node.js

Node.js[6] è un ambiente di runtime JavaScript che consente di eseguire codice al di fuori del browser, sfruttando l'engine JavaScript V8. Questo motore ha la capacità nativa di compilare ed eseguire codice Wasm, che può offrire vantaggi rispetto al codice JavaScript. Utilizzando Node.js, è possibile integrare facilmente moduli Wasm all'interno del codice JavaScript, in modo semplice e trasparente.

L'utilizzo di moduli Wasm in Node.js consente di sfruttare diverse funzionalità ed avere migliori prestazioni. Questo perché, grazie alla compilazione in codice macchina nativo, Wasm offre in molti casi prestazioni superiori rispetto a JavaScript. Inoltre, Wasm può essere utilizzato per implementare funzionalità non

disponibili in JavaScript, come accesso diretto all'hardware o l'esecuzione di calcoli complessi. La portabilità del codice Wasm consente inoltre di eseguirlo senza particolari modifiche su diverse piattaforme. Il passaggio da JavaScript a Wasm, però, può comportare un certo overhead nelle prestazioni, infatti in alcune situazioni, come ad esempio per operazioni semplici, potrebbe non valer la pena utilizzare WebAssembly.

Le moderne versioni di Node.js offrono un supporto sperimentale per WASI all'interno di moduli Wasm. WASI definisce un'interfaccia API portabile per le applicazioni Wasm, consentendo loro di interagire con le risorse di sistema in modo standard, come file, rete e memoria, anche se l'implementazione di WASI in Node.js è ancora in fasi di modifica e potrebbe non essere del tutto stabile.

3.3.4 Wazero

Wazero[13] è un runtime WebAssembly progettato specificatamente per il linguaggio di programmazione Go. Questo runtime consente di inserire facilmente moduli Wasm all'interno delle applicazioni Go, offrendo molti vantaggi. Una delle caratteristiche di Wazero è la sua totale indipendenza da librerie esterne, rendendolo semplice da incorporare in qualsiasi programma Go senza dover gestire i vari requisiti di dipendenza. Inoltre, Wazero è altamente portabile, disponibile per molte piattaforme come Windows, macOS, Linux, Android e iOS.

L'interfaccia di programmazione di Wazero è progettata per essere semplice e intuitiva, in modo da facilitare l'esecuzione di moduli Wasm all'interno delle applicazioni Go esistenti. Inoltre, Wazero sfrutta la tecnica di compilazione Just-In-Time (JIT) per ottimizzare le prestazioni del codice Wasm. Tuttavia è importante fare anche un'altra considerazione, ovvero, poiché Wazero esegue gran parte della compilazione durante l'esecuzione, può avere un overhead maggiore rispetto ad altri runtime che adottano tecniche di compilazione più pesanti. Per sopperire a questo problema, Wazero offre la possibilità di compilare i moduli Wasm in anticipo, tramite la compilazione AOT, per ridurre l'overhead. Questa funzionalità, tuttavia, è ancora in fase di sviluppo e potrebbe non essere ottimizzata.

3.3.5 iwasm

Wasm[2] fa parte di WebAssembly Micro Runtime (WAMR), framework open source che facilita l'esecuzione di applicazioni Wasm in ambienti con risorse limitate, perciò, mentre WAMR fornisce un ambiente completo per l'esecuzione di applicazioni Wasm, iwasm si concentra sul fornire un ambiente runtime minimale ed essenziale, che gestisce la memoria, le chiamate di sistema e l'interazione tra il modulo Wasm e il sistema host. Inoltre è estremamente compatto, il che lo rende preferibile per dispositivi con memoria limitata. Per questo motivo è progettato per l'esecuzione di moduli Wasm in ambienti con risorse limitate, come dispositivi embedded o browser web.

Una delle caratteristiche di iwasm è l'utilizzo della compilazione AOT (Ahead-of-Time) per convertire il codice Wasm in codice macchina nativo in modo da offrire buone prestazioni e un basso utilizzo di memoria, molto importanti in contesti con risorse limitate.
Quanto alle sue applicazioni quindi, iwasm è adatto ad ambienti con risorse limitate. Può essere utilizzato su dispositivi embedded come sensori, attuatori e sistemi di controllo, nell'IoT (Internet of Things), ma anche all'interno dei browser web per eseguire applicazioni Wasm in modo più efficiente rispetto al codice JavaScript.

3.3.6 Wasmedge

Wasmedge[10] si distingue tra i runtime WebAssembly per l'attenzione di progettazione riguardo a leggerezza, velocità e sicurezza. A differenza di altri runtime, che mirano a offrire un vasto insieme di funzionalità, Wasmedge si concentra sull'esecuzione di moduli WebAssembly, rendendolo particolarmente adatto per utilizzi come l'edge computing, i microservizi e WASM in ambienti con risorse limitate.

Grazie all'utilizzo di LLVM per la compilazione ahead-of-time (AOT), dovrebbe consentire un'esecuzione efficiente del codice WASM. Tuttavia, questa attenzione alle prestazioni comporta una riduzione delle funzionalità, in quanto al momento non supporta alcune funzioni avanzate come, ad esempio, l'interoperabilità diretta con JavaScript.

Inoltre, Wasmedge offre un'implementazione solida e orientata alla sicurezza di WASI. Favorisce l'esecuzione in sandbox garantendo così che i moduli WASM vengano eseguiti in un ambiente limitato per prevenire azioni dannose sul sistema host. Queste caratteristiche lo rendono particolarmente adatto in scenari dove le prestazioni, l'utilizzo ridotto di memoria e la sicurezza sono cruciali, anche a costo di rinunciare a funzionalità più avanzate che si potrebbero trovare in altri runtime. Ciò nonostante, all'interno dei risultati mostrati, Wasmedge è stato omesso in quanto i suoi tempi di esecuzione risultano sproporzionati rispetto a tutti gli altri runtime per tutti i file testati.

Capitolo 4

Test e Configurazione

Contents

4.1 Libreria Libsodium

Libsodium [3] è una libreria software open source riguardante la crittografia, focalizzata sull'offrire strumenti ad alte prestazioni e multi-piattaforma per una serie di operazioni crittografiche. Queste operazioni includono crittografia, decrittografia, firme digitali, hashing delle password e altro. È un fork della libreria crittografica NaCl [5], cioè libsodium è stata sviluppata partendo dal codice sorgente di NaCl, tuttavia, libsodium ha subito modifiche, miglioramenti e aggiunte. Ciò significa che, pur offrendo API simili, libsodium ha sviluppato nuove funzionalità rispetto a NaCl.

Un principio di progettazione chiave di libsodium è la facilità d'uso e l'impiego corretto rispetto a librerie crittografiche di livello inferiore, come OpenSSL.

Libsodium è progettata per essere intuitiva e minimizzare il rischio di errori degli sviluppatori durante l'implementazione di operazioni di sicurezza sensibili. Inoltre, libsodium fornisce strumenti di alto livello anziché solo primitive grezze. Ad esempio, offre la crittografia autenticata, che protegge sia i dati che la loro integrità, e offre anche funzionalità per la memoria sicura delle password.
I punti di forza di libsodium includono:

- Crittografia a chiave segreta: include operazioni come crittografia e decrittografia simmetriche, utilizzando una chiave condivisa, ma anche crittografia autenticata.

- Crittografia a chiave pubblica: libsodium supporta operazioni come firme digitali e scambio di chiavi.

- Hashing delle password: libsodium offre funzioni progettate per fare hashing in modo sicuro e fare salting delle password, riducendo il rischio di accessi non autorizzati.

- Generazione di numeri casuali: libsodium fornisce anche funzionalità per la generazione di numeri casuali di alta qualità, che sono essenziali per molte operazioni crittografiche.

4.2 Testare i runtime

Il mondo dei runtime WebAssembly è in continua evoluzione e, negli ultimi anni, hanno visto una vera e propria accelerazione. Nuove proposte come Wazero sono emerse, mentre altri progetti sono sempre più scemati. Le prestazioni complessive sono aumentate considerevolmente, in gran parte grazie all'ascesa di compilatori come Cranelift o LLVM, molto validi in termini di ottimizzazione del codice.

Perciò, al giorno d'oggi, la scelta del runtime più adatto dipende dalle priorità e obiettivi del progetto. Ad esempio, Wasmer associato ad LLVM offre un'incredibile velocità, rendendolo ideale quando la priorità è la massima performance. Node.js rimane un solido concorrente grazie alle sue API stabili e all'integrazione

con l'ecosistema JavaScript. Se l'obiettivo è invece l'embedding con un overhead minimo e alta velocità, si potrebbe prediligere iwasm. Per esigenze specifiche, si possono considerare Wasmer per l'ampio ecosistema, Wasmedge per il design estensibile o wasmtime per l'attenzione alla sicurezza. Insomma, a seconda della propria necessità, ci sarà un runtime che implementa al meglio delle proprie potenzialità quell'aspetto.

Oltre alle prestazioni, è importante pure considerare la standardizzazione di WASI, che definisce le interazioni sicure tra WebAssembly e le risorse di sistema. Il supporto di WASI varia tra i runtime, questo perché, alcuni lo implementano completamente, mentre altri offrono solo un supporto parziale. La scelta del runtime influenza quindi il livello di sicurezza che si ottiene.

Testare libsodium con vari runtime WASM/WASI come Wasmer, Wasmtime, Node, Wazero, IWasm e Wasmedge ha diversi scopi importanti quali:

- **Benchmarking completo**: Libsodium, come spiegato prima, fornisce un set standardizzato di funzioni crittografiche, rendendolo un benchmark ideale per valutare come diversi runtime gestiscono le operazioni crittografiche più importanti.

- **Valutazione della sicurezza**: Data la natura sensibile della crittografia, testare le prestazioni di libsodium su diversi runtime fornisce informazioni preziose su come gestiscono operazioni di sicurezza critiche, potenzialmente scoprendo vulnerabilità.

- **Portabilità multi-piattaforma**: La compilabilità di libsodium in WASM lo rende adatto a testare il potenziale multi-piattaforma di WASM e WASI. Valutare la funzionalità crittografica su diversi sistemi operativi e architetture può aiutare a misurarne la portabilità.

- **Rilevanza nel mondo reale**: L'utilizzo di libsodium in applicazioni reali rende i risultati dei test direttamente applicabili alle prestazioni di WASM e WASI nel proteggere dati e operazioni sensibili in scenari reali.

In questo capitolo, è importante testare i vari runtime con i file crittografici di libsodium al fine di comprendere al meglio le loro caratteristiche di prestazio-

ne. Stressare un aspetto così specifico dell'esecuzione del runtime, contribuisce alla comprensione delle tecnologie WASM/WASI e alla loro implementazione attraverso i diversi runtime. Inoltre, l'analisi di WASM, WASI e libsodium offre un'importante opportunità per osservare le prestazioni e la sicurezza di queste tecnologie. Questo può fornire preziose informazioni sulle strategie di ottimizzazione adottate dai diversi runtime. Ciò consente di fare scelte più consapevoli riguardo alla selezione del runtime più adatto alle specifiche esigenze di un determinato progetto.

4.3 Ambiente di test

Di seguito vengono indicate le specifiche hardware della macchina sulla quale sono stati eseguiti tutti i test:

- CPU: Intel(R) Core(TM) i3-6100 CPU @ 3.70GHz
- RAM: 8GB
- Sistema Operativo: Ubuntu 22.04 su WSL 1(Windows Subsystem for Linux) con Windows 10

Versioni dei programmi utilizzati:

- wasmtime: 17.0.1
- wasmer: 4.2.5
- iwasm: 1.3.2
- Node.js: 21.6.2
- wazero: 1.6.0
- Wasmedge: 0.13.5

4.4 Dati raccolti

Per ottenere un risultato più preciso delle prestazioni di ciascun runtime, è stato pensato un metodo per ripetere la compilazione di ogni file più volte. Infatti ognuno di quest'ultimi, è stato eseguito 100 volte con ogni runtime. Questo per poter minimizzare il rumore casuale presente nei dati e ottenere un valore finale più realistico.

Per semplificare l'esecuzione ripetitiva dei test, è stato creato uno script `.bash`. Lo script svolge diverse funzioni come:

- **Ricerca**: Trova tutti i file che corrispondono a un determinato pattern (ad esempio, file con estensione ".wasm").
- **Esecuzione**: Esegue ogni file trovato con tutti i runtime disponibili.
- **Utilizzo dei comandi**: Lo script utilizza i comandi presenti nel file, denominato `execute_benchmarks.sh`, per eseguire i test con i vari runtime.

L'automazione con uno script `.bash` offre vari vantaggi tra cui: efficienza, dato che elimina la necessità di eseguire manualmente i test per ogni file e runtime, precisione, che riduce il rischio di errori durante l'esecuzione dei test e continuità, ovvero garantisce che tutti i test vengano eseguiti con le stesse impostazioni e condizioni. Per questo l'esecuzione ripetuta dei test e l'utilizzo di uno script `.bash` per l'automazione hanno contribuito a ottenere risultati più precisi e affidabili.

Bisogna porre un'attenzione particolare a Node.js, in quanto, nonostante sia ampiamente utilizzato come runtime per WebAssembly e WASI, questo ha un limite, ovvero non offrire un metodo diretto per eseguire file `.wasm` da riga di comando. Per ovviare questo problema, è stato fatto uno script chiamato `execute_node.js`. In questo maniera, il modulo .wasm specificato viene caricato nella memoria di Node.js, creando un ambiente di esecuzione per il codice wasm che consente poi l'interazione tra JavaScript e il file wasm.

I vantaggi voluti da questo script sono principalmente due: eliminare il problema dell'assenza dell'esecuzione di file .wasm da riga di comando e il voler au-

```
# Iterazione dei file
for wasm_file in $file_pattern; do
    # Iterazione dei runtime
    for runtime in "${runtimes[@]}"; do
        # Iterazione per eseguire 100 volte ogni test
        for i in $(seq 1 $num_executions); do
            # Esegue il $wasm_file con il $runtime corrente...
            if [ "$runtime" == "wasmtime" ]; then
                wasmtime "$wasm_file"
            elif [ "$runtime" == "node" ]; then
                node --no-warnings execute_node.js "$wasm_file"
            elif [ "$runtime" == "wasmer_singlepass" ]; then
                wasmer run  "$wasm_file" --singlepass
            elif [ "$runtime" == "wasmer_cranelift" ]; then
                wasmer run  "$wasm_file" --cranelift
            elif [ "$runtime" == "wasmer_llvm" ]; then
                wasmer run  "$wasm_file" --llvm
            elif [ "$runtime" == "wazero" ]; then
                wazero run "$wasm_file"1
            elif [ "$runtime" == "iwasm" ]; then
                iwasm "$wasm_file"
            else
                echo "Unsupported runtime: $runtime"
            fi
        done
    done
done
```

Listing 6: `execute_benchmarks.sh`

tomatizzare questo processo per eliminare la necessità di complesse operazioni manuali.

4.5 Lettura e scrittura file

Oltre a testare le prestazioni di un runtime WASM/WASI su argomenti di gestione della sicurezza, è importante valutare anche la sua efficienza nella gestione di operazioni di I/O, come la lettura e la scrittura di file. Questo tipo di test può offrire diversi vantaggi tra cui:

- *Miglior comprensione delle prestazioni*: La velocità di lettura e scrittura è

fondamentale per la performance di un'applicazione. Testando quest'area, si ottiene un'idea più completa delle capacità del runtime.

- *Valutazione dell'implementazione di WASI*: WASI definisce le API per l'accesso alle risorse di sistema, inclusa la gestione dei file. Testando la velocità di I/O, si può valutare l'efficienza con cui il runtime implementa queste API e come questa influenza le prestazioni.

- *Ottimizzazione per scenari reali*: Testando la velocità di lettura e scrittura, si possono ottenere informazioni utili per ottimizzare il runtime in base a specifici scenari d'uso.

- *Confronto tra runtime*: Eseguendo lo stesso test su diversi runtime WASM/-WASI, è possibile confrontarne le prestazioni in termini di I/O e identificare il più adatto per applicazioni con esigenze specifiche.

Per ottenere un valore realistico e minimizzare il "rumore" casuale, è stato deciso, come nel caso dei test sulla libreria libsodium, di eseguire il test 100 volte per ogni runtime in modo da aiutare a ottenere una media più precisa. Inoltre è stato utilizzato un file di dimensioni di 50MB per permettere di testare il comportamento del runtime in condizioni realistiche.

I risultati di questo test permettono, quindi, di confrontare le prestazioni di I/O dei diversi runtime WASM/WASI e vedere eventuali differenze nell'implementazione di WASI per la gestione dei file, in modo da dare informazioni utili per la scelta del runtime più adatto al bisogno specifico.

```
const { readFile } = require('node:fs/promises');
const { WASI } = require('wasi');
const { argv, env } = require('node:process');

const wasi = new WASI({
  version: 'preview1',
  args: argv,   // Argomenti della linea di comando forniti al processo
  env,          // Variabili d'ambiente del processo
  preopens: {
    '/local': '.', // Percorso al quale è possibile accedere dal modulo WASM
  },
});
const filename = argv[2];

if (!filename) {
  console.error('Utilizzo: node execute_node.js <nomefile>');
  process.exit(1); // Terminazione del processo segnalando un errore
}

(async () => {
  try {
    // Compilazione del file WASM
    const wasm = await WebAssembly.compile(await readFile(filename));
    // Istanziazione del modulo WASM con gli import di WASI
    const instance = await WebAssembly.instantiate(wasm, wasi.getImportObject());
    // Avvio dell'esecuzione del modulo WebAssembly
    wasi.start(instance);
  } catch (error) {
    console.error(`Errore durante l'esecuzione di  ${filename}:`, error);
  }
})();
```

Listing 7: `execute_node.js`

```c
int main(int argc, char *argv[]) {
    int fd, buffer_size;
    char *buffer;
    clock_t start, end;
    double time_taken;
    // Get buffer size from argument
    buffer_size = strtol(argv[1], NULL, 10); // Convert string arg to integer
    buffer = malloc(buffer_size);
    fd = open("lorem.txt", O_RDONLY); // Open the file

    start = clock(); // Start timing
    while (read(fd, buffer, buffer_size) > 0) {
        // Do nothing; We're just focusing on read speed
    }
    end = clock(); // End timing

    // Calculate elapsed time
    time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("%f\n", time_taken);
    close(fd); // Close the file
    free(buffer); // Release memory
    return 0;
}
```

Listing 8: `read_test.c`

Capitolo 5

Risultati

Contents

In questo capitolo verranno analizzati tutti i vari test svolti, i risultati ottenuti e, quindi, come si comportano diversamente i vari runtime all'interno dei vari test.

5.1 AEAD benchmark

I file `AEAD` (Authenticated Encryption with Associated Data) di libsodium sono un tipo di file crittografato che offre diverse funzionalità come: la crittografia simmetrica, il che significa che, la chiave di crittografia utilizzata per la cifratura,

è la stessa utilizzata per la decifratura, l'autenticazione del messaggio, ovvero un codice di autenticazione MAC, generato per garantire che il messaggio non sia stato modificato dopo la crittografia, e i dati associati, informazioni aggiuntive incluse nel file crittografato, come ad esempio metadati o informazioni di routing.

Gli scenari d'uso dei file AEAD di libsodium sono utilizzati per poter avere delle comunicazioni sicure. autenticando tutti i dati e un'archiviazione sicura.

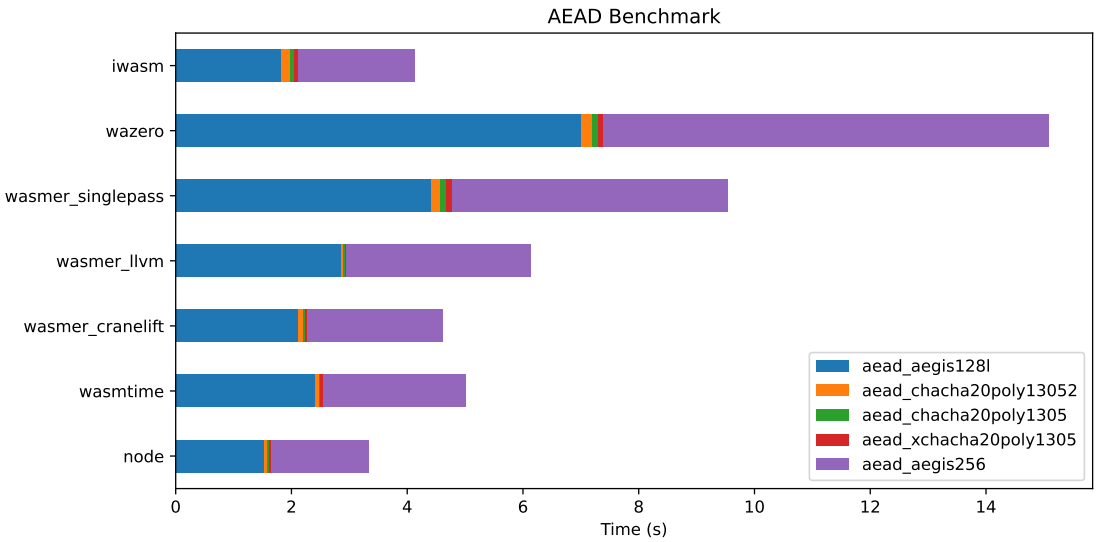


Figura 5.1: AEAD Benchmarks

In questo caso si può notare da figura 5.1 che il runtime più performante risulta essere Node, con tempi di esecuzione inferiori per tutti gli algoritmi testati, mentre quello più lento è sicuramente Wazero. Questo può essere a causa di diversi fattori, come le differenze di implementazione e ottimizzazione per I/O, ovvero Node potrebbe essere più ottimizzato rispetto agli altri in operazioni di lettura e scrittura di file.

5.2 Authentication benchmark

I file di tipo `auth` sono progettati principalmente per garantire l'autenticità dei messaggi attraverso l'utilizzo di un Message Authentication Code (MAC). Questo

codice assicura che i dati non siano stati alterati o manomessi mentre sono in transito o archiviati, tuttavia non offre alcuna garanzia di riservatezza, poiché i dati sono trasferiti in chiaro e, quindi, leggibili da chiunque. I principali utilizzi di questo tipo di file includono sicuramente la verifica dell'integrità dei messaggi, così come dei firmware e la firma digitale.

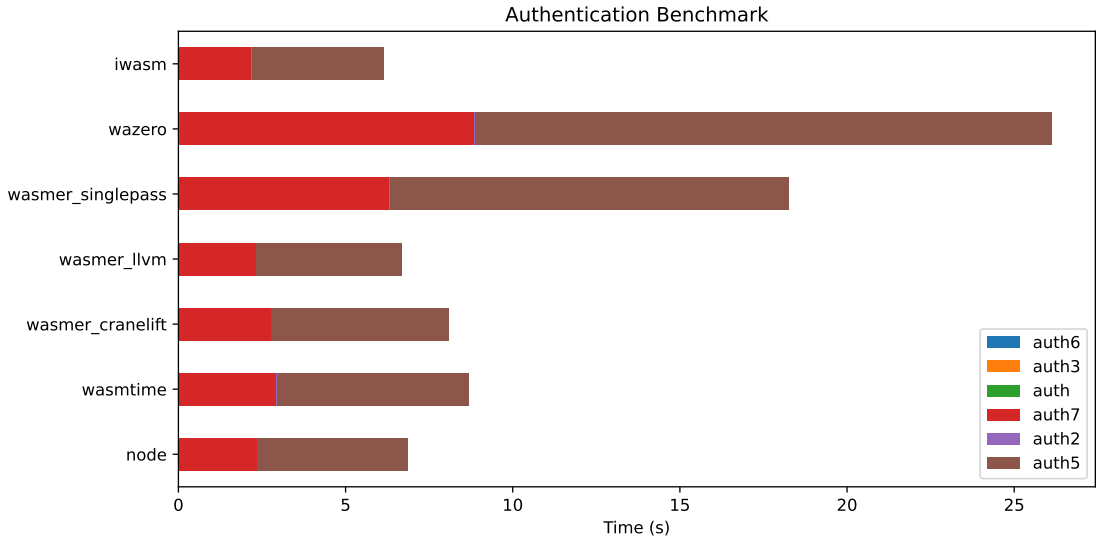


Figura 5.2: Authentication Benchmarks

Da come si può vedere in figura 5.2, i file che richiedono più tempo computazionale sono `auth5` e `auth7`. Entrambi i file generano un codice di autenticazione del messaggio per garantire l'integrità dei dati, solo che il primo crea un MAC da 32 bit, mentre il secondo da 64, motivazione per cui probabilmente risulta essere più lento, ma al contempo più sicuro. Come nel precendente grafico, Wazero rimane il meno performante, mentre il più veloce questa volta risulta essere wasmer_llvm subito seguito da Node e iwasm, che sono di poco più lenti.

5.3 Hashing benchmark

I file di hashing di libsodium sono utilizzati per generare un hash di un messaggio o di un file. Le modifiche al messaggio originale causano un cambiamento riconoscibile dell'hash, rendendolo utile per diverse applicazioni. Il più immediato è la

verifica dell'integrità: I file di hashing possono essere utilizzati per verificare che un file non sia stato modificato, questo perché, se l'hash calcolato per un file non coincide con quello memorizzato, significa che il file è stato modificato.

Un'altra applicazione è l'autenticazione del mittente: i file di hashing possono essere utilizzati per autenticare il mittente di un messaggio. Il mittente può includere l'hash del messaggio con il messaggio stesso. Il destinatario può, quindi, calcolare l'hash del messaggio ricevuto e confrontarlo con quello ricevuto. A quel punto, se gli hash coincidono, è sicuro che il messaggio provenga dal mittente previsto. Ricerca di file duplicati: i file di hashing possono essere utilizzati per identificare file duplicati, infatti se due file hanno lo stesso hash, è molto probabile che siano identici.

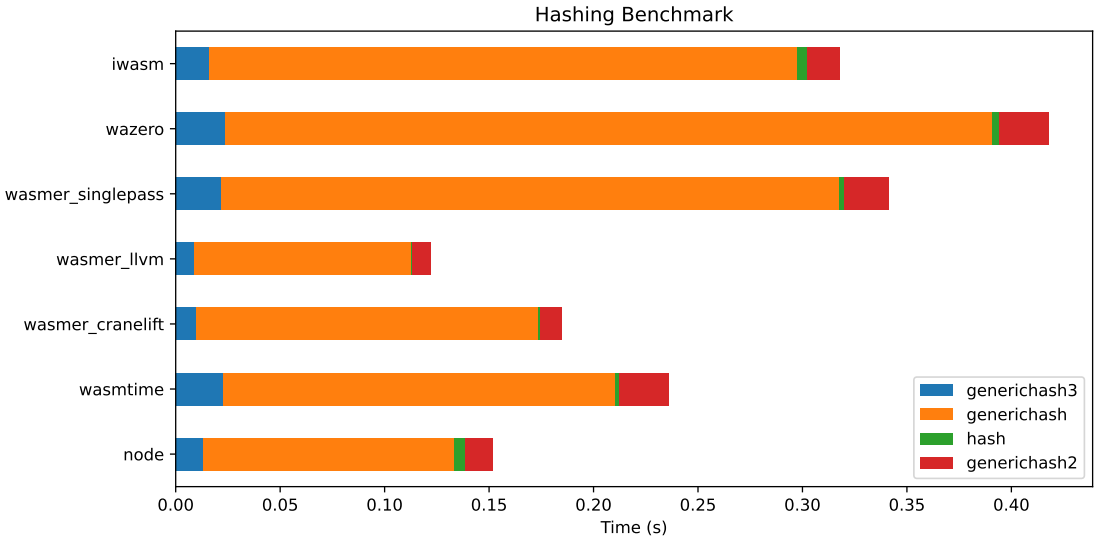


Figura 5.3: Hashing Benchmarks

Le differenze principali tra `generichash3`, `generichash2` e `generichash` riguardano l'implementazione di funzioni all'interno della famiglia Blake2b, che è una serie di funzioni di hash crittografico progettate per essere veloci ed efficienti su molte piattaforme.

Il file `generichash3` utilizza le funzioni `crypto_generichash_blake2b_*` da una libreria come libsodium e sfrutta ampiamente il concetto di *salt*, ovvero valori casuali che vengono aggiunti agli input prima di essere processati da una fun-

zione di hash. Al contrario, il file `generichash2` utilizza le funzioni più generali `crypto_generichash_*` e `generichash` sembra essere un'implementazione più snella della precedente, focalizzandosi sull'hashing di base, senza le funzionalità aggiuntive di sicurezza come il *salt*. Queste differenze potrebbero rallentare o velocizzare i tempi di esecuzione di ciascun file, motivo per cui si nota che il file `generichash` è quello che richiede più tempo fra tutti, nonostante la somma di tutti i tempi di esecuzione non richieda dei tempi elevati.

5.4 Metamorphic benchmark

Questo file contiene un insieme di funzioni di test denominate con il prefisso *mm_*. Ogni funzione di test si concentra nel verificare diverse primitive crittografiche all'interno della libreria libsodium. In particolare, il codice implementa una forma di *testing metamorfico*. Questo tipo di testing non si concentra su coppie specifiche di input/output, ma controlla se diversi modi di eseguire la stessa operazione crittografica producono risultati uguali, aiutando a scoprire degli errori di implementazione. Bisogna tener conto, quindi, che questo codice non è progettato per un utilizzo crittografico pratico. Al contrario è un metodo di testing implementato all'interno della libreria libsodium per aumentare la correttezza di varie funzioni crittografiche, garantendo che i loro risultati siano consistenti, indipendentemente dal modo in cui vengono utilizzate.

5.5 One-time authentication benchmark

Il file `onetimeauth7` contiene una funzione di test progettata per mettere alla prova le funzioni `crypto_onetimeauth` e `crypto_onetimeauth_verify`, provenienti dalla libreria libsodium. Lo scopo principale è verificare la resistenza alla falsificazione dei codici di autenticazione monouso, assicurando l'autenticità dei messaggi e la protezione contro i tentativi di falsificazione. Ciò significa che, modificare il messaggio, produrrà una successiva verifica non valida quando viene utilizzata la chiave originale.

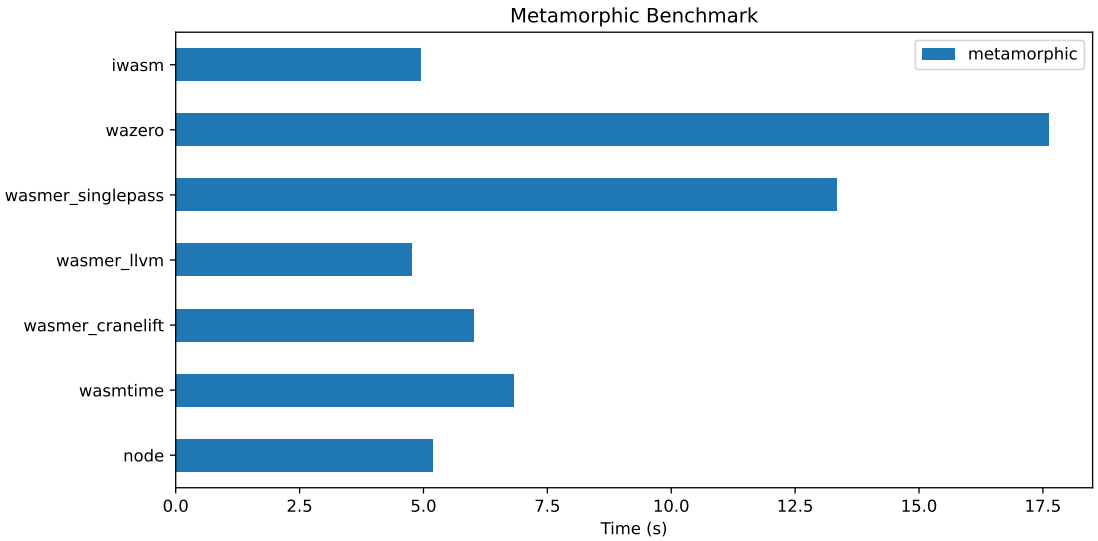


Figura 5.4: Metamorphic Benchmarks

5.6 Benchmark scambio e derivazione di chiavi

I file esaminati in questo test sono essenziali per garantire la robustezza, l'affidabilità e la sicurezza delle funzioni crittografiche, implementate all'interno della libreria libsodium. In particolare, il file `KDF` è dedicato a testare la "Key Derivation Function", in cui vengono generati e verificati sottochiavi di diverse lunghezze. Similmente, il file `KDF_HKDF` è progettato per testare l'"HMAC-based Key Derivation Function" e viene verificato che, le chiavi derivate, siano generate correttamente. Infine il file `KX` si concentra sul testing del meccanismo di "Key Exchange". Questo file simula lo scambio di chiavi tra client e server, verificando che entrambe le parti generino la stessa chiave di sessione condivisa e, inoltre, vengono anche testati scenari di fallimento con chiavi pubbliche non valide.

Analizzando la figura 5.6, è evidente che il runtime più lento, come anche nel caso precedente, è *wasmer_singlepass*, seguito immediatamente da *wazero*, dimostrando che sono entrambi i più lenti dal punto di vista computazionale durante tutti i test svolti. Al contrario, il più veloce risulta essere *wasmer_llvm*, seguito da *Node.js*, ma questa volta non si osserva la presenza di *iwasm* all'interno della lista dei runtime più rapidi.

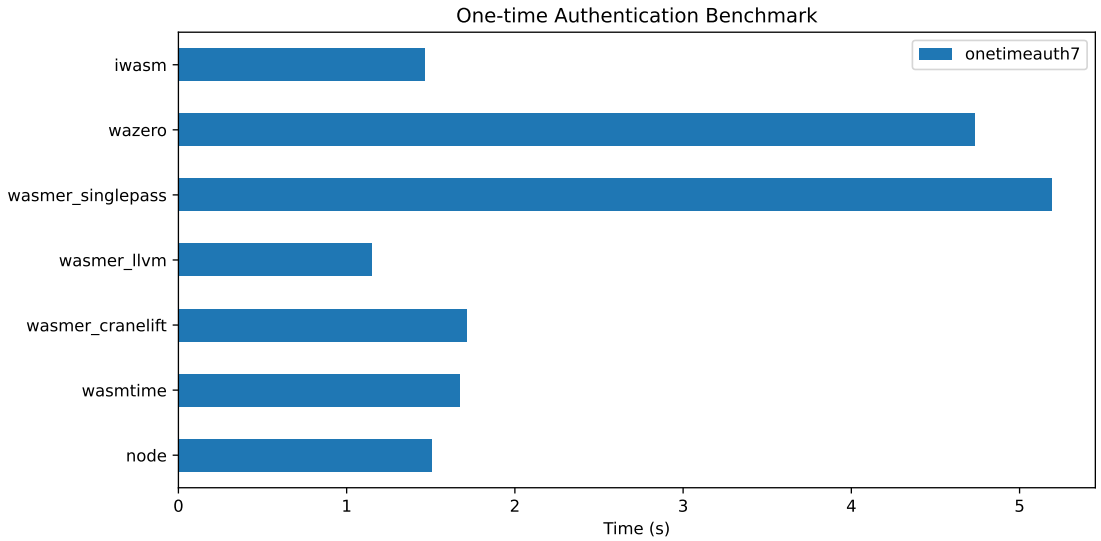


Figura 5.5: Benchmark autenticazione One-time

Un'altra osservazione rilevante è che il file `kx` risulta essere quello che richiede più tempo per venir testato su ogni runtime. Questo potrebbe essere dovuto al fatto che, all'interno del file, vengono eseguiti anche test relativi a scenari di fallimento, che richiedono una maggiore elaborazione e verifica dei risultati rispetto ad altri tipi di test.

5.7 Confronto con il codice nativo

Dopo aver valutato le performance dei diversi runtime di WebAssembly, è stato fatto un confronto diretto tra le loro performance e le performance degli stessi file, ma compilati in codice macchina per essere eseguiti in modo nativo. L'obiettivo di questo confronto è quello di analizzare quanto l'overhead introdotto dai vari runtime di Wasm impatti sul tempo di esecuzione.

Dalla Figura 5.7 si può notare che la differenza di prestazioni tra il runtime più performante, *Node.js*, e il codice nativo è importante. Tuttavia, ad eccezione dei due runtime peggiori, *wazero* e *wasmer_singlepass*, tutti gli altri, anche se non raggiungono ovviamente le prestazioni del codice nativo, rappresentano comunque

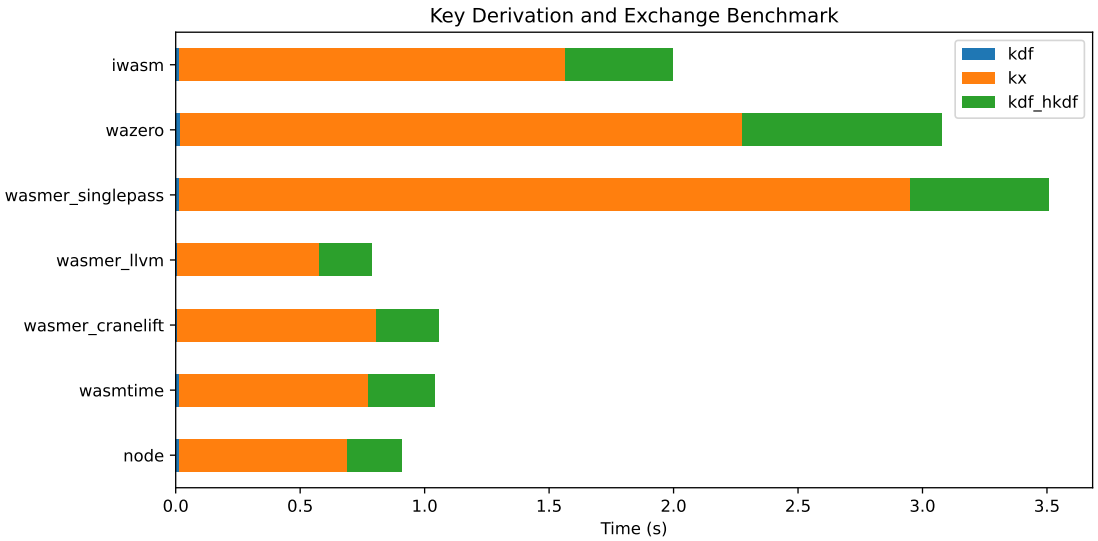


Figura 5.6: Benchmark scambio e derivazione di chiavi

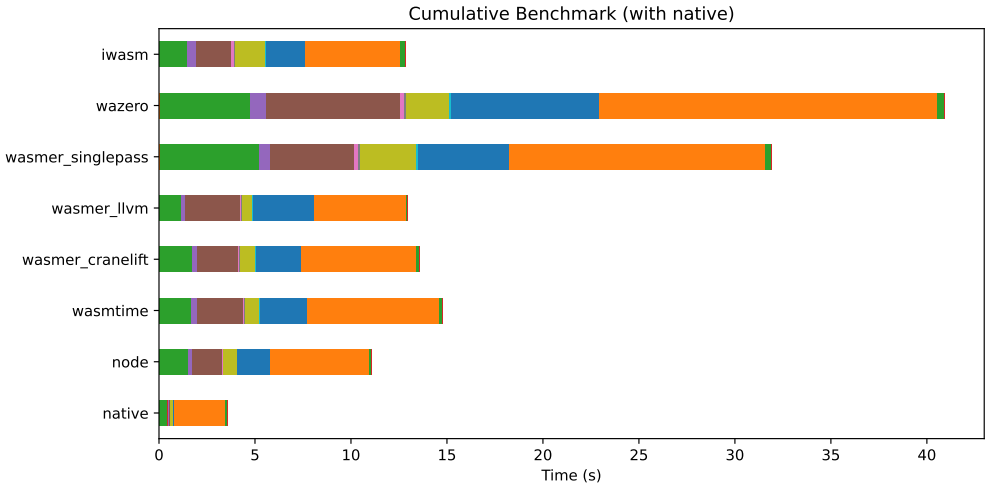


Figura 5.7: Confronto runtime con esecuzione nativa

una valida alternativa. Questo suggerisce che, nonostante l'overhead introdotto dai runtime di Wasm, essi possono offrire buone prestazioni in molti casi.

Nella Figura 5.8 vengono confrontati i tempi di esecuzione del miglior runtime Wasm di ciascun test con quelli del codice nativo. Ad eccezione delle funzioni di `AEAD`, che presentano tempi di esecuzione talmente elevati anche con il miglior

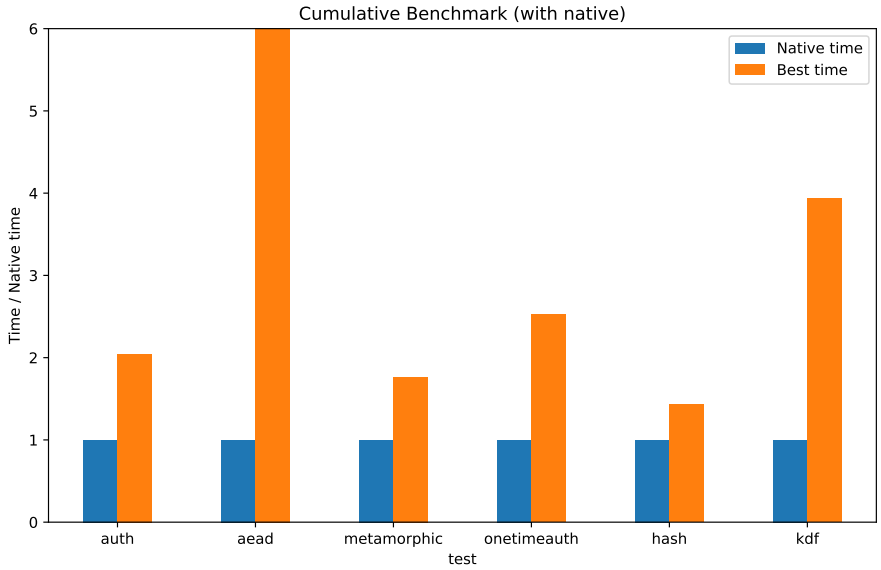


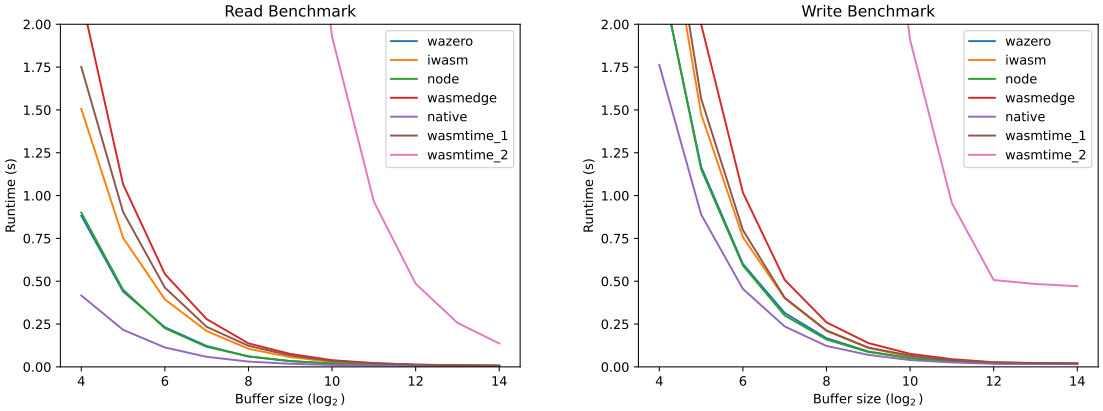
Figura 5.8: Confronto nativo con miglior runtime

runtime, del quale in figura non è stato riportato il tempo preciso per motivi grafici (circa 30 volte in più rispetto al codice nativo), tutti gli altri test mostrano tempi di esecuzione relativamente vicini a quelli del codice nativo. In particolare, per i test di `hash`, le prestazioni del miglior runtime di Wasm sono molto simili a quelle del codice nativo, mostrando che per queste operazioni l'overhead introdotto dai runtime di Wasm è praticamente trascurabile.

5.8 Benchmark di lettura e scrittura file

Per testare ulteriormente i vari runtime e la loro implementazione di WASI, sono stati fatti dei semplici test di lettura e scrittura di file, misurando in questo modo il loro tempo di esecuzione in funzione della dimensione del buffer.

La figura 5.9 mostra i tempi medi di lettura e scrittura di file per differenti runtime WASI, in funzione della dimensione del buffer utilizzato. Per buffer di piccole dimensioni, si può notare una differenza significativa nei tempi di esecuzione tra il codice nativo e i vari runtime WASI, dato che ogni chiamata alle funzioni `read()` e `write()` introduce un overhead, che fa la differenza in particolare su



(a) Lettura file

(b) Scrittura file

Figura 5.9: Benchmark di lettura e scrittura file

buffer piccoli. Il codice nativo, al contrario, è ottimizzato per operazioni di I/O su file, quindi riesce a minimizzare tale overhead.

Con l'aumentare della dimensione del buffer invece, i tempi di esecuzione dei diversi runtime Wasm, tendono a convergere verso lo stesso valore del codice nativo, grazie alla riduzione del numero di chiamate alle funzioni `read()` e `write()`, che va in questo modo a diminuire l'overhead totale. L'unica eccezione è il runtime Wasmtime con l'implementazione della WASI preview 2 (`wasmtime_2` in figura), che, come si può notare anche dai grafici precedenti, non è ancora molto stabile. Ció si riflette nei test con dei risultati pessimi, non paragonabili a nessuno degli altri runtime.

La scelta del runtime WASI più adatto dipende, però, da diversi fattori, tra cui quindi la dimensione dei buffer utilizzati, il tipo di operazioni di I/O da compiere e le esigenze dell'applicazione. Da questi test, si capisce che è importante considerare la dimensione del buffer, poiché può influenzare in modo importante le prestazioni di lettura e scrittura di file in caso si utilizzi WASI.

In conclusione, bisogna ricordare, però, che le prestazioni dei runtime WASI sono in continuo miglioramento, perciò la scelta del runtime dipende dalle specifiche esigenze dell'applicazione e dai parametri di utilizzo.

5.9 Benchmark socket

È stata fatta una ricerca sull'implementazione delle funzionalità di rete, utilizzando le socket nei diversi runtime WebAssembly, dato che potrebbe risultare utile per valutare l'efficacia dei vari runtime. Per questo è stato scritto un codice in linguaggio C, che utilizza le primitive POSIX per la gestione delle socket come `socket()`, `accept()`, `listen()` al fine di stabilire una connessione TCP tra due endpoint che scambiano messaggi per un intervallo di tempo costante, in modo da misurare la velocità di trasmissione, il throughput e la latenza.

Tuttavia, durante il processo di test è emerso un problema, ovvero, nonostante aver provato a compilare ed eseguire il file di test con svariate configurazioni differenti, non si riuscivano a connettere le due applicazioni. Questo potrebbe essere stato causato dal fatto che, le funzionalità delle socket di POSIX, appartengono a un modulo specifico di WASI che non è sempre implementato in modo uniforme nei diversi runtime Wasm. Un altro possibile problema potrebbe essere quello dell'incompatibilità con il sistema operativo sottostante (WSL). In alcuni casi, le funzionalità di rete non sono implementate affatto, rendendo impossibile l'esecuzione del codice di test. Analizzando la documentazione dei runtime Wasm testati, infatti, è emerso che solo alcuni di essi implementano le funzionalità di rete del test.

Da queste considerazioni, si può dire che il mondo dei runtime Wasm è in continua evoluzione e in pieno sviluppo, con miglioramenti continui, infatti da questo test si vede come l'implementazione delle funzionalità di rete non è ancora completa. Per questo, la ricerca e lo sviluppo sono fondamentali per il futuro delle applicazioni web.

Conclusione

In questa tesi è stato studiato WebAssembly un linguaggio creato nel 2015 che permette di eseguire codice in modo efficiente e sicuro, inizialmente pensato all'interno del browser web. Successivamente, è stato discusso WebAssemby System Interface, spiegando che è un'interfaccia standard che permette alle applicazioni Wasm di comunicare direttamente con le risorse di sistema, con particolare attenzione alla sicurezza. Sono stati anche introdotti tutti i runtime che sono stati utilizzati nei test. Questo perché l'obiettivo di questa tesi è la valutazione delle performance di vari runtime WebAssembly e delle loro implementazioni di WebAssembly System Interface, confrontandoli in base a diversi parametri e scenari d'uso, facendo test sulla libreria libsodium. I test condotti hanno evidenziato come WebAssembly sia una tecnologia ancora in fase di sviluppo e presenti alcune criticità, come la gestione delle socket e la performance non ottimale di alcuni runtime, come wasmer_singlepass e wazero. Nonostante le criticità, WebAssembly ha dimostrato un grande potenziale in termini di portabilità e velocità di esecuzione, aprendo la strada a nuove possibilità per lo sviluppo di applicazioni web performanti e complesse.

Grazie allo sviluppo di WASI, le applicazioni scritte in WebAssembly possono ora comunicare direttamente con le risorse di sistema, come file e socket, e possono essere eseguite al di fuori del browser.

Essendo però WASI un'interfaccia, spetta a ogni runtime implementare tutte le varie funzionalità, e, nonostante possa essere questo un vantaggio, consentendo a ogni runtime un'implementazione specifica della propria architettura, diventa a volte uno svantaggio a causa delle implementazioni eterogenee e, a volte, non

complete di WASI. Questo può portare a situazioni in cui lo stesso codice, eseguito con runtime diversi, produce risultati diversi o non funziona affatto, come nel caso di connessione tra due applicazioni con l'uso di socket, discusso precedentemente. Per questo, degli sviluppi futuri potrebbero concentrarsi su queste criticità e sulla loro evoluzione, andando ad eseguire nuovi test.

Nonostante ciò, quasi tutti i runtime Wasm analizzati hanno mostrato delle performance decisamente promettenti sui test della libreria Libsodium. Tra questi ultimi, i migliori e i più performanti sono risultati essere Node.js, iwasm e Wasmer con il compilatore LLVM.

Bibliografia

[1] Lin Clark. *Standardizing WASI: A system interface to run WebAssembly outside the web*. https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/.

[2] *iwasm*. https://github.com/bytecodealliance/wasm-micro-runtime.

[3] *Libsodium*. https://github.com/jedisct1/libsodium.

[4] *LLVM*. https://llvm.org/.

[5] *NaCl: Networking and Cryptography library*. http://nacl.cr.yp.to/.

[6] *Node.js*. https://nodejs.org/en.

[7] Partha Ray. «An Overview of WebAssembly for IoT: Background, Tools, State-of-the-Art, Challenges, and Future Directions». In: *Future Internet* 15 (ago. 2023), p. 275. DOI: 10.3390/fi15080275.

[8] *WABT: The WebAssembly Binary Toolkit*. https://github.com/WebAssembly/wabt.

[9] *WASI Preview 2*. https://github.com/WebAssembly/WASI/blob/main/preview2/README.md.

[10] *Wasmedge*. https://wasmedge.org/.

[11] *Wasmer*. https://wasmer.io/.

[12] *Wasmtime*. https://wasmtime.dev/.

[13] *Wazero*. https://wazero.io/.