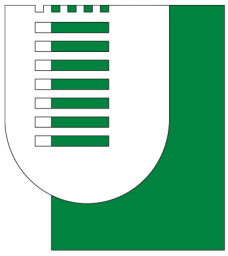

TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

DIPARTIMENTO di INGEGNERIA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

TESI DI LAUREA

Protezione della proprietà intellettuale mediante offuscamento basato su virtualizzazione

Relatore:

Prof. Pellegrini Alessandro

Candidato:

Caliandro Pierciro

Anno Accademico 2021/2022

Indice

Per nonno Michele, la mia Famiglia ed i miei Amici

Capitolo 1

Introduzione e scopo

In questa tesi viene presentato lo studio e l'implementazione di una macchina virtuale (*virtual machine*, VM), per la protezione della proprietà intellettuale (*intellectual property*, IP), legata ad artefatti software, senza la necessità di avere a disposizione il relativo codice sorgente.

1.1 Introduzione

La protezione della proprietà intellettuale riguarda la possibilità di un individuo di poter detenere i diritti legali di una creazione dell'intelletto in campo scientifico, artistico ed industriale.

Nel caso specifico degli artefatti software, spesso ciò che viene distribuito non è il codice sorgente, bensì solo il file eseguibile prodotto finale della compilazione.

Ciononostante, vi sono diverse tecniche che possono essere messe in atto da attori malevoli per ricostruire tali file sorgenti, suddivisibili in due macro-categorie:

- **Analisi statica** del codice: viene eseguita un'analisi del file eseguibile con strumenti di *disassemblaggio*, quali ad esempio [6];

- **Analisi dinamica** del codice, dove viene analizzato il comportamento del programma eseguendolo e tenendolo sotto controllo con strumenti di ***debugging***.

Alcune delle motivazioni che possono guidare un attore malevolo a voler violare la proprietà intellettuale sono:

- di natura economica, ad esempio per rivendere una soluzione come propria;
- per aggirare protezioni inserite all'interno del programma, come licenze di acquisto;
- per recuperare segreti presenti all'interno del codice, come chiavi di (de)cifratura;
- per scoprire vulnerabilità intrinseche nel codice ed usarle a proprio vantaggio.

D'altra parte, sono presenti in letteratura differenti tecniche di difesa attuate da chi scrive il software.

Fra queste, vi sono tecniche che rientrano nell'area dell'**offuscamento del codice**, più nello specifico nella realizzazione di una macchina virtuale per eseguire il programma originale.

Il codice viene eseguito in una ***sandbox***, quindi un ambiente controllato all'interno del programma macchina virtuale.

Questo metodo rende le tecniche di analisi statica (mediante *disassembler*) e dinamica (mediante *debugger*) del codice molto più ostiche.

Difatti, quando l'attore malevolo prova ad analizzare il disassemblato dell'eseguibile, non si trova più di fronte al codice macchina del programma originale,

bensì ad una sequenza di byte apparentemente senza senso ma che, quando interpretata dalla macchina virtuale, implementa la stessa logica del programma originale.

Senza avere a disposizione i sorgenti di tale macchina virtuale, le operazioni necessarie per ricostruire il programma originale diventano molte di più e richiedono un *effort* maggiore.

Infatti, un utente malevolo che tenta di applicare tecniche di *reverse engineering* al file binario risultante dall'applicazione del *layer* di virtualizzazione, deve effettuare il *reversing* di tutto il codice della macchina virtuale stessa, per poi eventualmente concentrarsi sul programma originale che viene eseguito al suo interno.

L'approccio usato in questo lavoro, ovvero quello di operare direttamente a livello del file binario a valle della compilazione, permette di integrare qualsiasi libreria di terze parti usata: difatti, sarà sufficiente andare a tradurre il codice della libreria presente nel file binario nel *bytecode* eseguibile dalla macchina virtuale.

Questo offre un grande vantaggio in diverse situazioni, come ad esempio **integrazione di librerie di crittografia**: se il programma da proteggere utilizza delle librerie crittografiche, l'attore malevolo può essere interessato a vedere l'interazione con tali librerie all'interno del programma.

Con l'approccio proposto, lo stesso codice della libreria può essere inserito nel codice virtualizzato, andandolo a "confondere" con il resto ed oscurando tale interazione che sarebbe altrimenti esplicita.

Un altro caso di interesse può essere l'integrazione della libreria di virtualizzazione all'interno degli *step* di una *pipeline* di *Continuous Integration/Continuous Development* arbitrariamente complessa: difatti, l'operazione di virtualizzazione può avvenire a valle delle fasi di compilazione e *linking* del programma, producendo così il binario virtualizzato alla fine di tutti gli *step*.

Ulteriore aspetto interessante nella scelta di lavorare a livello di binario è che questo permette di essere agnostici rispetto al sistema operativo su cui verrebbe eseguita la VM.

Infatti, oltre alla macchina virtuale, è stata realizzata un'applicazione di transcodifica che effettua la mappatura dall'ISA dell'architettura per cui è stato compilato il programma verso l'ISA della macchina virtuale.

La macchina virtuale offre un'interfaccia verso le chiamate di sistema differente a seconda del sistema operativo sottostante e tale applicazione di transcodifica può essere adattata per supportare le diverse convezioni per le chiamate di sistema in modo da poter sempre convertire il binario senza preoccuparsi del Sistema Operativo su cui esso verrà eseguito.

1.2 System and Threat model

Il ***system and threat model*** preso in considerazione in questo lavoro presenta diverse assunzioni:

I Il codice sorgente del programma eseguito nella macchina virtuale non è reso disponibile, l'unica informazione a disposizione è quindi formata dal file eseguibile distribuito;

II L'attaccante ha possibilità di eseguire tale file eseguibile un numero arbitrario di volte monitorando l'esecuzione mediante l'uso di *debugger*;

III Non conoscendo i dettagli implementativi dell'ISA della macchina virtuale, tali molteplici esecuzioni servono a ricostruire queste informazioni mancanti, per poter poi risalire al codice disassemblato del programma oggetto dell'attacco;

IV l'attaccante ha inoltre la possibilità di vedere differenti file binari protetti da macchina virtuale.

Un esempio di applicazione della VM per proteggere la IP è il caso in cui nel codice vi sono segreti o chiavi di cifratura *hard-coded*: mediante la virtualizzazione, si può offuscare l'uso esplicito di tali segreti, in modo da rendere meno immediato per un attaccante capire dove essi vengono usati ed il loro valore.

Un altro esempio, come detto in Sezione 1.1, è l'offuscamento delle librerie crittografiche: utilizzando la VM, un attaccante che cerca di effettuare il *reverse engineering* del codice per vedere le interazioni con tali librerie a *run time* non vede direttamente la chiamata alla libreria bensì del codice eseguito come se fosse una qualunque funzione.

Inoltre, operativamente lavorare a livello di file binario offre vantaggi fra cui:

- un minore *overhead* nel tempo necessario a convertire il programma originale nel *bytecode* interpretato dalla VM;
- un semplicità maggiore nell'applicazione di transcodifica, in quanto la traduzione non parte dal codice sorgente, bensì dall'Assembly dell'ISA target, per poi essere direttamente convertito in quello della VM.

Capitolo 2

Lavori collegati

In questo capitolo vengono analizzate le soluzioni proposte in letteratura, raffrontandole a quanto implementato in questo lavoro di tesi.

È utile ricordare le caratteristiche implementative del lavoro svolto per confrontarlo con quanto presente in letteratura:

I l'input iniziale della macchina virtuale è il file eseguibile in formato binario;

II la VM virtualizza i programmi indipendentemente dal sistema operativo sottostante;

III vi è la *reimplementazione* della libreria standard.

2.1 Analisi della letteratura

Lo sviluppo di soluzioni basate su macchina virtuale è una delle tecniche di offuscamento del codice, fra le tante presenti in letteratura, più promettenti [10].

Per rendere più complessa l'analisi dinamica mediante *debugger*, spesso vengono utilizzate tecniche per mutare dinamicamente il comportamento della *virtual*

machine stessa: l'ISA interno è infatti collegato agli ***opcode*** definiti per rappresentare le istruzioni, quindi se tale configurazione cambia dinamicamente da esecuzione ad esecuzione, diventa più complesso per l'attore malevolo riuscire a seguire il flusso operativo e ricostruire l'ISA della macchina virtuale.

Un esempio grafico è mostrato in Figura 2.1: in questo caso, ogni volta che avviene il processo di traduzione dell'ISA e compilazione dell'eseguibile virtualizzato finale, cambia l'associazione fra gli *opcode* e le istruzioni.

Questo può avvenire anche in questo lavoro e fa si che, se anche un attaccante entra in possesso di più file eseguibili compilati usando lo stesso *virtualizzatore* e tenta di incrociare le informazioni ottenute a seguito dell'analisi di uno di essi, essendo cambiato il *mapping* fra *opcode* ed istruzioni tale conoscenza non potrà essere sfruttata.

Ulteriormente, all'interno del lavoro proposto ogni istruzione può avere un certo numero di byte di *padding* generati casualmente.

Questo ha l'effetto di confondere gli *opcode* associati alle istruzioni della VM, in quanto uno stesso byte può essere:

- un byte che effettivamente rappresenta un'istruzione dell'ISA;
- un byte di *padding* generato casualmente.

Vi sono poi ulteriori strategie per contrastare l'analisi dinamica: in [1], viene proposto di proteggere la VM salvando il codice che si occupa del *dispatching* degli *opcode* dell'ISA virtuale all'interno della memoria di CPU.

Tale approccio, differentemente da quanto presentato in questo lavoro di tesi, parte dai file sorgente dell'applicativo.

In [7] viene proposta una soluzione cambia l'*execution path* ad ogni esecuzione della macchina virtuale, combinando due tecniche:

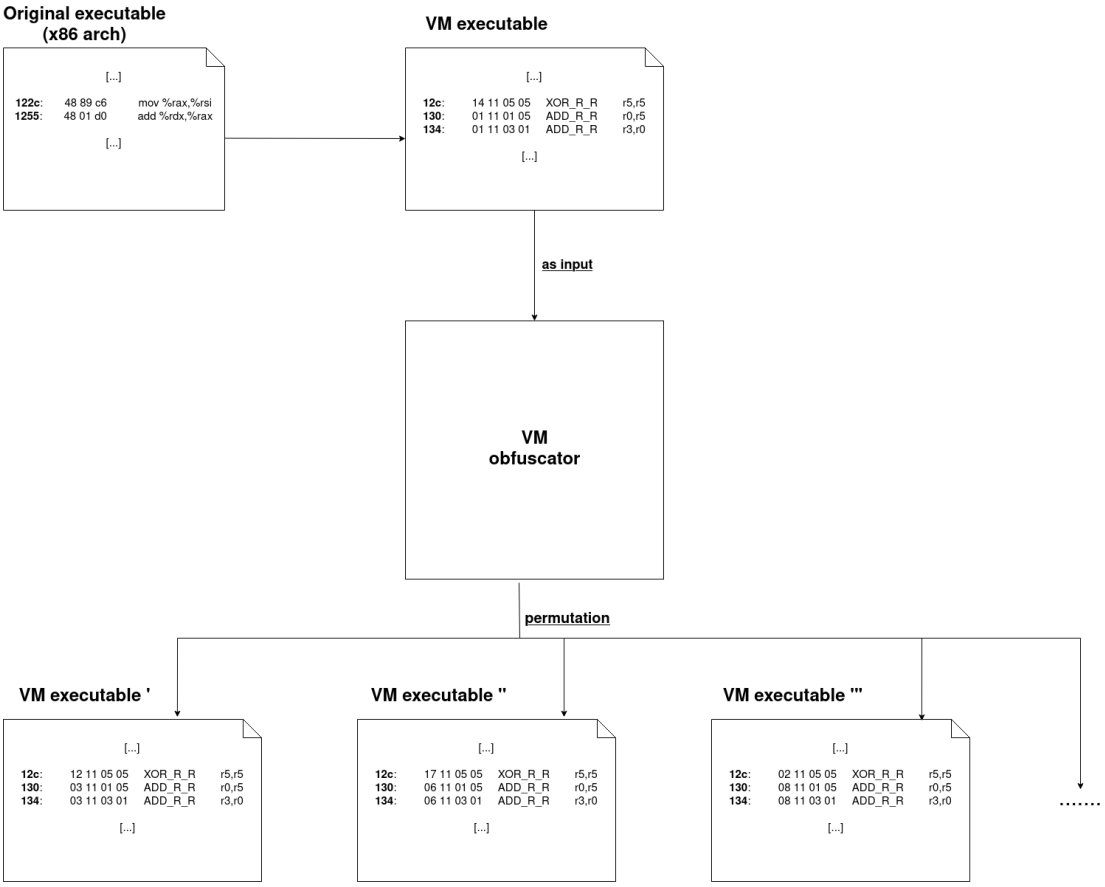

Figura 2.1: Esempio di permutazione degli *opcode* della macchina virtuale

- utilizzo di uno *scheduler* di istruzioni dinamico che segue diversi *execution path* ad ogni esecuzione;
- utilizzo di uno schema multi-VM, in modo che differenti porzioni di codice siano protette usando insiemi di istruzioni virtuali differenti.

Tale approccio prende in input il file binario del programma, ma a differenza di questo lavoro di tesi non vi è la *reimplementazione* della `libc`.

In [5] viene proposto un meccanismo di offuscamento basato su molteplici stadi, dove viene estesa la classica traduzione da ISA originale ad ISA virtuale:

- viene generato un numero random n;
- dato il programma originale P, vengono calcolate n copie $\{P_0, ..., P_n\}$ usando le chiavi $\{K_0, ..., K_n\}$;
- gli offuscamenti P_i sono calcolati mediante la funzione Obf come segue

$$P_{i+1} = \text{Obf}(P_i, K_i) \tag{2.1}$$

;

- per quanto riguarda le chiavi K_i, queste sono generate mediante una funzione f che prende come input dei dati del programma P_i o il suo *control flow graph*, nel seguente modo

$$K_i = f(P_i) \tag{2.2}$$

tale funzione f ha le proprietà di una funzione di *hash* (non invertibilità, anti-collisione, univocità).

Anche in questo caso, come per [7], la traduzione avviene partendo dal file binario ma non vi è integrazione con le librerie standard.

Un meccanismo di difesa ulteriore [15] consiste nel cifrare il codice del programma eseguibile, andando poi a *run time* a decifrarlo quando necessario.

Tale approccio prevede anche tecniche per la protezione delle zone di memoria critiche in cui vengono salvati i metadati per la gestione del codice cifrato, come ad esempio le chiavi di decifratura, ma a differenza di questo lavoro di tesi parte da un input formato dai file sorgenti del programma da proteggere.

Un approccio di protezione della VM è proposto in [12], dove viene messa in primo piano la sicurezza della VM, ottenuta usando 2 nuovi insiemi di istruzioni:

- ***Tamper Proofing Instructions (TPIs)***, che estendono le istruzioni virtuali della macchina aggiungendo delle informazioni di controllo come indirizzi di inizio e fine delle sezioni di codice protetto e *checksum* del codice per evitare azioni illecite;
- ***Anti-Debug Instructions (ADIs)***, per proteggere l'ambiente in cui è eseguita la VM.

Tale approccio considera come input file eseguibili in formato PE, quindi ha come target soltanto i sistemi operativi Windows.

In [2] viene proposto un meccanismo di offuscamento basato sul paradigma della *Return Oriented Programming* (ROP): questa tecnica permette di codificare comportamenti arbitrari all'interno del programma, andando a creare delle catene di esecuzione, le *ROP chains*, formate da pezzi di codice, detti *gadget*.

Ognuno dei *gadget* termina con un'istruzione di `ret`, che da il controllo al gadget successivo nella catena.

L'approccio di offuscamento proposto consiste nell'implementazione di un'applicazione di riscrittura, che similmente al lavoro svolto in questa tesi, opera a livello di binario: tale applicazione riscrive le funzioni da proteggere come catene ROP auto-contenute, salvate nella sezione dati del file binario.

Esistono poi prodotti commerciali che sono lo stato dell'arte che applicano meccanismi di virtualizzazione mediante VM per proteggere la IP [16, 14] ed evitare la distribuzione di copie pirata del software, ad esempio nel caso di prodotti dell'industria dei videogiochi [4, 11].

Capitolo 3

Analisi della soluzione proposta

Basandosi su quanto definito nel *system and threat model* in Sezione 1.2, per questo lavoro è stata sviluppata una soluzione che andasse ad offuscare i file eseguibili interpretati nella VM.

In particolare, si ci è focalizzati su applicazioni compilate per architetture x86 a 64 bit e di tutte le applicazioni considerate si è supposto di non avere a disposizione il codice sorgente.

3.1 Analisi ad alto livello della soluzione

L'architettura di alto livello per la soluzione proposta è mostrata in Figura 3.1: come detto in precedenza, il primo passo è convertire il binario `x86_64` in *bytecode*, che verrà poi caricato nella macchina virtuale, andando a separare le istruzioni dai dati.

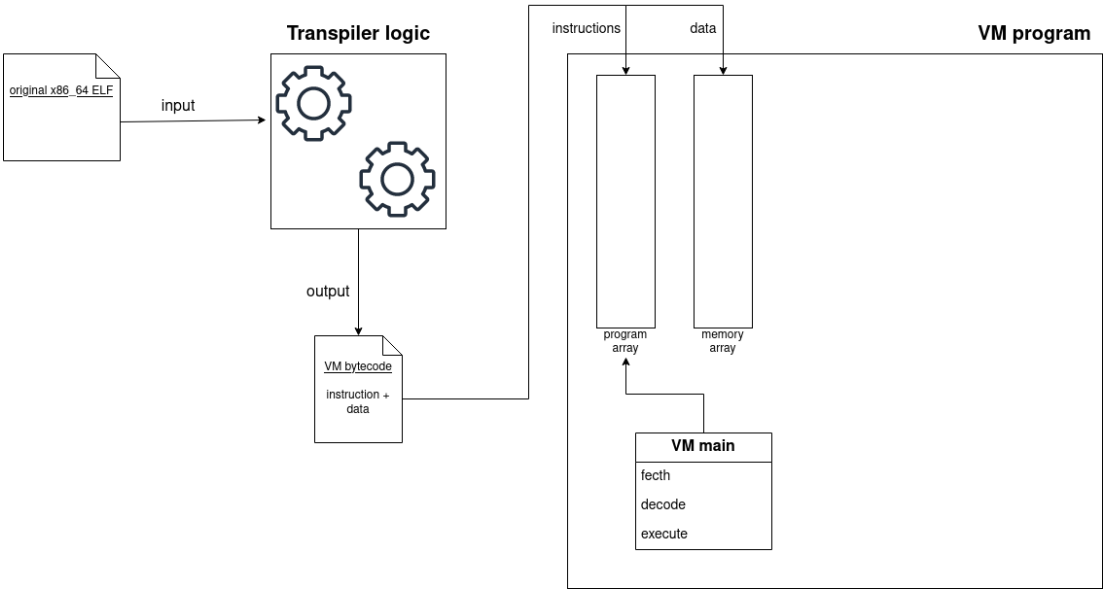

Figura 3.1: Architettura ad alto livello della VM

3.1.1 ISA virtuale

Il primo aspetto considerato per ottenere l'offuscamento del binario è stata la definizione di un ISA virtuale all'interno della VM. Ciascuna istruzione può contenere i campi mostrati in Figura 3.2, ovverosia:

- `opcode`: l'*opcode* è formato da 8 bit. Quando viene letta un'istruzione, il primo byte sarà quindi l'*opcode*, che verrà suddiviso come segue:
  - i primi 5 bit identificano l'istruzione da eseguire;
  - il bit più significativo pari ad 1 identifica il numero di byte di *padding*. Ad esempio, se viene letto il byte `0x4a`:
    - `0xa` identifica l'istruzione, che è una `LOAD_R_I`, che effettua il caricamento dell'immediato sorgente all'interno del registro destinazione;
    - `0x4` identifica il *padding*, ovvero 2 byte.

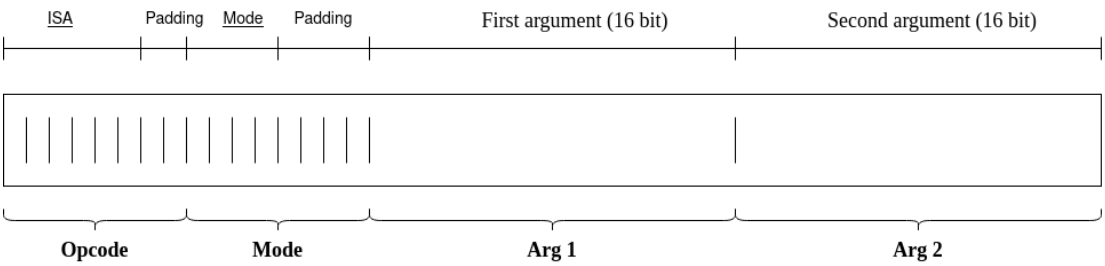

Figura 3.2: Campi di un'istruzione della VM (i campi sottolineati sono obbligatori)

- **`mode`**: composto da 8 bit. Quando viene letto tale byte (dopo aver saltato eventuali byte di *padding* dovuto all'`opcode`), i primi 4 bit forniscono gli operandi che verranno usati nell'operazione, per un totale di 4 combinazioni:
  - `OP_NOARG`: non viene considerato alcun operando;
  - `OP_REG`: l'operando è un registro;
  - `OP_MEM`: l'operando è la memoria;
  - `OP_IMM`: l'operando è un immediato;

  Successivamente, prima di ogni operando, possono esserci fino a 3 *bytes* di *padding* dati dai rimanenti 4 bit, rispettivamente 2 bit per ogni operando;
- **`Arg 1` ed `Arg 2`**: i due operandi (se la `mod` non è 0), ciascuno di 16 bit.

Questa scelta implementativa, ovvero di aggiungere un certo numero di byte di *padding* che possono variare ed essere generati in maniera casuale, è uno degli aspetti chiave per contrastare i tentativi di analisi dinamica volti a ricostruire l'ISA della macchina virtuale: infatti, potendo generare byte casuali che rappresentano degli *opcode* legittimi, questi vanno impediscono di distinguere chiaramente se tale byte sia appunto l'*opcode* dell'istruzione correntemente eseguita o no.

Lo stesso vale anche per il *mapping* che sussiste per la mode e per gli argomenti: potendo avere un numero arbitrario di byte di *padding* prima di essi, non è più deterministico che a seguito del byte di *opcode* vi siano, rispettivamente, il byte per la mode ed i due operandi ed inoltre i valori random potrebbero risultare nella traduzione, da parte di un avversario attaccante, di una tripla <`opcode, mode, arg_1, arg_2`> diversa da quella originale.

Come indicato in Figura 3.1, il primo passo è infatti la conversione dal Assembly `x86_64` in *bytecode* della macchina virtuale.

Differenza fra architettura RISC e CISC ed approccio utilizzato

L'ISA x86 è un ISA **CISC** (Complex Instruction Set Computer), mentre l'ISA offerto dalla macchina virtuale è di tipo **RISC** (Restricted Instruction Set Computer).

Fra le due architetture vi sono svariate differenze, fra cui ad esempio:

1. In architetture CISC, un'istruzione può richiedere più cicli di *clock* per essere eseguita, mentre in architetture RISC le istruzioni sono a singolo ciclo;

2. CISC ha istruzioni più complesse e di lunghezza variabile, mentre RISK ha istruzioni semplici e standardizzate;

3. L'ISA CISC ha un grande numero di istruzioni mentre il RISC ha un insieme di istruzioni piccolo e di taglia fissa;

4. L'ISA CISC ha un insieme di modi di indirizzamento complesso, mentre l'ISA RISC ha un insieme di modi di indirizzamento limitato;

5. La fase di decodifica delle istruzioni è più complessa in CISC che in RISC;

6. In RISC le istruzioni possono avere formato da 16 bit a 64 bit, mentre in CISC solo a 32 bit (64 bit nel caso della macchina virtuale implementata).

Per i motivi elencati sopra, per questo lavoro ci si è concentrati su un sottoinsieme ristretto delle istruzioni dell'ISA CISC da tradurre verso l'ISA della macchina virtuale (che è appunto RISC), considerando solo istruzioni che usano operandi a 64 bit.

Questa scelta è dettata dal fatto che gli indirizzamenti ed i registri della macchina virtuale sono a 64 bit, quindi considerare anche istruzioni che usino registri a 32 o 16 bit avrebbe comportato un *effort* non trascurabile nella realizzazione dell'applicazione di transcodifica.

Non si tratta comunque di una limitazione semantica, dal momento che tutti i programmi a 16 o 32 bit possono essere riscritti come programmi a 64 bit.

In particolare ogni istruzione `x86_64` viene mappata nel modo seguente:

- se nell'ISA RISC della macchina virtuale è presente una istruzione equivalente a quella di partenza, la traduzione è 1-a-1;
- mentre, se l'istruzione considerata non ha un corrispettivo nell'ISA di destinazione, allora viene utilizzata una sequenza di istruzioni che siano semanticamente uguali a quella di partenza.

3.1.2 Organizzazione della memoria virtuale

La disposizione della memoria del programma virtualizzato è raffigurata in Figura 3.3.

Tale partizione non è fissata e può variare in base al fatto che, ad esempio, siano o meno utilizzati nel programma C virtualizzato variabili globali, dati *read-only* o argomenti da riga di comando.

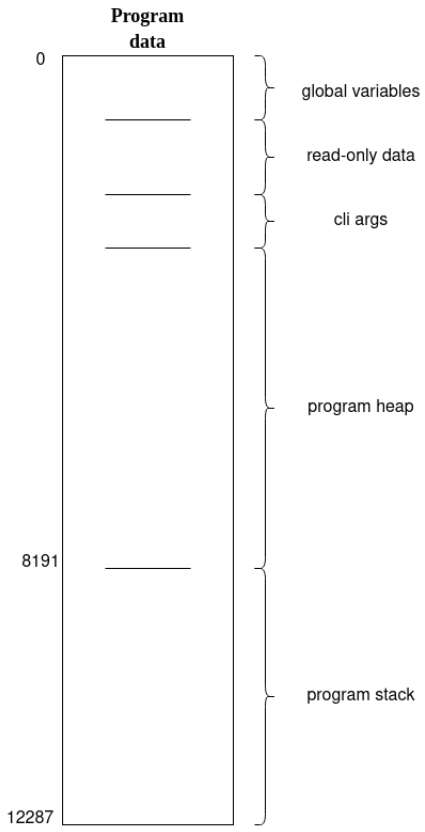

Figura 3.3: Suddivisione della memoria virtuale del programma C

Nel caso in cui non vi fossero, l'*heap* di programma inizierà prima, andando ad usare la memoria non utilizzata da tali dati.

È importante sottolineare che questo *layout* della memoria è totalmente arbitrario e può essere modificato secondo le necessità di utilizzo della macchina virtuale.

3.1.3 Offuscamento della libreria standard

Un ulteriore aspetto considerato per migliorare l'offuscamento offerto dalla soluzione è stato quello di re-implementare parte della libreria standard di C (`libc`): difatti, se uno degli obiettivi è quello di offuscare dettagli algoritmici legati al-

l'implementazione (come introdotto in Sezione 1.1), offuscando anche il codice delle funzioni della `libc` eventualmente invocate permette di rendere ancora più complesso per un attaccante capire che tipo di algoritmo sia stato effettivamente implementato.

Per alcune funzioni della libreria standard è stato fornito un *wrapper* che andasse a richiamare la funzione originale, questo principalmente per due motivi:

1. un programma che non effettua alcuna chiamata verso la libreria standard può destare maggiori sospetti da un punto di vista di chi fa il *reversing* del codice;

2. le funzioni per cui è stato realizzato tale *wrapper* sono poco legate ad aspetti algoritmici, difatti sono ad esempio funzioni di standard I/O come ad esempio; `printf` o funzioni per lettura/scrittura su file.

Questa scelta è stata fatta per aumentare lo sforzo necessario all'attaccante per ricostruire il programma originale.

Infatti, nel momento in cui viene analizzato il binario, si trova una chiamata che sembra essere ad una funzione della VM, rendendo quindi più difficile la distinzione da una funzione di libreria.

Peraltro, l'organizzazione dell'ISA scelta è tale per cui tutte le funzioni di libreria afferiscono ad una sola funzione della macchina virtuale che si comporta come *dispatcher*.

3.1.4 Convezioni per le chiamate di funzioni

La macchina virtuale specifica delle convenzioni per le chiamate di funzioni che differiscono da quelle tradizionali.

Vengono quindi riassunte di seguito tali *calling conventions*, valide sia per le chiamate a sottofunzione, che anche per le chiamate a funzioni della `libc` o a funzioni di sistema:

- Tutti i parametri delle funzioni vanno passati nei registri `r0`-`r15` in questo ordine;
- nel caso di una chiamata di sistema, il registro `r0` contiene il numero identificativo della funzione chiamata, quindi i parametri vengono passati a partire dal registro `r1` in ordine;
- per invocazioni di funzioni di libreria deve essere impostato il registro `libc`
- Il valore di ritorno delle funzioni va salvato nel registro `v0`;
- prima di invocare una funzione, il valore del registro `ip` va salvato nel registro `ret`, per poi essere recuperato al ritorno della funzione.

Tutti i dettagli relativi ai registri vengono approfonditi in Sezione 4.1.

3.1.5 Gestione della memoria dinamica del programma virtualizzato

Per quanto riguarda l'allocazione di memoria dinamica, questa avviene avvalendosi di un vettore pre-allocato nella VM in cui vengono salvati i dati dinamicamente o disallocati nel momento in cui non sono più necessari.

Tale meccanismo di de/allocazione viene gestito da un **core allocator**, che si occupa di mantenere lo stato su dati e metadati associati ad ogni blocco di memoria richiesto dall'applicazione.

Questo meccanismo permette di segregare nella maniera corretta l'applicazione virtualizzata (il binario x86 convertito) all'interno della *sandbox*, tenendo separati lo spazio di indirizzamento di quest'ultima da quello della VM stessa.

È interessante evidenziare che le varie implementazioni offerte per la libreria `malloc` si basano su meccanismi del sistema operativo sottostante, analogamente a quanto è stato fatto per l'implementazione di questa soluzione.

Pertanto, la soluzione proposta per l'allocazione e gestione della memoria dinamica è perfettamente compatibile con qualsiasi altra implementazione della libreria `malloc`.

3.1.6 Applicazione di transcodifica

Uno degli elementi centrali del lavoro di tesi è l'applicazione di transcodifica, che permette a partire da un file compilato per architetture x86 a 64 bit di ottenere il *bytecode* corrispondente da eseguire all'interno della macchina virtuale.

L'architettura ad alto livello per l'applicazione di transcodifica è riportata in Figura 3.4

Il file binario in input viene disassemblato e vengono effettuate una serie di operazioni dal transcodificatore, divisibili in:

- **Gestione dei dati**:
  1. Ricerca di eventuali dati globali o variabili *read-only*;
  2. Gestione dei parametri a riga di comando, se usati.
- **Gestione delle chiamate verso funzioni di libreria o di sistema**: si tiene traccia degli offset dove avvengono tali chiamate, per andare successivamente a tradurle in maniera appropriata;

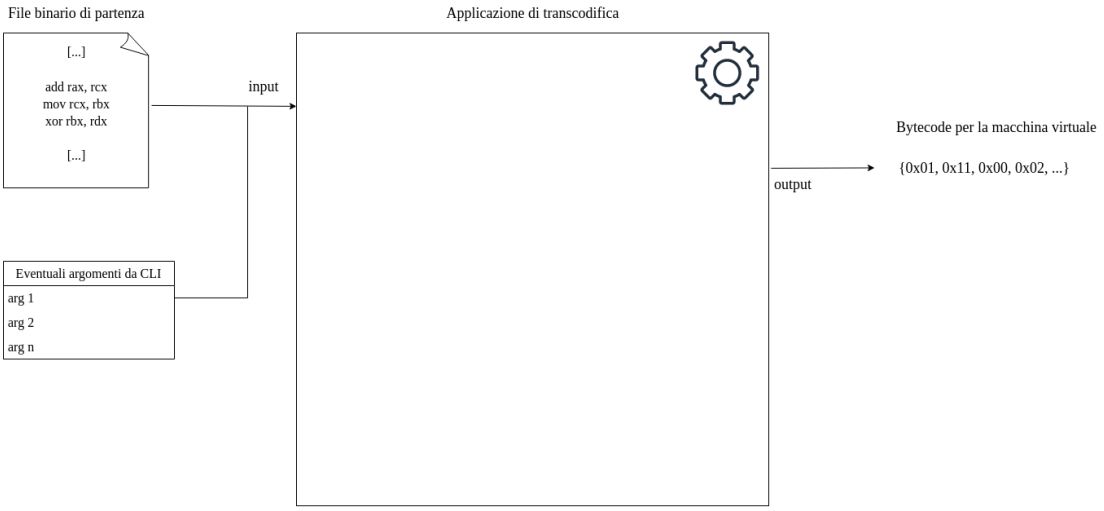

Figura 3.4: Architettura ad alto livello dell'applicazione di transcodifica

- **Gestione delle istruzioni**: ogni istruzione disassemblata viene tradotta in un'istruzione corrispondente, o in un gruppo di istruzioni semanticamente identiche nel caso in cui non vi sia un *mapping* 1 ad 1.

Per ognuno dei passi citati, viene prodotto il *bytecode* che verrà eseguito poi dalla macchina virtuale, che viene via via aggiunto all'output finale restituito dal transcodificatore.

Capitolo 4

Architettura ed implementazione

4.1 Banco dei registri

La definizione dei registri usati nella macchina virtuale è riassunta nel Listato 1.

Listato 1: Registri dell'ISA della macchina virtuale

```
1  typedef uint64_t reg_content_t;
2  struct registers {
3     reg_content_t r0;
4     reg_content_t r1;
5     reg_content_t r2;
6     reg_content_t r3;
7     reg_content_t r4;
8     reg_content_t r5;
9     reg_content_t r6;
10    reg_content_t r7;
11    reg_content_t r8;
12    reg_content_t r9;
13    reg_content_t r10;
14    reg_content_t r11;
15    reg_content_t r12;
16    reg_content_t r13;
```

```
17      reg_content_t r14;
18      reg_content_t r15;
19      reg_content_t ret;
20      reg_content_t v0;
21      reg_content_t tgt;
22      reg_content_t sp;
23      reg_content_t ip;
24      reg_content_t libc;
25  };
```

I registri utilizzati combinano le ISA di ARM e MIPS, nel dettaglio il modo in cui se ne fa uso:

- i registri da `r0` ad `r15` sono *general purpose*;
- `ret` viene usato per salvare l'indirizzo di ritorno a seguito di una chiamata a funzione, per poi usarlo per ripristinare il registro `ip`;
- `ip` è l'*instruction pointer*, mantiene l'indirizzo dell'istruzione correntemente analizzata e viene incrementato mano a mano che le istruzioni vengono processate (tenendo conto ovviamente anche di eventuali byte di *padding*);
- `v0` è usato per salvare il valore di ritorno delle chiamate a funzione e questo vale anche per le funzioni di libreria o di sistema;
- `tgt` viene utilizzato per caricare l'indirizzo di destinazione prima di effettuare qualunque operazione di salto o prima di ritornare da una chiamata a funzione;
- `sp` è lo *stack pointer*, tiene traccia della cima dello *stack* ed è usato per operazioni di `push` e `pop`;
- infine, il registro `libc` viene usato per scrivervi il numero della funzione di libreria da invocare.

4.2 Instruction Set Architecture

Le istruzioni dell'ISA della macchina virtuale sono definite nel Listato 2:

Listato 2: Istruzioni dell'ISA virtuale

```
1     enum insn_type {
2     ADD_R_I,
3     ADD_R_R,
4     AND_R_I,
5     AND_R_R,
6     JMP,
7     JE_R_I,
8     JE_R_R,
9     JLT_R_I,
10    JLT_R_R,
11    LOAD_R_M,
12    LOAD_R_I,
13    LTGT,
14    NOT_R,
15    NEG_R,
16    SHL_R_I,
17    SHR_R_I,
18    SAR_R_I,
19    STORE_M_R,
20    SYSCALL,
21    XOR_R_I,
22    XOR_R_R,
23    JAL,
24    RET,
25    BRK,
26    PUSH_R,
27    PUSH_RET,
28    POP_R,
29    LEA_R_M,
30    LIBC,
31    MORPH,
32    NOP
```

```
33 };
```

Le istruzioni implementate dall'ISA sono suddivise in base alla funzionalità:

- `load`: comprende le istruzioni di caricamento, incluse le `pop` e `push` da/verso lo *stack*;
- `store`: istruzione di salvataggio in memoria;
- `brk`: gestione della memoria dinamica mediante *program break*;
- `alu`: istruzioni di aritmetica intera e operazioni logiche (`and`, `or` e `xor`);
- `flow`: istruzioni sul flusso del programma, come `jmp` e `ret`;
- `lea`: istruzione di caricamento di indirizzo dalla memoria del processo eseguito in *sandbox* in un registro della VM;
- `syscall`: chiamata alle *system call* mediante l'uso di ***asm inline***.

  Questo è l'unico punto in cui vi è un'effettiva interazione della macchina virtuale con il sistema operativo sottostante, quindi è possibile offrire compatibilità con qualsivoglia Sistema Operativo permettendo l'esecuzione della macchina virtuale al di sopra di questo senza la necessità di altre dipendenze.

4.3 Programma virtualizzato

Il programma virtualizzato e la memoria corrispondente sono definiti nel Listato 3.

Listato 3: Strutture di memoria e programma virtualizzati

```
1
2  [...]
3
4  typedef uint8_t mem_cell;
5
6  #define MEM_LOCATIONS 8192
7  #define STACK_LOCATIONS 4096
8
9  extern mem_cell memory[MEM_LOCATIONS+STACK_LOCATIONS];
10
11 [...]
12 extern __attribute__((weak)) unsigned char
13                                  program[MEM_LOCATIONS];
```

Tale *array* `program` viene poi ogni volta ridichiarato nel file del *main* di programma, scrivendovi al suo interno il *bytecode* del programma virtualizzato.

4.4 Operatività della VM

Il punto d'ingresso della macchina virtuale è mostrato nel Listato 4

Listato 4: Operazioni principali eseguite dalla VM

```
1  [...]
2
3  int execute_main_program(void)
4  {
5     regs.sp = STACK_LOCATIONS;
6     struct instruction decoded;
7
8     compress_program();
9
10    while(regs.ip < MEM_LOCATIONS) {
11        decode(&decoded, program, &regs.ip);
```

```
12          execute(&decoded);
13      }
14      return 0;
15  }
16
17  // This is the entry point of the VM.
18  // It's declared weak because the library
19  // could be used independently of the
20  // default implementation of the main
21  // method.
22  // This is actually done in the tests.
23  __attribute__((weak)) int main(void)
24  {
25      return execute_main_program();
26  }
```

Il `main` è definito in modo che la libreria possa essere usata ridichiarando tale metodo, cosa che avviene ad esempio nei programmi di test.

La modalità operativa è composta da 3 operazioni principali:

1. `compress_program`: prima operazione effettuata, vengono dichiarate due strutture:

   - `encoded`, struttura di tipo `qbuf` che manterrà l'operazione finale da riscrivere nel vettore programma;
   - `decoded` di tipo `instruction`, mantiene l'istruzione letta inizialmente dal vettore programma.

   Viene poi eseguito un ciclo che itera tutti i byte del vettore `program` dove:

   - viene chiamata la funzione `decode` per mappare la sequenza di *bytes* letti in un'istruzione dell'ISA;

- successivamente, viene invocata la funzione di `encode` che scrive l'istruzione decodificata all'interno del *quickbuffer* allocato precedenza. Tale operazione va a generare gli eventuali byte di *padding* mediante l'uso del generatore random appositamente implementato;
- infine, tale buffer viene riscritto nel vettore `program`.

2. `decode`: decodifica un'istruzione, impostandone l'`opcode`, la `mode` e gli operandi. In questa fase vengono anche considerati eventuali byte di *padding*, che semplicemente non vengono considerati durante il *decoding*;

3. `execute`: esegue l'istruzione appena decodificata.

4.5 Implementazione della `libc`

Per poter utilizzare le funzioni della libreria standard C, è stato necessario effettuare un ***porting*** delle funzioni per cui garantire il supporto.

Il vettore di `struct` introdotto nel Listato 5 viene poi inizializzato, in modo che per le funzioni implementate venga impostato come valore del *function pointer* della `struct` tale funzione, come mostrato nel Listato 6.

Listato 5: Struttura dati per ogni funzione della libc

```
1 struct libc_table {
2     enum libc_fn code;
3     void (*fn)(enum libc_fn code);
4 };
5
6 extern struct libc_table libc_table[];
```

Listato 6: Estratto della tabella delle funzioni della libc supportate

```
1  struct libc_table libc_table[] = {
2     { LONGJMP, v_libc_jmp},
3     { SETJMP, v_libc_jmp},
4     { PRINTF, v_printf},
5     { FPRINTF, v_printf},
6     { DPRINTF, NULL},
7     { SPRINTF, NULL},
8     { SNPRINTF, v_printf},
9
10    [...]
11 };
```

Il *porting* ha richiesto la modifica di due aspetti fondamentali, per essere coerenti con le convenzioni introdotte in Sottosezione 3.1.4:

- il modo con cui vengono passati i parametri, i cui valori vengono scritti all'interno dei registri ***general purpose*** dell'ISA `r0--r15` (nell'ordine);
- il valore di ritorno della funzione, passato nel registro `v0`

Per invocare la funzione viene usata l'istruzione dell'ISA `LIBC` e l'apposito registro `libc` in cui viene scritto il numero della funzione da chiamare, in ciascun file contenente l'implementazione vi è l'uso del costrutto `switch/case` per determinare l'effettiva funzione da invocare.

Questo meccanismo contribuisce ulteriormente a rendere meno chiaro il *path* di esecuzione del programma virtualizzato, sempre per andare a complicare tentativi di analisi dinamica del codice.

Un esempio di implementazione è mostrato nel Listato 7

Listato 7: Gestione del codice della funzione `libc` invocata

```
1  void v_string(enum libc_fn code)
2  {
3     switch(code) {
4          case STRNCAT:
5              regs.v0 = (reg_content_t)strncat(
6                  (char *)virt_to_phys(get_register(1)),
7                  (const char *)virt_to_phys(
8                  get_register(0)),
9                  (size_t)get_register(2));
10             break;
11         case STRLEN:
12             regs.v0 = strlen(
13                     (const char*)virt_to_phys(
14                     get_register(0)));
15             break;
16         case STRNLEN:
17             regs.v0 = strnlen(
18                     (const char*)virt_to_phys(
19                     get_register(0)),
20                 (size_t)get_register(1));
21             break;
22    [...]
23    default:
24             abort();
25     }
26 }
27
28 [...]
```

Come detto nella Sottosezione 3.1.3, mentre alcune funzioni è stata riscritta un'implementazione completa, per altre è stato realizzato un *wrapper*, che si occupasse soltanto di recuperare i parametri passati mediante i registri della macchina virtuale per poi andare a chiamare la funzione effettiva della libreria standard.

Un esempio è mostrato nel Listato 8.

Listato 8: Wrapper per la funzione `printf`

```
1  void v_printf(enum libc_fn print_code)
2  {
3     switch(print_code) {
4
5         [...]
6
7         case PRINTF:
8             regs.v0 = wrap__vfprintf(
9                 stdout,
10                (const char *)get_register(0),
11                get_register(1), get_register(2),
12                get_register(3), get_register(4),
13                get_register(5), get_register(6),
14                get_register(7), get_register(8),
15                get_register(9), get_register(10),
16                get_register(11), get_register(12),
17                get_register(13), get_register(14),
18                get_register(15));
19            break;
20
21        [...]
22    }
23 }
24
25 [...]
26
27 int wrap__vfprintf(FILE *restrict s, const char *fmt,
28                    ...)
29 {
30    int ret;
31    va_list arg;
32
33    va_start(arg, fmt);
34    ret = vfprintf(s, fmt, arg);
35    va_end(arg);
36
```

```
37      return ret;
38  }
39
40  [...]
```

Per l'implementazione delle singole funzioni della `libc` è stato fatto riferimento a [8].

4.6 Allocatore di memoria dinamica

Per mantenere separati gli spazi di indirizzamento della *sandbox* e del programma in esecuzione al suo interno, è anche necessario che la memoria dinamica venga allocata all'interno della *sandbox* stessa.

Per questo motivo l'*heap* del programma virtuale è incluso nel vettore `memory` introdotto in Sezione 4.3

Per riservare lo spazio nel vettore `memory`, è stato implementato un allocatore di memoria, il quale è costruito sull'istruzione `brk`.

Tale istruzione mantiene il ***program break***, ovvero il primo byte libero di memoria da cui iniziare per allocare lo spazio eventualmente necessario al programma.

L'allocatore considera la memoria già allocata come una sequenza contigua di blocchi, utilizzando la `struct` del Listato 9:

Listato 9: Blocco di memoria mantenuto dall'allocatore

```
1  typedef struct memblock {
2      uint16_t size;
3      bool used;
4  } mem_block;
```

viene quindi mantenuta la taglia del blocco e l'indicazione sul fatto che tale blocco sia o meno utilizzato.

L'allocatore espone due funzioni, che vengono chiamate all'interno di `malloc` e `free` rispettivamente:

- `sb_alloc`: alloca la memoria richiesta dal programma nel seguente modo:
  - se vi è un blocco libero di taglia sufficiente, utilizza tale blocco eventualmente dividendolo in due blocchi se la taglia richiesta è inferiore alla taglia totale del blocco;
  - se non ci sono blocchi liberi, invoca la `brk` richiedendo memoria sufficiente ad ospitare sia i dati che i metadati per tale blocco;
  - altrimenti, non è più possibile allocare memoria per il programma.
- `sb_free`: libera un blocco di memoria, andando a cancellare i dati mediante la funzione `memset` imposta il campo `used` a `false` così che il blocco possa essere usato in allocazioni successive.

4.7 Applicazione di transcodifica

Per realizzare l'applicazione di transcodifica, è stato fatto uso della libreria **capstone** [3], che permette la scrittura di codice in diversi linguaggi per l'analisi di istruzioni che vengono convertite nell'ISA di riferimento, in questo caso x86 a 64bit.

4.8 Struttura del programma di transcodifica

Il transcodificatore è composto dalle seguenti classi:

- `X86VmTranspiler`: classe contenente la logica principale, in cui vi sono metodi per la conversione dei dati e delle istruzioni partendo dal file oggetto;
- `Instruction`: classe contenente l'implementazione dei pattern individuati e la loro conversione nell'ISA virtuale;
- `X86VmIsa`: contente diversi dizionari Python che mantengono *mapping* fra informazioni di x86 e informazioni virtuali;
- `VmMemory`: usata per gestire la conversione degli indirizzamenti presenti nel file oggetto verso quelli che saranno usati nella VM.

4.9 Individuazione dei limiti della funzione

Tutte le applicazioni virtualizzate sono composte di un'unica funzione principiale, ovvero il *main* di programma e la prima operazione effettuata è il *parsing* del file oggetto originale.

Tale *parsing* viene effettuato mediante l'uso del *tool* da riga di comando `objdump`, per individuare:

- indirizzi di inizio e di fine della funzione *main*;
- l'uso di funzioni di libreria o di *system call*;
- uso di eventuali variabili globali e/o dati *read-only*.

In particolare, per questi ultimi, viene ancora usato `objdump`, passando i seguenti argomenti:

```
objdump -s .section -j objfile
```

dove `.section` è la sezione del file oggetto dove cercare i dati, rispettivamente `.data` per le variabili globali e `.rodata` per i dati *read-only*.

4.10 Disassemblaggio del file

Una volta individuate tutte le informazioni necessarie, il disassemblaggio del file avviene mediante l'uso di **capstone**, come mostrato nel Listato 10

Listato 10: Utilizzo della libreria `capstone` nel transcodificatore

```
1  md = Cs(CS_ARCH_X86, CS_MODE_64)
2
3  md.syntax = CS_OPT_SYNTAX_ATT
4
5  [...]
6
7  instructions = md.disasm(binary, func_start)
8
9  [...]
10
11 # Main loop, for each instruction:
12 #   1. extract key features
13 #   2. group blocks of instructions
14 #      (e.g deref of cli params)
15 #   3. manage syscall VM convention
16 for inst in instructions:
17
18     [...]
19
20     newInsn = Instruction()
21
22     [...]
23
24     else:
25         newInsn.set_insn(inst.address, inst.mnemonic,
26             self._divide_operand(inst.op_str))
```

```
27
28        [...]
```

Scorrendo istruzione per istruzione, vengono individuati pattern quali:

- uso di variabili globali;
- uso di variabili *read-only*;
- chiamate a *system call* o funzioni di libreria.

e vengono conseguentemente virtualizzate facendo riferimento all'ISA di destinazione.

4.11 Caricamento dei dati in memoria

Ciascuna zona della memoria viene popolata in base a quanto parsato dal file oggetto.

4.11.1 Variabili globali e read-only

Le variabili globali ed i dati *read-only* vengono letti, rispettivamente dalle sezioni `.data` e `.rodata`, del file oggetto mediante l'uso del *tool* da riga di comando `objdump`:

```
objdump -s .section -j objfile
```

L'indirizzo all'interno delle sezioni viene estrapolato dal file oggetto, un esempio è mostrato nei Listati 11, 12, 13.

Listato 11: Esempio di utilizzo di variabili globali e *read-only*

```
1 122f:   48 8d 05 d6 0d 00 00    lea 0xdd6(%rip),%rax
  # 200c <_IO_stdin_used+0xc>
2 1236:   48 89 c7                mov %rax,%rdi
3 1239:   b8 00 00 00 00          mov $0x0,%eax
4 123e:   e8 2d fe ff ff          call 1070
5        <printf@plt>
6
7     [...]
8
9 1266:   48 8b 05 f3 2d 00 00    mov 0x2df3(%rip),%rax
10 # 4060 <loop_2>
11 126d:   48 31 45 d8             xor %rax, 0x28(%rbp)
12 1271:   48 83 45 e8 01          addq $0x1, 0x18(%rbp)
13 1276:   48 8b 05 e3 2d 00 00    mov 0x2de3(%rip),%rax
14 # 4060 <loop_2>
```

Listato 12: Esempio di disassemblamggio della sezione `.data`

```
1         formato del file elf64 x86 64
2
3 Contenuto della sezione .data:
4 4040 00000000 00000000 48400000 00000000
  ........H@......
5 4050 32000000 00000000 0b000000 00000000
  2...............
6 4060 05000000 00000000                    ........
```

Listato 13: Esempio di disassemblaggio della sezione `.rodata`

```
1         formato del file elf64 x86 64
2
3 Contenuto della sezione .rodata:
4 2000 01000200 6d6f7270 68766d00 436f6d70
```

```
 ....morphvm.Comp
5 2010 75746174 696f6e20 666f7220 76616c20
 utation for val
6 2020 256c643a 2000256c 6420002f 746d702f
 %ld: .%ld ./tmp/
7 2030 6c6f672e 766d0055 73657220 00207265
 log.vm.User . re
8 2040 71756573 74656420 636f6d70 75746174
 quested computat
9 2050 696f6e0a 00                              ion..
```

4.11.2 Argomenti a riga di comando

Il passaggio degli argomenti a riga di comando per il programma C avviene usando il medesimo meccanismo, ma lanciando il programma di transcodifica.

Difatti, tale programma fornisce una classe `Main` che procede poi a passare gli argomenti ricevuti alla classe `X86VmTranspiler`.

È quindi sufficiente lanciare il programma da terminale con il comando:

```
python3 Main.py <percorso_eseguibile> <argomento_1> <argomento_2>
...
```

Quindi, per gestire gli argomenti a riga di comando, sono state apportate le seguenti semplificazioni:

- Normalmente, `argv` è un *array* di puntatori a `char *` (`char **` o `char *[]`);
- in questo caso, la memoria della VM memory è un array logico contiguo, quindi `argv` viene trattato come una sequenza di caratteri.

Detto ciò, per gestire gli argomenti a riga di comando, la prima operazione da fare è caricarli in memoria, come avviene per i dati globali e quelli *read-only*.

Dopodiché, quando il programma accedere agli argomenti da riga di comando, tipicamente questo avviene partendo dal puntatore salvato sullo stack (ovvero il contenuto del registro `%rsi` andandosi a spiazzare per accedere al parametro successivo.

Un esempio viene mostrato nello Listato 14.

Listato 14: Caricamento degli argomenti da riga di comando

```
11b4:   48 89 75 b0   mov %rsi, 0x50(%rbp)
11cf:   48 8b 45 b0   mov  0x50(%rbp),%rax
11d3:   48 83 c0 08   add $0x8,%rax
11d7:   48 8b 00      mov (%rax),%rax
```

Dato che i parametri a riga di comando vengono salvati in memoria sotto forma di sequenza di caratteri, nel programma di transcodifica viene individuato il pattern di accesso a tali parametri e convertito semplicemente in:

- `mov` dell'indirizzo del parametro, in caso di chiamate a `libc`;
- `lea` dell'indirizzo del parametro nel registro in tutti gli altri casi.

Questa scelta permette di risparmiare *bytes* del vettore `program`, in quanto l'operazione di `lea` ha un costo di 5 *bytes* (senza padding) contro i 10 bytes se vengono virtualizzate sia la `mov` che la `add`; questo si riesce ad ottenere spostando la complessità dell'operazione sul transcodificatore.

4.12 ISA di destinazione

Nelle sezioni successive, vengono mostrate alcune traduzioni delle istruzioni dall'ISA x86 verso quello virtuale.

Queste traduzioni mostrano quanto detto in Sottosottosezione 3.1.1, ovvero la necessità di dover mappare una singola istruzione dell'ISA x86_64, un ISA CISC formato da un grande numero di istruzioni di dimensione arbitraria, verso un gruppo di istruzioni della macchina virtuale che siano semanticamente uguali.

4.12.1 Schema per istruzioni `mov`

Alcuni dei possibili schemi per operazioni `mov` considerati dalla logica di transcodifica:

- **`mov` con il registro `(%rbp)` come uno degli operandi**:
  - se l'altro operando è un registro, ad esempio `mov %rx -0xi(%rbp)`:
    - `STORE_M_R`: viene salvato il valore del registro `%rx` sullo stack, a spiazzamento dato da `-0xi`.
  - se il secondo operando è un immediato, e.s. `mov 0xij -0xi(%rbp)`:
    - `LOAD_R_I`: salva il valore dell'immediato in un registro ***general purpose***, fra `r0 - r15`, non utilizzato (la scelta di tale registro viene effettuata dalla logica di transcodifica);
    - `STORE_M_R`: salva il contenuto di tale registro sullo *stack*, a spiazzamento dato da `-0xi`.
  - se `(%rbp)` è l'operando sorgente, e.s. `mov -0xi(%rbp) %rx`:
    - `LOAD_R_M`: carica il valore dallo *stack* in un registro ***general purpose*** non utilizzato, caricando dalla memoria a spiazzamento `-0xi`.
- **`mov %rx %ry`**, operazione fra due registri:

  - `XOR_R_R`: azzera il registro di destinazione mediante un'operazione di `XOR` con se stesso;
  - `ADD_R_R`: "carica" il valore `%rx` dentro `%ry`.

4.12.2 Schema per istruzioni `add`

Alcuni dei possibili schemi individuati per le operazioni di `add`:

- **Viene usato lo stack come uno degli operandi, e.s. `add %rx (o imm), -0xi(%rbp)`**:
  - `LOAD_R_M`: carica il valore dallo *stack* in un registro non usato;
  - `ADD_R_R / ADD_R_I`: somma il registro con l'altro operando, che può essere un registro o un immediato;
  - `STORE_M_R`: salva il valore aggiornato dal registro sullo *stack*.

  Nel caso in cui lo *stack* è l'operando sorgente, non viene effettuata l'ultima operazione di `STORE_M_R`

4.12.3 Schema per istruzioni di `jmp` e `jmp` condizionali

L'ISA virtuale fa uso del registro `tgt` per caricare gli indirizzi di destinazione per un'operazione di salto.

Per questo, per ogni operazione di questo tipo, la prima cosa da fare è caricare la destinazione in tale registro mediante l'uso dell'apposita istruzione `LTGT`. Dopodiché, alcuni degli schemi individuati sono:

- **salto condizionale, e.s. `jmp + cmp`**. L'operazione di comparazione viene saltata, ovvero vengono salvati solo gli operandi per essere processati in

seguito, così da virtualizzare sia la comparazione che il salto insieme come segue:

- se la `cmp` è fra un immediato e lo *stack*
  - `LOAD_R_I`: carica il valore immediato in un registro non utilizzato;
  - `LOAD_R_M`: carica il valore dallo stack in un altro registro non utilizzato.
- se la `cmp` è fra un immediato ed un registro:
  - `LOAD_R_I`: carica l'immediato in un registro non utilizzato
- se la `cmp` è fra due registri:
  - `XOR_R_R`: azzera un registro non utilizzato;
  - `ADD_R_R`: somma il primo operando con tale registro.

Ora è possibile effettuare l'operazione di salto, che eseguirà anche una comparazione:

- se il salto è un'operazione fra `jl, jg, jb, jne`:
  - `XOR_R_R`: finché la comparazione risulterà maggiore o minore, il risultato sarà $\neq 0$;
  - `JLT_R_I`: salta finché il risultato della XOR è > 0.
- `jbe`:
  - Come sopra, ma aggiunge 1 per cambiare la condizione da *less than or equal* a *less than*
- `je`:
  - `XOR_R_R` come in precedenza
  - `JE_R_I`: salta se il risultato della `XOR` è $== 0$

4.12.4 Istruzione di `ret`

Per terminare il programma con la `ret`, è stato scelto di inserire un'ultima istruzione all'interno del *bytecode* che opera come segue:

- dato che il programma è una sequenza di *bytes* di lunghezza predefinita, ad esempio 8192, viene utilizzata un'operazione di `jmp`;
- prima di saltare, viene impostato il registro `tgt` ad 8191, ovvero l'ultimo byte del programma.

4.12.5 Syscalls

Le *syscall* in Linux per architetture `x86_64` seguono alcuni semplici regole nel passaggio dei parametri:

- il numero della *syscall* così come il valore di ritorno vengono scritti nel registro `%rax`;
- gli eventuali altri parametri sono passati mediante i registri `%rdi, %rsi, %rdx, %r10, %r8, %r9`.

L'ISA della macchina virtuale, come detto in precedenza, unisce le convenzioni di architetture ARM e MIPS in termini dei nomi dei registri e di come viene gestito il passaggio dei parametri per le *syscall*:

- il numero della *syscall* viene passato mediante `r0`;
- il valore di ritorno viene scritto in `v0`;
- gli eventuali altri parametri sono passati (in ordine) mediante `r1, r2, r3, r10, r8, r9`.

Quindi, nella virtualizzazione di operazioni di `call` che riguardano *syscall*, la logica di transcodifica deve:

- cambiare i registri di destinazione per il passaggio dei parametri, passando dallo standard `x86_64` a quello specifico della VM;
- cambiare l'istruzione sottostante alla `call` in modo che il registro da cui si recupera il valore di ritorno sia `v0` e non `%rax`.

Un esempio è nel Listato 15, che viene tradotto dal transcodificatore come indicato nel Listato 16.

Listato 15: Esempio di system call in assembly

```
12c6:   ba 80 01 00 00          mov $0x180,%edx
12cb:   be 41 04 00 00          mov $0x441,%esi
12d0:   48 8d 05 54 0d 00 00    lea 0xd54(%rip),%rax
# 202b <_IO_stdin_used+0x2b>
12d7:   48 89 c7                mov %rax,%rdi
12da:   b8 00 00 00 00          mov $0x0,%eax
12df:   e8 bc fd ff ff          call 10a0
       <open@plt>
12e4:   89 45 cc                mov %eax, 0x34(%rbp)
```

Listato 16: Risultato di transcodifica della system call `open`

```
4806   movl        ['$0x180'  ,    '%rdx']
4811   movl        ['$0x441'  ,    '%rcx']
4816   leaq        ['0x37'    ,    '%rax']
4823   movq        ['%rax'    ,    '%rbx']
4831   syscallq    ['$2'      ,    '%rax']
```

Capitolo 5

Valutazione prestazionale

Le sezioni sottostanti mostrano i risultati ottenuti nel valutare le *perfomance* della libreria di virtualizzazione.

Il primo aspetto di interesse è capire quale sia l'*overhead* prestazionale introdotto dal *layer* di virtualizzazione. In secondo luogo, questo permette di capire quanto si paga vista la necessità di dover tradurre istruzioni dall'ISA CISC verso l'ISA RISC della macchina virtuale.

5.1 Valutazione di *performance* ed *overhead*

Sono state valutate le *perfomance* e l'*overhead* introdotte dalla VM, andando a considerare programmi intensivi da un punto di vista dell'uso della CPU.

Tali programmi sono interessanti per valutare quanto detto riguardo la traduzione da un ISA CISC verso uno RISC, in quanto la maggior parte delle istruzioni dei programmi presi in analisi nel *test set* richiedono una traduzione in più istruzioni del set della macchina virtuale, questo quindi comporta un maggior numero di cicli di *clock* necessari per eseguire le istruzioni del binario virtualizzato.

Inoltre, è inevitabile che la virtualizzazione delle istruzioni comporti dei tempi di esecuzione maggiori, in quanto questa sarà certamente più lenta dell'esecuzione nativa del programma.

Infine, la test suite scelta è composta da programmi che, una volta transcodificati, vanno ad utilizzare la quasi totalità dell'ISA implementato nella VM.

Tutti i dati sono stati raccolti mediante l'uso del *tool perf* [9].

La piattaforma hardware su cui sono stati eseguiti i test presenta le seguenti caratteristiche:

- CPU: Intel core i5-1035G1 @ 1.00GHz;
- RAM: 12GB DDR4;
- OS: Manjaro kernel 5.15.78.

Tutti i risultati numerici mostrati sono stati mediati su 10 esecuzioni di ciascun programma di test.

5.1.1 Test 1: calcolo del numero di Fibonacci

Il primo esempio considerato è un programma che **calcola l'*i-esimo* numero di Fibonacci**.

I risultati numerici vengono riportati in Tabella 5.1 e Tabella 5.2.

Vengono poi mostrati i tempi di esecuzione a confronto in Figura 5.1 e l'*overhead* sul tempo di esecuzione dovuto all'uso della VM in Tabella 5.3.

L'*overhead* sul tempo di esecuzione si mantiene in media su un fattore di 2.46, questo mostra che nonostante la presenza del virtualizzatore e l'utilizzo di un numero di cicli di *clock* molto maggiore (un'ordine di grandezza in più), le performance subiscono un calo controllato.

5.1.2 Test 2: calcolo del fattoriale

Il secondo esempio di test preso in analisi è il calcolo del **fattoriale**. I risultati delle esecuzioni sono riportati in Tabella 5.4 ed in Tabella 5.5.

Vengono poi rappresentati i tempi di esecuzione in Figura 5.2 e l'*overhead* dovuto alla presenza del *layer* di virtualizzazione in Tabella 5.6.

Anche in questo caso, il fattore di l'*overhead* si attesta in media attorno a 2.43, rispetto al caso senza l'utilizzo della VM.

5.1.3 Test 3: calcolo della potenza n-esima

Il successivo test è stato effettuato considerando un programma che calcola il valore n^n. I risultati numerici per le esecuzioni vengono riportati in Tabella 5.7 ed in Tabella 5.8.

Il confronto dei tempi di esecuzione viene raffigurato in Figura 5.3 ed i dati riportati in Tabella 5.9.

Come per i casi precedenti, l'*overhead* dovuto al *layer* di virtualizzazione è in media pari ad un fattore di 2.32.

Anche in questo caso, il risultato conferma quanto atteso, dato che il programma non presenta un set di istruzioni da eseguire molto diverse da quello dei due precedenti.

5.1.4 Test 4: *shift* a destra di interi

Il successivo test svolto prevede la divisione per 2 (mediante operazione di *shift* a destra) di valori interi passati in input.

I risultati numerici per le esecuzioni vengono riportati in Tabella 5.10 ed in Tabella 5.11.

Il confronto dei tempi di esecuzione viene raffigurato in Figura 5.4 ed i dati riportati in Tabella 5.12.

Similmente ai casi precedenti, l'*overhead* dovuto al *layer* di virtualizzazione è in media pari ad un fattore di 2.34.

5.1.5 Test 5: *shift* a destra e somma di interi

Basandosi sul test 4, è stato realizzato un ulteriore test che combinasse lo *shift* a destra e la soma di interi passati in input.
I risultati numerici per le esecuzioni vengono riportati in Tabella 5.13 ed in Tabella 5.14.

Il confronto dei tempi di esecuzione viene raffigurato in Figura 5.5 ed i dati riportati in Tabella 5.15.

Similmente ai casi precedenti, l'*overhead* dovuto al *layer* di virtualizzazione è in media pari ad un fattore di 2.59. L'aumento dell'*overhead* nel tempo di esecuzione deriva dall'aggiunta dell'operazione di somma ulteriore allo *shift*, che comporta un maggior numero di istruzioni da eseguire.

5.1.6 Test 6: calcolo di una sequenza numerica

Il quarto test considerato per la valutazione prestazionale della soluzione proposta prevede l'uso di un programma che esegue una computazione che produce come output una sequenza numerica. Tale sequenza viene calcolata sulla base di un valore fornito in input dall'utente, come mostrato nel Listato 17

Listato 17: Computazione della sequenza del test 3

```
1
2 [...]
3
4 for(uint64_t i = 1; i < loop_1; i++) {
5     computation += user_val + i;
6
7     for(uint64_t j = 0; j < loop_2; j++) {
8         computation ^= loop_2;
9     }
10    printf("%ld", computation);
11 }
12
13 [...]
```

Pertanto, le performance sono state valutate al variare dei due limiti `loop_1` e `loop_2`.

Vengono riportati i risultati numerici in Tabella 5.16 e Tabella 5.17.

Infine, vengono rappresentati in Figura 5.6 i tempi di esecuzione a confronto ed in Tabella 5.18 i risultati numerici per l'*overhead* sul tempo di esecuzione.

Per questo esempio è molto più visibile la presenza del *layer* di virtualizzazione, anche all'aumentare del numero di iterazioni svolte per ciascun ciclo. È infatti evidente dai dati come, al crescere del numero di iterazioni effettuate per ogni ciclo, crescano di conseguenza il numero di cicli di *clock* ed il tempo di esecuzione.

Numero medio di cicli di *clock*	**Numero medio di istruzioni per ciclo di *clock***	**Numero di Fibonacci**
217926	0.590	20
223217	0.579	40
223175	0.579	60
222132	0.583	80

Tabella 5.1: Calcolo del numero di Fibonacci mediato su 10 esecuzioni senza virtualizzazione

Numero medio di cicli di *clock*	**Numero medio di istruzioni per ciclo di *clock***	**Numero di Fibonacci**
2318553	3.238	20
2400393	3.200	40
2444250	3.219	60
2515629	3.195	80

Tabella 5.2: Calcolo del numero di Fibonacci mediato su 10 esecuzioni con virtualizzazione

Fattore di *overhead*	**Numero di Fibonacci**
2.51	20
2.38	40
2.44	60
2.52	80

Tabella 5.3: *Overhead* sul calcolo del numero di Fibonacci mediato su 10 esecuzioni con virtualizzazione

Numero medio di cicli di *clock*	**Numero medio di istruzioni per ciclo di *clock***	$n!$
237320	0.56	5
245113	0.545	10
240294	0.557	15
238979	0.562	20

Tabella 5.4: Calcolo del fattoriale, mediato su 10 esecuzioni senza virtualizzazione

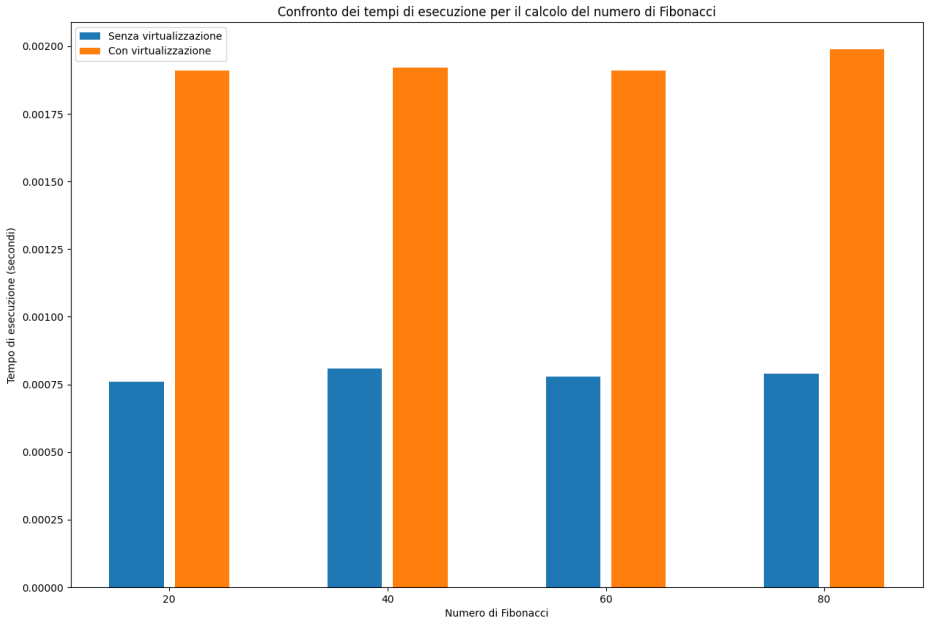

Figura 5.1: Tempo di esecuzione per il calcolo del numero di Fibonacci, con e senza virtualizzazione

Numero medio di cicli di *clock*	**Numero medio di istruzioni per ciclo di *clock***	$n!$
2385930	3.128	5
2426683	3.223	10
2666423	3.140	15
2828383	3.222	20

Tabella 5.5: Calcolo del fattoriale, mediato su 10 esecuzioni con virtualizzazione

Fattore di *overhead*	$n!$
2.39	5
2.24	10
2.47	15
2.63	20

Tabella 5.6: *Overhead* sul calcolo del fattoriale mediato su 10 esecuzioni

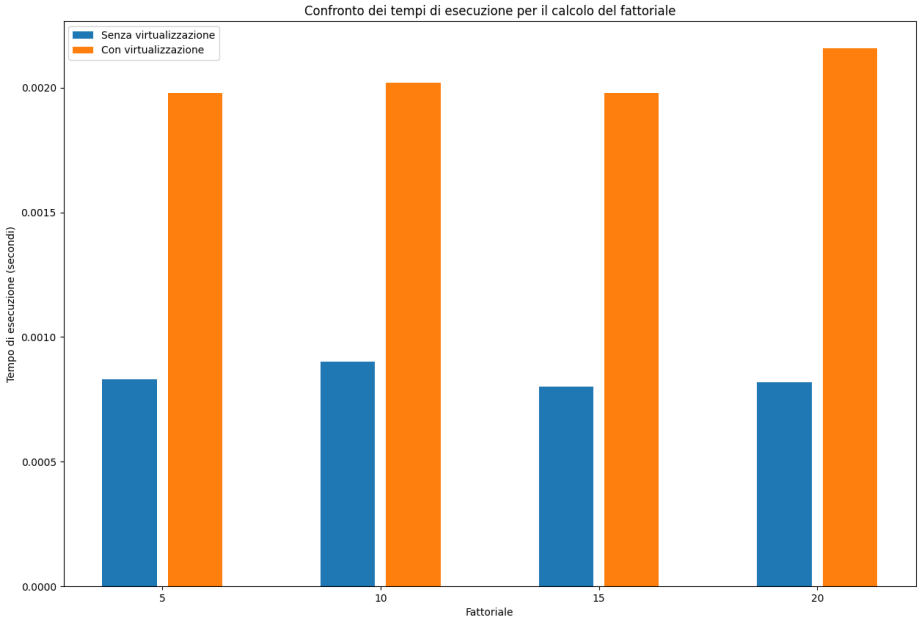

Figura 5.2: Tempo di esecuzione per il calcolo del fattoriale, con e senza virtualizzazione

Numero medio di cicli di *clock*	**Numero medio di istruzioni per ciclo di *clock***	n^n
248122	0.536	6^6
256730	0.520	9^9
251907	0.530	12^{12}
251963	0.534	15^{15}

Tabella 5.7: Calcolo di n^n, mediato su 10 esecuzioni senza virtualizzazione

Numero medio di cicli di *clock*	**Numero medio di istruzioni per ciclo di *clock***	n^n
2342773	3.194	6^6
2424845	3.192	9^9
2593107	3.124	12^{12}
2690905	3.186	15^{15}

Tabella 5.8: Calcolo di n^n, mediato su 10 esecuzioni con virtualizzazione

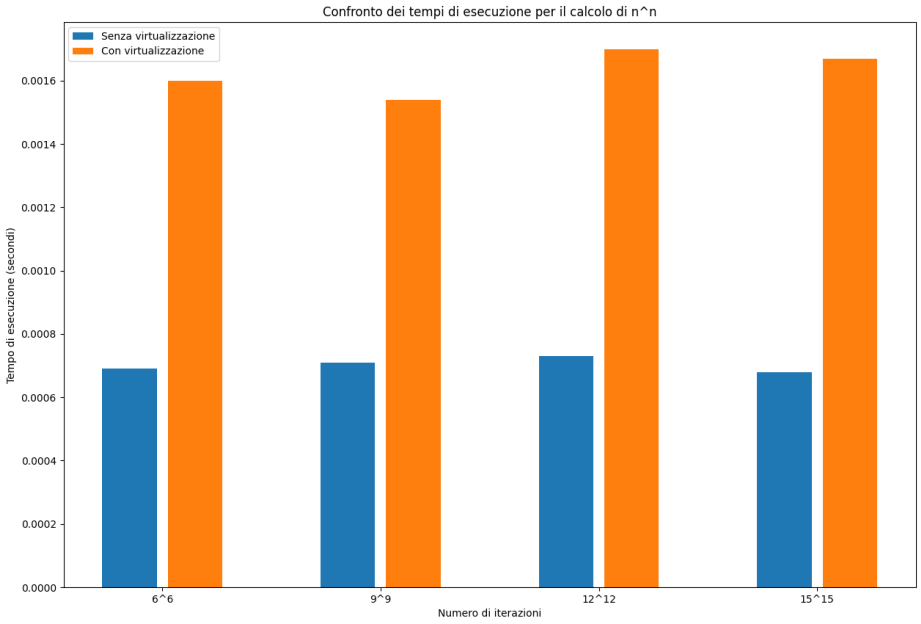

Figura 5.3: Tempo di esecuzione per il calcolo di n^n, con e senza virtualizzazione

Fattore di *overhead*	n^n
2.33	6^6
2.16	9^9
2.34	12^{12}
2.47	15^{15}

Tabella 5.9: *Overhead* sul calcolo di n^n mediato su 10 esecuzioni

Numero medio di cicli di *clock*	**Numero medio di istruzioni per ciclo di *clock***	**Valore**
257723	0.583	987654321
267897	0.569	123456789
273273	0.635	9876543219876543 21
286286	0.621	1234567891234567 89

Tabella 5.10: Divisione per 2 mediante *shift*, mediato su 10 esecuzioni senza virtualizzazione

Numero medio di cicli di *clock*	**Numero medio di istruzioni per ciclo di *clock***	**Valore**
2351129	3.240	987654321
2386544	3.181	123456789
2442960	3.236	9876543219876543 21
2455277	3.208	1234567891234567 89

Tabella 5.11: Divisione per 2 mediante *shift*, mediato su 10 esecuzioni con virtualizzazione

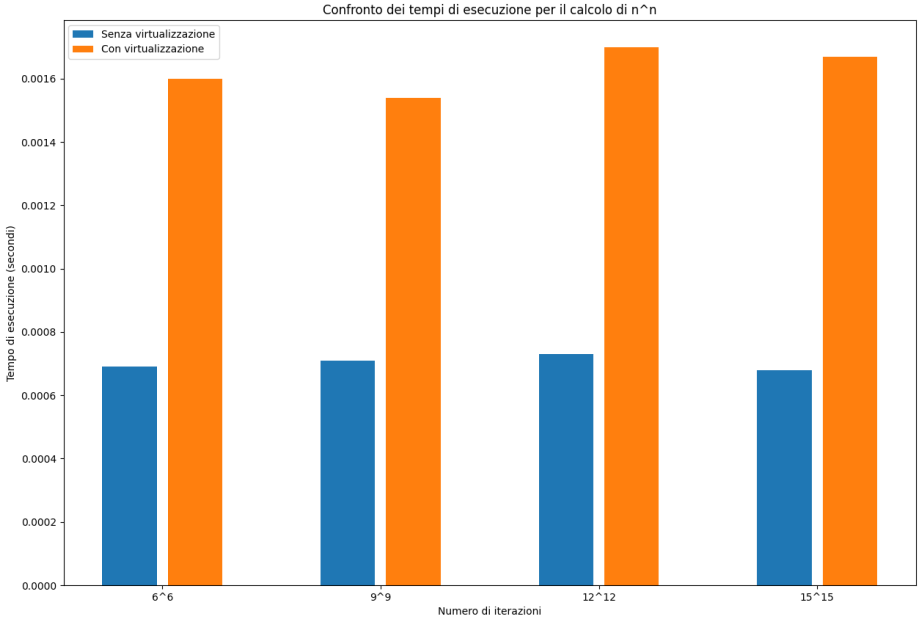

Figura 5.4: Tempo di esecuzione per il calcolo dello *shift* di numeri interi, con e senza virtualizzazione

Fattore di *overhead*	**Valore**
2.25	987654321
2.44	123456789
2.35	9876543219876543 21
2.32	1234567891234567 89

Tabella 5.12: *Overhead* per lo *shift* di interi, mediato su 10 esecuzioni

Numero medio di cicli di *clock*	Numero medio di istruzioni per ciclo di *clock*	Valore
251329	0.628	15446744073709551615
247347	0.638	16446744073709551615
253102	0.623	17446744073709551615
248014	0.636	18446744073709551615

Tabella 5.13: *shift* e somma di interi, mediato su 10 esecuzioni senza virtualizzazione

Numero medio di cicli di *clock*	Numero medio di istruzioni per ciclo di *clock*	Valore
2493958	3.034	15446744073709551615
2444992	3.095	16446744073709551615
2553031	2.975	17446744073709551615
2387246	3.153	18446744073709551615

Tabella 5.14: *shift* e somma di interi, mediato su 10 esecuzioni con virtualizzazione

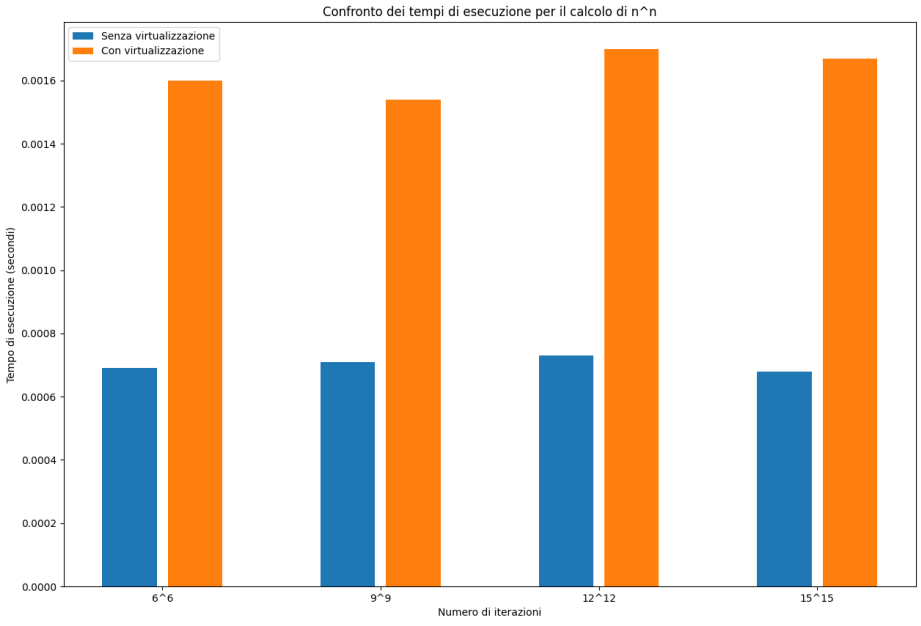

Figura 5.5: Tempo di esecuzione per lo *shift* e somma di numeri interi, con e senza virtualizzazione

Fattore di *overhead*	**Valore**
2.66	15446744073709551615
2.65	16446744073709551615
2.62	17446744073709551615
2.43	18446744073709551615

Tabella 5.15: *Overhead* per lo *shift* e somma di interi, mediato su 10 esecuzioni

Numero medio di cicli di *clock*	**Numero medio di istruzioni per ciclo di *clock***	`loop_1`	`loop_2`
257307	0.583	20	20
262402	0.639	40	40
280288	0.683	60	60
308316	0.713	80	80

Tabella 5.16: Computazione di una sequenza numerica, mediata su 10 esecuzioni senza virtualizzazione

Numero medio di cicli di *clock*	**Numero medio di istruzioni per ciclo di *clock***	`loop_1`	`loop_2`
3034205	3.286	20	20
5396765	3.437	40	40
9389534	3.484	60	60
14831897	3.535	80	80

Tabella 5.17: Computazione di una sequenza numerica, mediata su 10 esecuzioni con virtualizzazione

Fattore di *overhead*	`loop_1`	`loop_2`
2.64	20	20
3.77	40	40
5.27	60	60
7.83	80	80

Tabella 5.18: *Overhead* sul calcolo di una sequenza numerica mediato su 10 esecuzioni

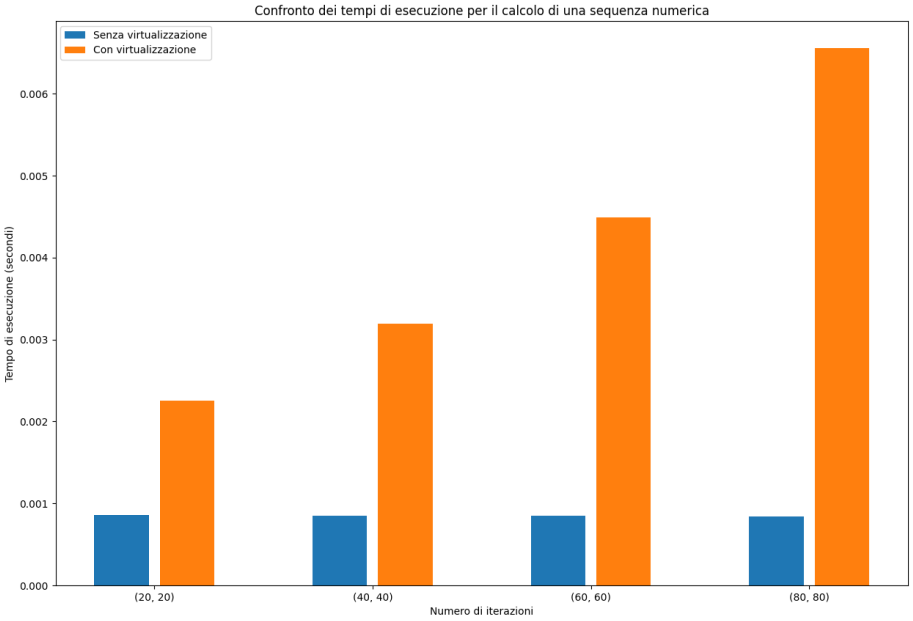

Figura 5.6: Tempo di esecuzione per il calcolo di una sequenza numerica, con e senza virtualizzazione

Capitolo 6

Conclusioni

In questo ultimo capitolo, vengono riassunti gli obiettivi raggiunti dal lavoro di tesi e vengono indicati quali possono essere alcuni degli sviluppi futuri.

La soluzione sviluppata è una macchina virtuale che permette di eseguire applicazioni che fanno uso di librerie.

Tale soluzione protegge la IP legata al codice sorgente dell'applicazione mediante diversi meccanismi di offuscamento, fra cui:

1. l'uso di byte di *padding* generati casualmente nelle istruzioni;
2. l'implementazione delle funzioni della libreria standard;
3. l'aggiunta di complessità all'*execution path* dell'applicazione quando questa va a chiamare le funzioni della libreria standard.

La libreria della macchina virtuale sviluppata è indipendente dal sistema operativo sottostante, inoltre rende possibile di virtualizzare e proteggere programmi di cui non è necessariamente presente il codice sorgente.

La soluzione è quindi applicabile a una serie disparata di contesti, in cui è importante preservare la Proprietà Intellettuale degli applicativi, in quanto non è

necessario neanche a livello di libreria di virtualizzazione stessa andare a leggere il contenuto del codice sorgente.

Per quanto riguarda i possibili sviluppi futuri:

- una possibile linea di sviluppo da seguire è l'introduzione di un **meccanismo di polimorfismo** all'interno della macchina virtuale, che vada ad esempio a mutare il *mapping* fra gli *opcode* delle istruzioni dell'ISA di partenza e quegli dell'ISA virtuale di destinazione, così da rendere ancora più complessa l'analisi dinamica del codice mediante *debugger*;

- sviluppare un *backend* mediante l'uso di `llvm` [13], in modo da supportare la virtualizzazione di programmi a partire dal codice sorgente;

- ampliare la logica di transcodifica, per arricchire il set di istruzioni `x86_64` supportate (e quindi conseguentemente anche la logica della VM), ed anche per supportare binari compilati per altri ISA, come ad esempio ARM;

- estendere il supporto ad applicazioni *multi-thread*, introducendo nella libreria stessa la possibilità di creare *thread* che vadano ad eseguire la porzione di codice del *thread* del programma virtualizzato, per poi riconciliare il flusso di esecuzione con il *main thread* della libreria.

Bibliografia

[1] Nezer Jacob Zaidenberg Amir Averbuch Michael Kiperberg. «*Truly-Protect*: An Efficient VM-Based Software Protection». In: *IEEE SYSTEMS JOURNAL* 7.3 (2018), pp. 1–23.

[2] Pietro Borrello, Emilio Coppa e Daniele Cono D'Elia. «Hiding in the Particles: When Return-Oriented Programming Meets Program Obfuscation». In: *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Virtual Event: IEEE, 2021, pp. 555–568. DOI: 10.1109/DSN48987.2021.00064.

[3] *Capstone - The Ultimate Disassembler*. URL: https://www.capstone-engine.org/lang_python.html.

[4] *Denuvo - The global #1 Games Protection and Anti-Piracy technology helping game publishers and developers to secure PC, console and mobile games.* URL: https://irdeto.com/denuvo/.

[5] Hui Fang1 et al. «Multi-stage Binary Code Obfuscation Using Improved Virtual Machine». In: (), pp. 1–14.

[6] *Ghdira - A software reverse engineering (SRE) suite of tools developed by NSA's Research Directorate in support of the Cybersecurity mission.* URL: https://ghidra-sre.org/.

[7] Kaiyuan Kuang et al. «Enhance virtual-machine-based code obfuscation security through dynamic bytecode scheduling». In: (2018), pp. 1–19.

[8] *musl - an implementation of the standard library for Linux-based systems.* URL: https://git.musl-libc.org/cgit/musl/.

[9] *perf - Linux profiling with performance counters.* URL: https://perf.wiki.kernel.org/index.php/Main_Page.

[10] Moritz Schloegel et al. «Loki: Hardening Code Obfuscation Against Automated Attacks». In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, ago. 2022, pp. 3055–3073. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/schloegel.

[11] *SecuROM - a Digital Rights Management (DRM) system for content distributed via CD-ROM, DVD-ROM or electronically.* URL: https://support.securom.com/.

[12] Zhanyong Tang et al. «*VMGuards*: A Novel Virtual Machine Based Code Protection System with VM Security as the First Class Design Concern». In: (2018), pp. 1–23.

[13] *The LLVM Compiler Infrastructure.* URL: https://llvm.org/.

[14] *Themida - Advanced Windows software protection system.* URL: https://www.oreans.com/Themida.php.

[15] Mihai Togan, Alin Feraru e Adrian Popescu. «Virtual Machine for Encrypted Code Execution». In: *ECAI 2017 - International Conference – 9th Edition* (2017), pp. 1–6.

[16] *VMProtect software.* URL: https://vmpsoft.com/.

Ringraziamenti

Come conclusione di questo percorso è doveroso da parte mia citare e ringraziare un bel po' di persone, che hanno contribuito a rendere questo percorso unico nel suo genere

Grazie ai miei genitori, per il sostegno e l'affetto costante.

Grazie a tutti i miei parenti: zii, nonni, cugini che si sono sempre interessati su come stesse andando, anche con una semplice domanda.

Grazie alle mie cugine Laura e Cristina, per i consigli di 5 anni fa, sulla scelta del percorso di studi.

Grazie a tutti i "colleghi" conosciuti lungo il percorso, con alcune particolari menzioni: grazie al gruppo "Tiburtina Valley" per aver reso più leggero ogni esame sostenuto durante la laurea triennale.

Grazie a Gian Marco e Simone: senza nulla togliere a nessun altro, con voi c'è stato un confronto continuo durante tutto il percorso, che è divenuta un'amicizia speciale che va profondamente oltre il semplice percorso universitario.

Grazie al mio relatore, che con la sua esperienza mi ha guidato in questo lavoro così importante per me, permettendomi di arricchire ulteriormente le mie conoscenze.

Grazie al mio amico Andrea, con cui durante gli anni ho coltivato discussioni e riflessioni che sono state cibo per la mente.

Grazie a Lorenzo e Lorenzo, per i bei momenti passati in L4 ed L5, dando massima espressione di ciò che siamo davvero, senza filtri.

Infine, un grazie speciale ed immenso va alla ”Banda”: su di loro si potrebbero scrivere trattati molto più lunghi di questa tesi, un collettivo di teste pensanti tutte diverse, eppure unite da un legame inscindibile che va avanti ormai da anni e che, ne sono certo, non si scioglierà mai più.

Noi, pochi. Noi, felici pochi. Noi, manipolo di fratelli!