UNIVERSITÀ DEGLI STUDI DI ROMA
TOR VERGATA

FACOLTÀ DI INGEGNERIA
Corso di Laurea Magistrale in Ingegneria Informatica

TESI DI LAUREA IN
Ingegneria Informatica

Engine Metamorfico per la Protezione della Proprietà Intellettuale

Relatore:
Chiar.mo Prof.
A. Pellegrini

Laureando:
Andrea Pepe
Matr. **0315903**

ANNO ACCADEMICO 2022/2023

Ai miei nonni,

fonti di amore, sostegno e saggezza.

Abstract

Il presente studio di tesi è volto allo sviluppo di una metodologia innovativa per la protezione della proprietà intellettuale del software, che prende forma nel contesto di realizzazione di una macchina virtuale su cui eseguire il software da proteggere. Il livello di difesa viene innalzato grazie all'utilizzo di tecniche di offuscamento e mutazione del codice.

Il componente chiave della macchina virtuale è un engine metamorfico, il cui compito è quello di modificare dinamicamente, ad ogni esecuzione, la struttura dell'eseguibile della macchina virtuale stessa, facendo sì che se ne abbiano sempre versioni differenti. Questo approccio mira a fornire una maggiore resistenza contro le pratiche di *reverse code engineering (RCE)*, costringendo un eventuale attaccante ad affrontare numerose varietà di versioni del software della macchina virtuale, complicando notevolmente qualsiasi tentativo di analisi, manipolazione o sfruttamento del codice.

La progettazione e l'ideazione del motore metamorfico sono state ispirate dall'utilizzo di tecniche di metamorfismo del codice adottate principalmente da alcuni malware, portando ad una strutturazione dell'engine che permette di elevare il livello di astrazione con cui le mutazioni del codice vengono applicate. Infatti, lo strumento realizzato permette di ottenere una rappresentazione intermedia ad alto livello di un eseguibile in formato ELF (*Executable and Linkable Format*), così da poterne manipolare il contenuto, seguendo delle regole di trasformazione. In particolar modo, ci si è concentrati su eseguibili ELF compilati per architetture x86.

Le operazioni eseguite dall'engine metamorfico hanno lo scopo di permettere la mutazione di gruppi di istruzioni macchina in altre istruzioni equivalenti,

tramite processi di espansione, compressione e riordinamento delle istruzioni stesse. In tal modo, si persegue l'obiettivo di modificare l'eseguibile software nella sua forma e aumentarne la complessità strutturale, ma mantenendone intatta la semantica.

Durante lo sviluppo del progetto è stato adottato un approccio *test-driven* all'implementazione, effettuando test funzionali, in particolar modo nelle prime iterazioni di sviluppo, su eseguibili *handcrafted* e su programmi della suite *GNU Binutils* [1]. Successivamente, sono stati svolti dei testi più avanzati con l'obiettivo di verificare la correttezza dello strumento e analizzarne le prestazioni. È stato fatto eseguire l'engine su un campione di malware in formato ELF forniti in input, applicando più passi di trasformazione, generando diverse versioni equivalenti per ogni eseguibile malevolo.
Dopodiché, ogni versione è stata analizzata da un insieme di anti-virus tramite la piattaforma *VirusTotal* [2]. I risultati sono stati messi a confronto per valutare la variazione del *detection rate* dei malware all'incrementare dei passi di trasformazione applicati. Ciò ha permesso anche di individuare quali fossero gli anti-virus maggiormente robusti alle mutazioni indotte dall'engine metamorfico e quali, invece, i più sensibili.
Un'ulteriore metrica di valutazione presa in considerazione per l'analisi riguardante i malware è l'entropia: le trasformazioni del codice applicate tendono a far incrementare lievemente il valore di tale metrica al crescere dei passi di applicazione dell'engine, indice di un aumento di complessità.
Infine, sono stati effettuati dei test funzionali su due benchmark, col fine di valutare l'impatto delle modifiche attuate dal motore metamorfico sulle prestazioni del software, in particolare in termini di variazione dei tempi d'esecuzione.

Nel capitolo 1 di questa tesi si propone un'introduzione alla proprietà intellettuale del software, all'attuale stato dell'arte riguardo tecniche di *binary instrumentation*, nonché una disamina della struttura di un eseguibile in formato ELF e delle istruzioni assembly x86, necessaria per la comprensione di alcune scelte di progettazione e implementazione dell'engine. Nel capitolo 2,

viene proposta un'introduzione al metamorfismo, presentando gli approcci e le metodologie adottate nella fase di design del progetto e illustrando la struttura dell'engine realizzato dal punto di vista concettuale. Nel capitolo 3, viene offerta una descrizione della fase di realizzazione del progetto e vengono esposti i dettagli implementativi. Nel capitolo 4, si descrive la fase di valutazione dell'engine, illustrandone i risultati ottenuti. Infine, nel capitolo 5, vengono mostrati alcuni spunti per sviluppi futuri e migliorie.

Indice

Abstract **ii**

1 Introduzione **1**

1.1 Reverse code engineering: una minaccia 1

1.2 Stato dell'arte 2

1.3 Obiettivi dello studio 3

1.3.1 MorphVM 4

1.4 Il formato ELF 5

1.4.1 ELF header 6

1.4.2 Program header table 7

1.4.3 Section header table 8

1.5 Instruction set x86 10

1.5.1 Formato delle istruzioni 10

1.5.2 Modalità di indirizzamento 14

1.5.3 Prefissi REX 15

2 Progettazione dell'engine **17**

2.1 Il metamorfismo 17

2.1.1 Definizione e origini 17

2.2 MetaPHOR 20

2.2.1 Il metamorfismo in MetaPHOR 20

2.3 Design dell'engine metamorfico 23

2.4 Rappresentazione intermedia di un ELF 26

2.4.1 Sezioni, segmenti ed entry point 27

2.4.2 Disassembling delle istruzioni 27
2.4.3 Tracciamento dei riferimenti nel codice 28
2.4.4 Funzioni e simboli 30
2.4.5 Le jump table 31
2.5 Expander 34
2.5.1 Regole di trasformazione 34

3 Implementazione di riferimento 37

3.1 Metodologie di sviluppo 37
3.2 Parsing di un file ELF 37
3.2.1 *libelf* 38
3.2.2 Le strutture dati 39
3.2.3 Gli header 39
3.2.4 Sezioni 40
3.2.5 Segmenti 43
3.2.6 Istruzioni 44
3.2.7 Simboli e stringhe 46
3.2.8 Funzioni 47
3.2.9 `Elf_program` e `Elf_state` 49
3.3 Il disassembler 50
3.3.1 La struttura `insn_info_x86` 51
3.3.2 Processo di disassemblaggio 53
3.3.3 Riferimenti tra istruzioni 54
3.4 Riscrittura di un ELF 56
3.5 Aggiunta di una `NOP` 58
3.6 Jump table detection 59
3.6.1 Strutture dati di supporto 59
3.6.2 Euristiche di detection 60
3.6.3 Numero di entry di una branch table 64
3.7 L'assembler 66
3.7.1 Effetti delle modifiche su funzioni ed *entry point* 66

3.7.2 Algoritmo di reassembling 67
3.7.3 Fixing delle istruzioni di salto e short jump 69
3.7.4 Fixing dei program header e delle jump table 71
3.7.5 Riscrittura delle sezioni e dei simboli 71
3.8 L'expander 72
3.8.1 Token associati delle istruzioni 73
3.8.2 Regole di trasformazione 74
3.8.3 *Keystone* 75
3.8.4 Istruzioni *garbage* 77

4 Valutazione sperimentale **80**
4.1 Entropia e detection rate dei malware 80
4.1.1 Entropia 81
4.1.2 Detection rate 82
4.2 Prestazioni su benchmark 83
4.2.1 *Pagerank* e *Moon-calc* 83
4.3 Analisi dei risultati 84

5 Conclusioni e future work **92**

A Regole di trasformazione di MetaPHOR **96**

Elenco delle figure

1.1 Layout di un file ELF . . . 6

1.2 Struttura di un'istruzione x86 . . . 10

1.3 Tabella degli opcode delle istruzioni x86 aventi 1 byte di opcode, in modalità a 32 bit . . . 13

1.4 Un esempio di byte ModR/M . . . 15

2.1 Visualizzazione delle diverse versioni di un virus polimorfico . . 18

2.2 Rappresentazione di diverse generazioni di un virus metamorfico 19

2.3 Struttura dell'engine metamorfico di *MorphVM* . . . 23

2.4 Tecnica dello "*spaghetti code*" . . . 25

3.1 Collocazione di una sezione in un file ELF . . . 41

3.2 Organizzazione dei dati in una sezione . . . 41

3.3 Layout di un descrittore `Elf_Data` . . . 42

3.4 Struttura della tabella dei nomi delle sezioni in `.shstrtab` . . . 42

3.5 Tracciamento dei riferimenti tra istruzioni . . . 55

3.6 Byte che rappresentano una branch table nella sezione dati . . . 65

3.7 Opcode a 1 e a 2 byte delle istruzioni x86 . . . 70

3.8 Esito del processo di metamorfosi applicato ad una funzione . . 78

4.1 Entropia dei malware al variare dei passi di trasformazione . . . 86

4.2 Detection rate dei malware al variare dei passi di trasformazione 87

4.3 Anti-virus le cui prestazioni peggiorano maggiormente al variare dei passi di trasformazione dei malware . . . 88

4.4 Anti-virus maggiormente robusti alle trasformazioni dei malware 88

4.5 Tempi di esecuzione del benchmark *Pagerank* al variare dei passi di trasformazione 89
4.6 Tempi di esecuzione del benchmark *Moon-calc* al variare dei passi di trasformazione 89

5.1 Esempio di albero generato da una grammatica 94

Elenco delle tabelle

3.1 Regole di espansione di *MorphVM* 75
3.2 Istruzioni *garbage* 77

4.1 Caratteristiche tecniche della macchina virtuale adottata per i test prestazionali 83
4.2 Entropia dei 10 malware che hanno registrato il maggior incremento al variare dei passi di trasformazione 85
4.3 Detection rate dei 10 malware che hanno registrato il minor detection rate nell'ultimo passo di trasformazione 86

A.1 Regole di trasformazione di MetaPHOR con mapping 1 ad 1 . . 97
A.2 Regole di trasformazione di MetaPHOR con mapping 2 ad 1 . . 101
A.3 Regole di trasformazione di MetaPHOR con mapping 3 ad 1 . . 102
A.4 Regole di trasformazione di MetaPHOR con mapping 3 a 2 . . . 104

Elenco dei listati

2.1 Esempio di espansione di un'istruzione 24
2.2 Correzione di un'istruzione di salto 29
2.3 Esempio di costrutto *switch/case* che genera una jump table . . 32
3.1 Struttura dati `Phdr_table` 40
3.2 Strutture dati per la gestione delle sezioni 43
3.3 Definizione della struttura dati `Segment` 44
3.4 Struttura dati `insn_info_x86` 45
3.5 Strutture dati per la gestione dei simboli 47
3.6 Struttura dei simboli di un file ELF 48
3.7 Definizione della struttura dati `Asm_function` 48
3.8 Definizione delle strutture dati `Elf_program` ed `Elf_state` . . . 50
3.9 Possibili flag per un'istruzione `insn_info_x86` 52
3.10 Strutture di supporto alla jump table detection 60
3.11 Codice assembly x86 che mostra l'utilizzo di una jump table . . 60
3.12 Codice assembly x86 che mostra il pattern per individuare una jump table nel caso di compilazione con ottimizzazione 63
3.13 Definizione della struttura `Jump_table` 66
3.14 Tracciamento degli effetti che l'aggiunta di un'istruzione può avere su funzioni ed *entry point* 67
3.15 Esempio di trasformazione da short jump a long jump 70
3.16 Tipi di token associabili alle istruzioni 74
3.17 Mappa dei nomi dei registri 76

Elenco degli algoritmi

1 Basic JT Heuristic 62

2 Reassemble ELF 68

1. Introduzione

1.1 Reverse code engineering: una minaccia

Nell'attuale panorama digitale, la proprietà intellettuale del software si trova sempre più a rischio a causa di diverse minacce e sfide. Una delle principali è rappresentata dal **reverse code engineering (RCE)**, una pratica che consiste nell'analisi del software a partire dal suo formato eseguibile o binario, al fine di comprenderne la struttura ed il funzionamento interno e, potenzialmente, copiarlo, imitarlo o modificarlo senza autorizzazione.
Il reversing del codice può essere eseguito per vari scopi e non tutti sono considerabili dannosi: infatti, tecniche di ingegneria inversa sono spesso applicate per finalità di studio o ricerca. Inoltre, l'RCE è la principale pratica attraverso cui si effettua analisi del malware, sebbene il rapido sviluppo tecnologico stia portando ad affermarsi sempre più meccanismi di detection e classificazione automatici basati su euristiche, algoritmi di machine learning e reti neurali.

Tuttavia, l'RCE è utilizzato anche in attività illegali come la **pirateria informatica** o la **violazione della proprietà intellettuale**. Quest'ultimo aspetto comporta una serie di rischi per gli sviluppatori e i titolari dei diritti del software, poiché può compromettere la riservatezza delle logiche proprietarie, dei segreti commerciali e degli algoritmi innovativi, rappresentando una seria minaccia sia per quanto riguarda aspetti di cybersecurity che per le eventuali conseguenze in termini di danni economici e d'immagine.

Un noto esempio di violazione della proprietà intellettuale è il caso legale

SCEA v. Hotz [3, 4, 5], risalente al periodo a cavallo tra la fine del 2010 e l'inizio del 2011: l'hacker George Francis Hotz, noto anche col nome di **geohot**, in collaborazione con altri componenti del gruppo hacker e di ricerca sulla sicurezza informatica *"fail0verflow"*, effettuò il **reversing del codice del sistema operativo della PlayStation 3 (PS3)**, con l'obiettivo di ottenere un accesso più approfondito alla piattaforma e consentire l'esecuzione di software non autorizzato.

Grazie all'attività di ingegneria inversa effettuata, Hotz riuscì ad individuare e sfruttare una vulnerabilità nel firmware della PlayStation 3, che gli permise di eseguire codice non firmato e ottenere il controllo completo della console. A ciò ebbe seguito la creazione di una modifica personalizzata del firmware, nota come "custom firmware", che consentiva agli utenti di eseguire software *homebrew* e copie non autorizzate di giochi sulla console. Hotz riuscì ad ottenere e rivelare la **chiave privata** della PlayStation 3, grazie alla quale gli utenti erano in grado di creare e firmare software da eseguire sulle loro console, senza la necessità di utilizzare alcun dispositivo USB esterno: di fatto, era come se il software fosse stato distribuito da Sony. Il gruppo *"fail0verflow"* giustificò lo sviluppo dell'hack affermando che avesse lo scopo di consentire a tutte le versioni del firmware di PS3 di eseguire Linux.

Il reversing effettuato da Hotz attirò chiaramente l'attenzione di Sony, che accusò Hotz di aver violato i diritti d'autore e ritenendo che quanto svolto dall'hacker danneggiasse le misure di sicurezza e protezione della console. Di conseguenza, Sony intentò una causa legale contro di lui.

1.2 Stato dell'arte

In risposta alle problematiche illustrate, spesso si promuove l'utilizzo di licenze d'uso specifiche e la registrazione dei diritti d'autore per garantire la tutela legale del software. Inoltre, diverse tecnologie sono state sviluppate ed adottate per la protezione della proprietà intellettuale (*IPP, Intellectual*

Property Protection), in particolar modo nel contesto dello sviluppo software. Alcuni approcci prevedono l'uso di pratiche particolari, come il *fingerprinting* [6]. Altri, più complessi, ricorrono all'uso di tecniche di offuscamento del codice [7, 8], che rendono più difficile la comprensione e l'analisi del software attraverso la confusione delle sue strutture e logiche interne. La complessità di tali approcci è elevata in quanto richiedono di effettuare *binary instrumentation*, ovvero manipolazione a compile-time dei programmi, operando sul loro codice macchina per aggiungere istruzioni o cambiarne alcune già esistenti. Questa tecnica è utilizzata in diversi campi, come il *behaviour monitoring* [9], l'analisi delle performance [10] e la rilevazione di attacchi [11]. Diversi tool come Pin [12], Dyninst [13], Valgrind [14] e DynamoRIO [15] sono usati per analizzare applicazioni e osservare il loro comportamento. Inoltre, tecniche di anti-disassemblaggio e anti-debugging sono fortemente adottate nell'ambito dello sviluppo di malware, con l'obiettivo di rendere il software difficilmente individuabile dagli strumenti di detection.

È evidente che il rischio per la proprietà intellettuale del software sia elevato: l'avanzamento delle tecnologie di reversing, l'accessibilità a strumenti sempre più sofisticati e l'ampia diffusione di conoscenze nel campo dell'ingegneria inversa richiedono un costante aggiornamento delle strategie di protezione e l'introduzione di metodologie moderne che possano garantire maggior robustezza.

1.3 Obiettivi dello studio

Il lavoro di tesi svolto ha come obiettivo quello di proporre una nuova metodologia per la protezione della proprietà intellettuale del software, che unisca tecniche di offuscamento del codice con il metamorfismo, realizzando uno strumento che contrasti in maniera efficace il reversing dei programmi.

Tutto ciò prende forma nel contesto di sviluppo di un progetto che prevede la realizzazione di **MorphVM**, una virtual machine che possa permettere di eseguire programmi, preservandone la proprietà intellettuale.

1.3.1 MorphVM

L'efficacia di *MorphVM* risiede in due componenti principali: l'utilizzo di un *instruction set* custom per l'esecuzione del software da proteggere e la presenza di un **engine metamorfico** all'interno della macchina virtuale stessa. Quest'ultimo componente ha il compito di modificare, ad ogni esecuzione, la struttura interna del codice eseguibile della macchina, applicando a run-time tecniche di *obfuscation* combinate ad approcci metamorfici, con lo scopo di applicare mutazioni al proprio binario, seguendo delle regole di trasformazione ben precise. Infatti, i cambiamenti devono essere tali per cui la struttura interna dell'eseguibile cambi da versione a versione, ma la semantica sia esattamente equivalente a quella del programma originale.

Il valore aggiunto del **metamorfismo** sta nella capacità della VM di modificare interamente se stessa, compreso l'engine metamorfico, complicandone considerevolmente il reversing e, di conseguenza, garantendo una difesa più robusta alla proprietà intellettuale del software che esegue. Lo strumento diventa tanto più efficace se le regole di trasformazione applicate hanno una componente probabilistica elevata, in particolare se le mutazioni da applicare sono ottenute da un processo generativo.

MorphVM intende essere la concretizzazione di un'idea innovativa e di una metodologia a salvaguardia dell'*intellectual property (IP)* in ambito software, facendo ricorso al metamorfismo, una pratica poco diffusa e che attualmente trova impiego quasi unicamente nello sviluppo di software malevolo.

Sebbene sia evidente come le operazioni che un engine metamorfico di questo genere deve compiere siano collocabili in un contesto di basso livello, la filosofia che viene seguita nel processo di realizzazione dello strumento è quella di elevare il più possibile il livello di astrazione, al fine di ottenere un prodotto altamente modulare, flessibile ed estendibile a diversi formati ed architetture.

Durante la fase di sviluppo, ci si è focalizzati sulla modifica strutturale di programmi eseguibili in formato **ELF (Executable and Linkable Format)**, il formato comune degli eseguibili per sistemi Unix e Linux, compilati per

architetture x86.

Dunque, la progettazione e lo sviluppo dell'engine metamorfico ha richiesto uno studio dettagliato sia del formato ELF che del formato delle istruzioni assembly x86, così da poterne apprendere la terminologia tecnica e la struttura interna. Si propone nelle successive Sezioni 1.4 e 1.5, rispettivamente, la descrizione della struttura che caratterizza un eseguibile ELF e l'organizzazione delle istruzioni dell'instruction set x86, con lo scopo di fornire le conoscenze necessarie per la piena comprensione degli argomenti esposti nel prosieguo della trattazione.

1.4 Il formato ELF

L'**Executable and Linkable Format**, abbreviato in ELF, è un formato standard per file eseguibili, codici oggetto, librerie condivise e core dump. Nel 1999 è stato scelto come formato standard dei file binari per i sistemi Unix e Unix-like. È un formato molto flessibile: supporta sia architetture a 32-bit che a 64-bit, entrambe le tipologie di endianess (little endian e big endian), differenti taglie di indirizzi, così da non escludere nessuna central processing unit (CPU) o instruction set particolare.

Ogni file ELF è composto di un **ELF header**, seguito dai dati. Tali dati possono includere (e tipicamente includono):

- Una **Program header table**, in cui sono descritti zero o più segmenti di memoria usati a run-time;
- Una **Section header table**, che descrive zero o più sezioni.

I segmenti contengono informazioni necessarie a run-time per l'esecuzione del programma, mentre le sezioni ospitano dati importanti per il linking e la rilocazione. Ogni byte nell'intero file può appartenere al massimo ad una sezione; è possibile che ci siano byte "orfani", non appartenenti ad alcuna sezione. La Figura 1.1 illustra nel dettaglio il layout descritto.

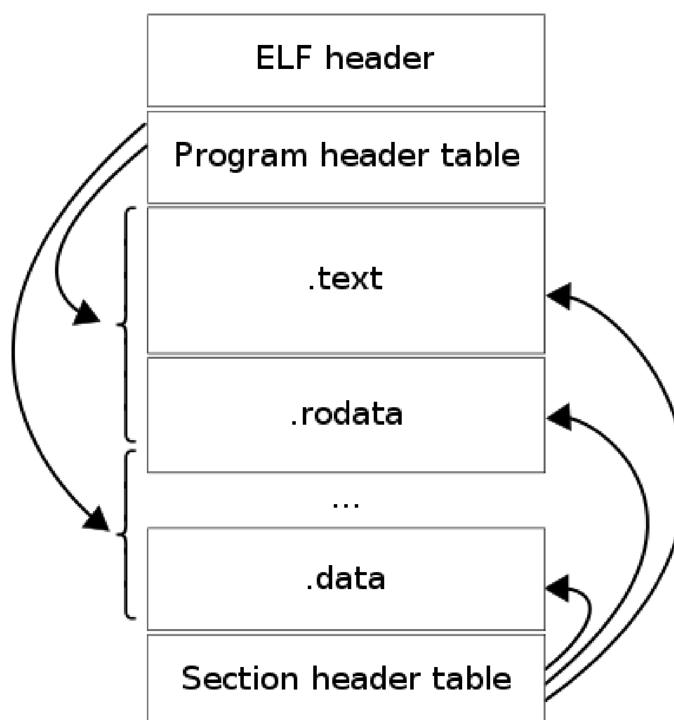

Figura 1.1: Layout di un file ELF

1.4.1 ELF header

L'ELF header, anche detto "file header" o "executable header", si trova all'inizio di ogni file che rispetta il formato ELF. Permette di identificarlo attraverso il **magic number** (`0x7F` seguito dalla stringa `"ELF"` in ASCII, cioè dai bytes `45 4C 46` in esadecimale). Specifica inoltre se l'eseguibile è in formato a 32-bit o a 64-bit; tale informazione è di fondamentale importanza, non solo per l'esecuzione, ma anche per la struttura sia dell'ELF header stesso che della program header table e della section header table. Infatti, le loro taglie in byte e quelle dei campi in esse contenute variano a seconda che ci si trovi in un caso piuttosto che in un altro.

Tra le numerose informazioni contenute nell'ELF header, le seguenti sono quelle ritenute maggiormente rilevanti. Vengono riportate con il nome canonico del rispettivo campo dell'header:

- `e_type`: specifica la tipologia di file eseguibile; i valori più comuni sono `ET_REL`, `ET_EXEC`, `ET_SYN` che specificano rispettivamente un file

rilocabile, un file eseguibile ed uno shared object;

- `e_machine`: valore che indica l'instruction set architecture (ISA) di riferimento; nel caso di studio affrontato, si è interessati ad architetture `x86`;

- `e_entry`: indirizzo di memoria dell'**entry point** da cui il processo inizia l'esecuzione. Se non è specificato un entry point, tale valore è pari a 0;

- `e_phoff`: punta all'inizio della program header table, specificando un offset da interpretare come spiazzamento dall'inizio del file. Tipicamente, la tabella puntata segue immediatamente dopo l'ELF header;

- `e_shoff`: analogamente al campo precedente, punta all'inizio della section header table, che, tipicamente, è posizionata in fondo al file;

- `e_phentsize`: contiene la taglia in byte di una entry della program header table;

- `e_phnum`: indica il numero di entry contenute nella program header table;

- `e_shentsize`: contiene la taglia in byte di una entry della section header table;

- `e_shnum`: indica il numero di entry contenute nella section header table;

- `e_shstrndx`: indica l'indice della entry nella section header table associata alla sezione che contiene i nomi delle sezioni.

1.4.2 Program header table

La program header table istruisce il sistema su come creare l'immagine del processo in memoria. È costituita da `e_phnum` entry, aventi tutte la medesima struttura, ciascuna delle quali contiene metadati relativi ad uno specifico **segmento**. I più importanti tra essi sono:

- `p_type`: identifica il tipo del segmento; valori comuni sono i seguenti:

  - PT_NULL: la entry della program header table non è utilizzata;
  - PT_LOAD: il segmento è caricabile in memoria;
  - PT_DYNAMIC: il segmento contiene informazioni sul linking dinamico;
  - PT_INTERP: il segmento ospita informazioni sull'interprete;
  - PT_PHDR: il segmento contiene la program header table stessa;
  - PT_TLS: il segmento contiene un template per il Thread-Local Storage;

- p_offset: indica l'offset del segmento nell'immagine del file;
- p_vaddr: indica l'indirizzo virtuale del segmento in memoria;
- p_filesz: specifica la taglia in byte del segmento nell'immagine del file; può essere pari a 0;
- p_memsz: specifica la taglia in byte del segmento in memoria; può essere pari a 0.

1.4.3 Section header table

La section header table contiene metadati importanti sulla struttura del file ELF, descrivendo la sua suddivisione in sezioni e le caratteristiche di ognuna di esse. Ogni entry della tabella presenta molteplici campi, tra cui i principali sono:

- sh_name: un offset che punta ad una stringa presente nella sezione **.shstrtab**; la stringa puntata rappresenta il nome della sezione;
- sh_type: specifica la tipologia della entry della section header table, in accordo alla tipologia di sezione associata; i valori più rilevanti che tale campo può assumere sono i seguenti:
  - SHT_NULL: l'entry della section table entry non è usata;

  - `SHT_PROGBITS`: la sezione contiene dati e/o codice di programma;
  - `SHT_SYMTAB`: la sezione ospita una tabella dei simboli;
  - `SHT_STRTAB`: la sezione ospita una tabella di stringhe;
  - `SHT_RELA`: la sezione contiene entry di rilocazione con addendi;
  - `SHT_HASH`: la sezione contiene una symbol hash table;
  - `SHT_NOBITS`: spazio di programma senza dati (bss);
  - `SHT_REL`: la sezione ospita entry di rilocazione senza addendi;
  - `SHT_INIT_ARRAY`: la sezione contiene un array di costruttori;
  - `SHT_FINI_ARRAY`: la sezione contiene un array di distruttori;

- `sh_flags`: flag che specificano gli attributi della sezione e i permessi d'accesso ad essa associati; i flag sono combinabili tra loro. Si riportano di seguito i principali:
  - `SHF_WRITE`: la sezione è scrivibile;
  - `SHF_ALLOC`: la sezione occupa memoria durante l'esecuzione;
  - `SHF_EXECINSTR`: la sezione contiene codice eseguibile;
  - `SHF_STRINGS`: la sezione contiene stringhe terminate da un carattere nullo (`'\0'`);
- `sh_addr`: specifica l'indirizzo virtuale della sezione in memoria, per sezioni che sono caricate in memoria;
- `sh_offset`: offset della sezione nell'immagine del file;
- `sh_size`: taglia in byte della sezione; può essere 0;
- `sh_addralign`: indica l'allineamento richiesto per la sezione. Questo campo deve necessariamente essere un potenza di due.

1.5 Instruction set x86

In questa sezione, si fornisce una descrizione sufficientemente dettagliata del **formato delle istruzioni x86** [16, Chapter 2] e dei campi che compongono un'istruzione, evidenziando in che modo essi determinano il comportamento e l'operazione eseguita. La conoscenza di questi dettagli è un requisito fondamentale per comprendere a pieno le tecniche di offuscamento e trasformazione applicate dall'engine metamorfico.

Dal punto di vista logico, un'istruzione x86 può essere suddivisa in 3 componenti principali:

- prefissi: aggiungono informazioni che modificano il comportamento dell'istruzione;
- opcode: identifica l'istruzione e determina l'operazione da eseguire;
- operandi: rappresentano i dati su cui l'istruzione deve operare.

Tuttavia, la loro struttura interna è abbastanza articolata e variabile.

1.5.1 Formato delle istruzioni

Un'istruzione x86 segue il formato rappresentato in Figura 1.2 e può avere una dimensione variabile da un minimo di 1 byte fino ad un massimo di 15 byte.

Figura 1.2: Struttura di un'istruzione x86

Prefissi

I **prefissi** sono un componente opzionale e vengono utilizzati per fornire informazioni aggiuntive sull'istruzione. Precedono sempre l'opcode dell'istruzione e possono essere suddivisi in quattro gruppi. Per ogni istruzione è utile includere al massimo un codice di prefisso per ognuno dei quattro gruppi.

- **Gruppo 1**: fanno parte di questo gruppo il prefisso `LOCK`, codificato dal byte esadecimale `F0`; figurano inoltre prefissi di ripetizione come `REPNE, REPNZ, REP, REPE, REPZ`;
- **Gruppo 2**: appartengono a questo gruppo prefissi utilizzati per *segment override* e per fornire dei suggerimenti al processore sul path di codice più probabile da seguire in caso di istruzioni di salto condizionato;
- **Gruppo 3**: prefisso che permette di indicare taglia degli operandi a 16 bit; è anche detto *operand-size override prefix* e codificato dal byte `66`;
- **Gruppo 4**: vi appartiene l'*address-size override prefix* (byte `67`), il quale permette di modificare indirizzamento da 32/64 a 16 bit.

Tipologie di prefissi particolari sono invece i **prefissi REX e VEX**, che, tra le diverse funzionalità che ricoprono, permettono l'estensione all'uso di registri a 64 bit e di registri vettoriali.

Opcode

Gli opcode possono variare in termini di lunghezza, da 1 a 3 byte. Inoltre, in casi particolari, 3 bit aggiuntivi di opcode possono essere codificati all'interno del byte `ModR/M`. Ogni byte dell'opcode può contenere informazioni specifiche che influenzano il comportamento dell'istruzione e codificare al proprio interno anche informazioni riguardanti gli operandi come, ad esempio, la taglia degli spiazzamenti, l'indicazione di un registro operando, codici che specificano condizioni o estensioni in segno.

Gli **opcode ad 1 byte** rappresentano le istruzioni più semplici e comuni nell'instruction set x86. Possono includere operazioni come l'assegnazione di

valori ai registri, il trasferimento di dati tra registri, operazioni logiche (`AND, OR, XOR`), operazioni aritmetiche semplici e altre operazioni di base.

Gli **opcode a 2 byte** consentono di eseguire operazioni più complesse e specifiche, come operazioni su registri di controllo, istruzioni di gestione delle eccezioni ed operazioni in virgola mobile. Nel caso di istruzioni general-purpose e SIMD (Single Instruction Multiple Data), la struttura dell'opcode a 2 byte è caratterizzata dalla presenza dell'*escape byte* `0F` come primo byte di opcode. Esso può essere preceduto o meno da prefissi, e richiede la presenza di un ulteriore byte di opcode successivo.

Gli **opcode a 3 byte** sono utilizzati per istruzioni specifiche, come quelle relative ad operazioni estese, l'accesso a registri di controllo o debug o l'esecuzione di operazioni privilegiate. Il loro formato è simile a quelli a 2 byte, prevedendo sempre l'utilizzo dell'*escape byte* come primo byte dell'opcode che però deve essere seguito da due byte ulteriori. Un esempio è dato dall'istruzione `PHADDW` per registri XMM, codificata da `66 0F 38 01`. Notare che il primo byte (`66`) è un prefisso mandatorio.

In Figura 1.3 è riportata una tabella che illustra l'associazione tra istruzioni e opcode di taglia pari ad un singolo byte.

I byte ModR/M e SIB

Molteplici istruzioni che fanno riferimento ad un operando in memoria o a registri utilizzano un byte aggiuntivo all'opcode, detto byte **ModR/M**, che specifica la forma di indirizzamento o i registri da usare come operandi. Esso può essere suddiviso in tre campi:

- *mod*: si combina con il campo *r/m* per formare 32 possibili valori: 8 registri e 24 modalità di indirizzamento;
- *reg*: può essere usato per specificare un registro oppure 3 bit aggiuntivi di opcode; il suo significato è da desumere dall'opcode dell'istruzione;
- *r/m*: può specificare un registro come operando oppure può essere combinato con il campo *mod* per codificare una modalità di indirizzamento.

ADD Eb Gb 00	ADD Ev Gv 01	ADD Gb Eb 02	ADD Gv Ev 03	ADD AL Ib 04	ADD eAX Iv 05	PUSH ES 06	POP ES 07	OR Eb Gb 08	OR Ev Gv 09	OR Gb Eb 0A	OR Gv Ev 0B	OR AL Ib 0C	OR eAX Iv 0D	PUSH CS 0E	TWOBYTE 0F
ADC Eb Gb 10	ADC Ev Gv 11	ADC Gb Eb 12	ADC Gv Ev 13	ADC AL Ib 14	ADC eAX Iv 15	PUSH SS 16	POP SS 17	SBB Eb Gb 18	SBB Ev Gv 19	SBB Gb Eb 1A	SBB Gv Ev 1B	SBB AL Ib 1C	SBB eAX Iv 1D	PUSH DS 1E	POP DS 1F
AND Eb Gb 20	AND Ev Gv 21	AND Gb Eb 22	AND Gv Ev 23	AND AL Ib 24	AND eAX Iv 25	ES: 26	DAA 27	SUB Eb Gb 28	SUB Ev Gv 29	SUB Gb Eb 2A	SUB Gv Ev 2B	SUB AL Ib 2C	SUB eAX Iv 2D	CS: 2E	DAS 2F
XOR Eb Gb 30	XOR Ev Gv 31	XOR Gb Eb 32	XOR Gv Ev 33	XOR AL Ib 34	XOR eAX Iv 35	SS: 36	AAA 37	CMP Eb Gb 38	CMP Ev Gv 39	CMP Gb Eb 3A	CMP Gv Ev 3B	CMP AL Ib 3C	CMP eAX Iv 3D	DS: 3E	AAS 3F
INC eAX 40	INC eCX 41	INC eDX 42	INC eBX 43	INC eSP 44	INC eBP 45	INC eSI 46	INC eDI 47	DEC eAX 48	DEC eCX 49	DEC eDX 4A	DEC eBX 4B	DEC eSP 4C	DEC eBP 4D	DEC eSI 4E	DEC eDI 4F
PUSH eAX 50	PUSH eCX 51	PUSH eDX 52	PUSH eBX 53	PUSH eSP 54	PUSH eBP 55	PUSH eSI 56	PUSH eDI 57	POP eAX 58	POP eCX 59	POP eDX 5A	POP eBX 5B	POP eSP 5C	POP eBP 5D	POP eSI 5E	POP eDI 5F
PUSHA 60	POPA 61	BOUND Gv Ma 62	ARPL Ew Gw 63	FS: 64	GS: 65	OPSIZE: 66	ADSIZE: 67	PUSH Iv 68	IMUL Gv Ev Iv 69	PUSH Ib 6A	IMUL Gv Ev Ib 6B	INSB Yb DX 6C	INSW Yz DX 6D	OUTSB DX Xb 6E	OUTSW DX Xv 6F
JO Jb 70	JNO Jb 71	JB Jb 72	JNB Jb 73	JZ Jb 74	JNZ Jb 75	JBE Jb 76	JA Jb 77	JS Jb 78	JNS Jb 79	JP Jb 7A	JNP Jb 7B	JL Jb 7C	JNL Jb 7D	JLE Jb 7E	JNLE Jb 7F
ADD Eb Ib 80	ADD Ev Iv 81	SUB Eb Ib 82	SUB Ev Ib 83	TEST Eb Gb 84	TEST Ev Gv 85	XCHG Eb Gb 86	XCHG Ev Gv 87	MOV Eb Gb 88	MOV Ev Gv 89	MOV Gb Eb 8A	MOV Gv Ev 8B	MOV Ew Sw 8C	LEA Gv M 8D	MOV Sw Ew 8E	POP Ev 8F
NOP 90	XCHG eAX eCX 91	XCHG eAX eDX 92	XCHG eAX eBX 93	XCHG eAX eSP 94	XCHG eAX eBP 95	XCHG eAX eSI 96	XCHG eAX eDI 97	CBW 98	CWD 99	CALL Ap 9A	WAIT 9B	PUSHF Fv 9C	POPF Fv 9D	SAHF 9E	LAHF 9F
MOV AL Ob A0	MOV eAX Ov A1	MOV Ob AL A2	MOV Ov eAX A3	MOVSB Xb Yb A4	MOVSW Xv Yv A5	CMPSB Xb Yb A6	CMPSW Xv Yv A7	TEST AL Ib A8	TEST eAX Iv A9	STOSB Yb AL AA	STOSW Yv eAX AB	LODSB AL Xb AC	LODSW eAX Xv AD	SCASB AL Yb AE	SCASW eAX Yv AF
MOV AL Ib B0	MOV CL Ib B1	MOV DL Ib B2	MOV BL Ib B3	MOV AH Ib B4	MOV CH Ib B5	MOV DH Ib B6	MOV BH Ib B7	MOV eAX Iv B8	MOV eCX Iv B9	MOV eDX Iv BA	MOV eBX Iv BB	MOV eSP Iv BC	MOV eBP Iv BD	MOV eSI Iv BE	MOV eDI Iv BF
#2 Eb Ib C0	#2 Ev Ib C1	RETN Iw C2	RETN C3	LES Gv Mp C4	LDS Gv Mp C5	MOV Eb Ib C6	MOV Ev Iv C7	ENTER Iw Ib C8	LEAVE C9	RETF Iw CA	RETF CB	INT3 CC	INT Ib CD	INTO CE	IRET CF
#2 Eb 1 D0	#2 Ev 1 D1	#2 Eb CL D2	#2 Ev CL D3	AAM Ib D4	AAD Ib D5	SALC D6	XLAT D7	ESC 0 D8	ESC 1 D9	ESC 2 DA	ESC 3 DB	ESC 4 DC	ESC 5 DD	ESC 6 DE	ESC 7 DF
LOOPNZ Jb E0	LOOPZ Jb E1	LOOP Jb E2	JCXZ Jb E3	IN AL Ib E4	IN eAX Ib E5	OUT Ib AL E6	OUT Ib eAX E7	CALL Jz E8	JMP Jz E9	JMP Ap EA	JMP Jb EB	IN AL DX EC	IN eAX DX ED	OUT DX AL EE	OUT DX eAX EF
LOCK: F0	INT1 F1	REPNE: F2	REP: F3	HLT F4	CMC F5	#3 Eb F6	#3 Ev F7	CLC F8	STC F9	CLI FA	STI FB	CLD FC	STD FD	#4 INC/DEC FE	#5 INC/DEC FF

Figura 1.3: Tabella degli opcode delle istruzioni x86 aventi 1 byte di opcode, in modalità a 32 bit

A volte, alcune combinazioni con il campo *mod* sono usate per esprimere informazioni aggiuntive all'opcode per determinate istruzioni.

Alcune codifiche del byte ModR/M richiedono un secondo byte di indirizzamento, il byte **SIB** (Scale-Index-Base). Ad esempio, le forme di indirizzamento del tipo *base-plus-index* e *scale-plus-index* a 32 bit richiedono il byte SIB. Esso include i seguenti campi:

- *scale*: specifica il fattore di scala;
- *index*: indica il codice del registro usato come indice;
- *base*: specifica il codice del registro usato come base.

L'indirizzo è ottenuto nel seguente modo:

$$indirizzo = base + scale \cdot index$$

In un indirizzamento del tipo $[RAX + RCX \cdot 4]$, il registro RAX contiene la base, mentre il fattore di scala è pari a 4 e l'indice è contenuto nel registro RCX. I 3 valori possono essere combinati in diversi modi; una descrizione completa è fornita in apposite tabelle presenti nella documentazione ufficiale [16, Chapter 2].

È da sottolineare come, in caso di indirizzamento *RIP-relative*, l'indirizzo specificato non sia da intendersi come indirizzo assoluto, bensì come offset a partire dall'indirizzo del byte immediatamente successivo all'istruzione.

Displacement e immediati

Alcune forme di indirizzamento includono uno spiazzamento (*displacement*) da specificare immediatamente dopo il byte ModR/M o il byte SIB, qualora quest'ultimo fosse presente. Se è richiesto un displacement, esso può occupare 1, 2 oppure 4 byte. Se l'istruzione specifica un operando immediato (costante), tale operando segue sempre i byte del displacement. Anche l'immediato può essere specificato tramite 1, 2 oppure 4 byte. Quando si opera in modalità a 64 bit, la taglia degli immediati rimane di 32 bit ed il processore li estende in segno a 64 bit prima di utilizzarli.

1.5.2 Modalità di indirizzamento

Le modalità di indirizzamento, come precedentemente descritto, sono determinate dai byte ModR/M e SIB. In particolare, quando il campo *mod* del byte ModR/M è pari a 11 in binario, il primo operando dell'istruzione è un registro general-purpose, XMM o della tecnologia MMX. Ad esempio, se *mod*=11, *r/m*=000, il registro specificato può essere RAX, EAX, AX, AL, MM0 oppure XMM0. Quale dei precedenti, è determinato dall'opcode dell'istruzione e dall'*operand-size attribute*.

Invece, qualora l'istruzione necessiti di un registro come secondo operando, esso può essere specificato dal campo *reg* del byte ModR/M. Il registro può essere, come in precedenza, general-purpose, XMM o MMX.

Ad esempio, il byte ModR/M di valore esadecimale `C8` potrebbe star a rappresentare il registro `EAX` come primo operando e quello `RCX` come secondo. Ciò è mostrato in Figura 1.4.

Mod 11
RM 000
/digit (Opcode); REG = 001
C8H 11001000

Figura 1.4: Un esempio di byte ModR/M

Tuttavia, non c'è una regola generale per specificare quale dei due registri è da intendersi come operando sorgente e quale come operando destinazione. Ciò varia da istruzione ad istruzione e viene determinato sia dall'opcode che da eventuali prefissi.

1.5.3 Prefissi REX

I prefissi REX (*Register Extensions*) sono prefissi di istruzioni utilizzabili in modalità a 64 bit. Essi permettono di specificare registri general-purpose e registri SSE (*Streaming SIMD Extensions*), indicare l'utilizzo di una taglia a 64 bit per gli operandi e specificare *extended control registers*. Un uso comune di tale prefisso si ha quando si intende usare come operandi i registri `r8-r15` disponibili nella modalità a 64 bit. Non tutte le istruzioni richiedono un prefisso REX in modalità a 64 bit, ma esso è necessario soltanto se l'istruzione riferisce uno dei registri estesi o usa operandi a 64-bit. Se il prefisso viene adoperato quando non necessario, viene semplicemente ignorato.

2. Progettazione dell'engine

2.1 Il metamorfismo

2.1.1 Definizione e origini

Il codice metamorfico è tale per la proprietà di fornire in output, quando eseguito, una versione logicamente equivalente del suo stesso codice, sotto determinate interpretazioni (un'interpretazione è da intendersi come un'assegnazione di significato ai simboli di un linguaggio formale). Da questo punto di vista, un codice metamorfico è simile ad un *quine*. Quest'ultimo è definito come un programma che non riceve nulla in input e produce in output una copia del proprio codice sorgente. Ciò che li differenzia è che un codice metamorfico, di solito, non produce in output il suo source code; tipicamente, ciò che esso fornisce in output è del codice macchina.

Il metamorfismo inizia a diffondersi tra la fine degli anni '90 e l'inizio degli anni 2000, trovando applicazione in modo particolare nello sviluppo di **virus**, con lo scopo di aggirare i meccanismi di *pattern recognition* sfruttati dagli anti-virus [17, 18]. I virus metamorfici sono soliti trasformare il proprio codice binario in una rappresentazione intermedia temporanea a cui applicano i cambiamenti necessari; dopodiché, la rappresentazione intermedia modificata viene nuovamente tradotta nel formato binario di partenza.

Il punto chiave è che la procedura descritta viene effettuata dal virus stesso durante la propria esecuzione. In questo modo, anche l'engine metamorfico contenuto nel virus è sottoposto ai cambiamenti e ciò comporta che, potenzialmente, non ci sia nessuna parte del virus che rimanga la stessa tra due

generazioni successive. Esso cambia completamente forma. È questa la principale differenza tra un **virus metamorfico** ed un **virus polimorfico**: quest'ultimo è dotato di un engine polimorfico che muta tutte le parti del virus, escluso se stesso, spesso ricorrendo alla cifratura del codice, utilizzando chiavi di encryption ogni volta differenti. Una chiara distinzione tra virus polimorfici e metamorfici è data dalla messa a confronto delle Figure 2.1 e 2.2.

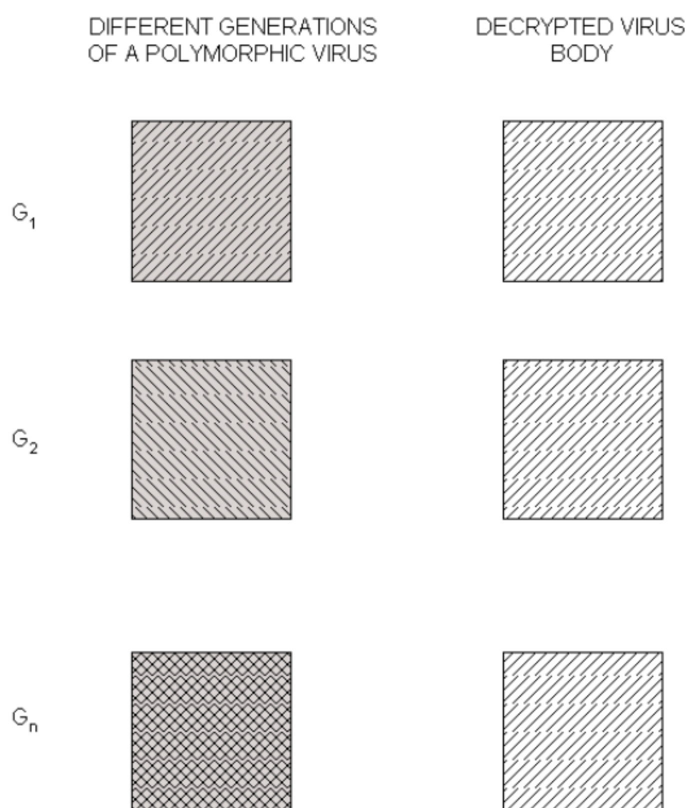

Figura 2.1: Visualizzazione delle diverse versioni di un virus polimorfico

Le mutazioni di forma possono essere ottenute mediante l'applicazione di diverse tecniche: inserendo istruzioni `NOP`, modificando i registri usati come operandi dalle istruzioni (tecnica adottata dal virus *Win95/Regswap*), cambiando il controllo di flusso aggiungendo opportunamente istruzioni di `jmp`, sostituendo istruzioni macchina con altre equivalenti o riordinando istruzioni tra loro indipendenti.

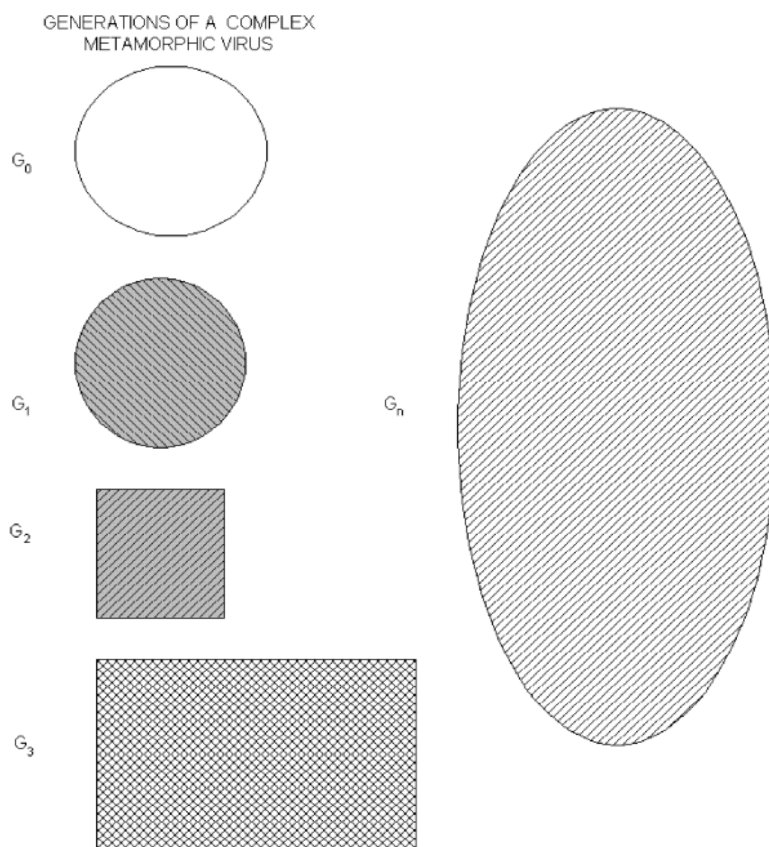

Figura 2.2: Rappresentazione di diverse generazioni di un virus metamorfico

Zmist

Uno dei primi virus metamorfici a fare la sua comparsa è stato **Zmist**, anche conosciuto come **Z0mbie.mistfall**, sviluppato dallo scrittore di virus russo noto col nome di **Z0mbie**. Si tratta del primo virus ad adottare una tecnica denominata "*code integration*" dagli analisti Ferrie e Ször [19]. L'engine "*Mistfall*" contenuto nel virus è in grado di disassemblare file in formato Portable Executable (PE), spostare blocchi di codice in maniera da creare spazio per inserire il proprio binario, inserirsi, rigenerare in maniera opportuna il codice pre-esistente nel PE e i riferimenti ai dati, comprese le informazioni di rilocazione, e ricostruire l'eseguibile infettato.

2.2 MetaPHOR

Uno dei virus metamorfici maggiormente noti, specialmente per la sua certificata abilità nell'evadere tecniche di detection di molteplici anti-virus, è **MetaPHOR** [20, 21]. Il nome è un acronimo per *Metamorphic Permutating High-Obfuscating Reassembler*, che descrive accuratamente le caratteristiche del virus. Inoltre, esso richiama il termine inglese *metaphor*, dal momento che ogni nuova generazione del virus risulta essere una "metafora" della precedente. Il virus è stato scritto in Spagna nel 2002 da **The Mental Driller**, allora membro del gruppo di sviluppatori di virus e worm **29A**.

MetaPHOR è probabilmente l'esempio di codice metamorfico, tra quelli conosciuti, che presenta l'engine più sofisticato, nonché il più studiato e citato in letteratura: É. Filiol *et al.* [17, 22] lo esaminano approfonditamente nello studio delle tecniche di mutazione del codice esistenti, mettendone in risalto la complessità e l'elevato livello di non determinismo.

La realizzazione dell'engine metamorfico per *MorphVM* trova sicuramente in MetaPHOR degli spunti interessanti, in particolar modo per quanto riguarda il design strutturale dell'engine stesso.

2.2.1 Il metamorfismo in MetaPHOR

Come afferma lo stesso *The Mental Driller*:

> *A metamorphic virus is like a 49cc motorbike with a spaceshuttle fuel deposit. In fact, the 90%+ of the code is the metamorphic engine to mutate the little part dedicated to infection, which is somewhat paradoxical.*

Le dimensioni non minimali dell'engine sono dovute principalmente alle diverse operazioni che esso deve compiere e alla sua struttura interna, che può essere suddivisa in cinque apparati, ognuno con una funzione ben precisa:

- *Disassembler*

- *Shrinker*
- *Permutator*
- *Expander*
- *Assembler*

Il **disassembler** è il componente iniziale dell'engine metamorfico; il suo scopo è quello di decodificare ogni istruzione del file eseguibile per conoscerne la lunghezza, i registri che usa e tutte le informazioni ad essa associate.

Lo **shrinker** ha un compito molto delicato ed è estremamente difficile da realizzare: esso deve comprimere il codice disassemblato, generato nella precedente generazione del virus, al fine di evitare che le dimensioni del virus crescano esponenzialmente di generazione in generazione. Può farlo eliminando istruzioni spurie, ridondanti e condensando insiemi di più istruzioni in istruzioni singole e semanticamente equivalenti.

Il **permutator** costituisce la parte basica dell'engine metamorfico e, come suggerito dal nome stesso, ha il compito effettuare permutazioni tra i blocchi di codice del programma, assegnando loro posizioni diverse, ma mantenendo intatto il flusso di esecuzione originariamente codificato, tramite l'utilizzo di istruzioni di salto. Può essere abbinato ad altre forme di metamorfismo, come la sostituzione di istruzioni con altre equivalenti (ad esempio `XOR eax,eax` per `SUB eax,eax`).

L'**expander** ha l'obiettivo inverso dello *shrinker*: ricodifica singole istruzioni in molteplici istruzioni che eseguono la stessa operazione dal punto di vista logico.

Infine, l'**assembler** riassembla quanto costruito dall'*expander*. Aggiusta le istruzioni di salto diretto o di chiamata a funzione (`JMP/CALL`), la lunghezza delle istruzioni, i registri e tutto ciò che risulta essere necessario per il corretto funzionamento della nuova versione. Se, invece, si fa uso di uno pseudo-assembler interno, c'è bisogno che faccia riassemblaggio del codice in linguaggio intermedio e che lo traduca successivamente nel linguaggio specifico del processore, verosimilmente quello dell'eseguibile originale.

In particolare, MetaPHOR fa uso di uno pseudo-assembler intermedio, grazie al quale riesce ad ottenere una rappresentazione delle istruzioni uniforme, tutte di dimensione pari a 16 byte. Ciò permette di avere abbastanza spazio nell'istruzione stessa per tenere traccia dei metadati necessari per l'applicazione delle trasformazioni e per la ricostruzione finale dell'eseguibile: utilizza, ad esempio, un byte detto "*label mark*" che assume il valore 1 quando l'istruzione è puntata da una label. Quest'informazione ritorna utile nella fase di shrinking, per comprendere se una coppia di istruzioni può essere compressa in una singola istruzione o meno: se la seconda delle due è puntata da una label, la compressione non è ammissibile, in quanto la correttezza dell'eseguibile sarebbe compromessa.

Altri aspetti di notevole interesse sono l'utilizzo di *label table* per il salvataggio di riferimenti a porzioni di codice individuati a seguito del disassemblaggio di istruzioni di salto. Tenere traccia delle varie istruzioni target di salti è di vitale importanza sia per valutare la fattibilità delle trasformazioni delle istruzioni, sia per la ricostruzione finale delle istruzioni di salto, che devono essere riscritte specificando gli offset corretti.

Per le fasi di compressione ed espansione delle istruzioni, l'autore ha definito dei mapping tra istruzioni equivalenti, suddividendoli in tre tipologie di trasformazioni, a seconda del numero di istruzioni coinvolte. In particolar modo, per la fase di compressione, sono stati considerati sia mapping che sostistuiscono singole istruzioni con altre equivalenti, che mappe che condensano coppie o triplette di istruzioni in istruzioni singole o doppie.

Un aspetto estremamente interessante che contribuisce in maniera cospicua al non determinismo di MetaPHOR è che le fasi di espansione e compressione sono codificate in modo **ricorsivo**. Ad esempio, un'istruzione di `MOV` di un immediato in un registro (`MOV Reg, Imm`) può essere espansa, tramite l'applicazione di una regola, in una coppia di istruzioni `PUSH Imm / POP Reg`. A questo punto, l'engine prende in considerazione ricorsivamente anche l'eventuale espansione delle "nuove" istruzioni generate: ad esempio, l'istruzione di `PUSH` potrebbe essere a sua volta tradotta in una `MOV` che coinvolge il registro

stack pointer. Chiaramente, viene applicato un limite alla ricorsione durante il processo di trasformazione per garantirne la terminazione.

Un elenco completo delle trasformazioni di istruzioni applicabili dal virus MetaPHOR è consultabile in **Appendice A**.

2.3 Design dell'engine metamorfico

La struttura dell'engine metamorfico di *MorphVM* è senza dubbio ispirata a quella di MetaPHOR, in particolar modo nella definizione dei cinque componenti di cui è formato, come illustrato in Figura 2.3.

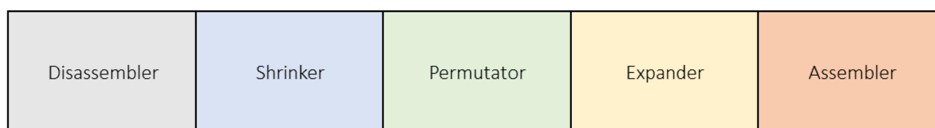

Figura 2.3: Struttura dell'engine metamorfico di *MorphVM*

Infatti, è previsto l'utilizzo di un **disassemblatore** che permetta di avere una rappresentazione in memoria delle istruzioni in assembly x86 che costituiscono il programma, arricchite da informazioni utili per la loro manipolazione. Questo componente iniziale svolge un ruolo fondamentale: è grazie alle operazioni che vengono effettuate in questa fase che si riesce ad ottenere una **rappresentazione intermedia dell'eseguibile ad alto livello**, la quale permette di poter applicare le trasformazioni necessarie per ottenere la mutazione lavorando ad un livello di astrazione più alto rispetto al codice binario di basso livello.

Per dualità, viene considerata anche una fase finale di **assemblaggio** del programma modificato, che consiste nella riorganizzazione della sua struttura e dei suoi metadati, nella ricostruzione corretta dei riferimenti sia tra istruzioni (e.g. istruzioni di salto) che tra istruzioni e dati, per poi infine riscriverlo adeguatamente in formato ELF.

Questi due componenti dell'engine sono i primi ad essere state considerati in fase di sviluppo e di implementazione, per diversi motivi. Innanzitutto, sarebbe risultato impossibile sviluppare la parte di espansione, compressione e permutazione del codice senza aver prima definito la rappresentazione intermedia ottenibile dalla fase di disassemblaggio iniziale. Inoltre, senza lo sviluppo del componente di assemblaggio, non si sarebbe potuto ottenere un file eseguibile di output che è chiaramente necessario per effettuare il testing funzionale dei componenti intermedi del motore metamorfico. Per di più, c'è da considerare che l'introduzione di tecniche metamorfiche che modificano il codice complica notevolmente il debugging e l'individuazione di eventuali errori presenti nel codice sorgente; quindi, iniziare dell'assembler e del disassembler dell'engine permette di ottenere uno scheletro robusto e funzionale dello strumento, opportunamente testato, a cui poi integrare gli altri componenti che introducono un notevole ammontare di non determinismo.

Il successivo componente sviluppato è l'**expander**, che ha il compito di modificare il codice del programma in due principali modi: aggiungendo istruzioni spurie, dette *garbage*, e applicando regole di trasformazione di istruzioni. Un esempio è fornito dal Listato 2.1, in cui un'istruzione di `MOV` del contenuto del registro `rbx` nel registro `rax`, viene espansa in due istruzioni equivalenti: una `PUSH rbx` seguita da una `POP rax`. Inoltre, tra le due istruzioni, ne vengono aggiunte altre due di *garbage*, che non influenzano la semantica del codice, ma ne complicano la forma.

```
1  ; istruzione originale
2  mov rax, rbx
3
4  ; istruzione espansa
5  push rbx
6  nop                  ; garbage
7  mov rbp, rbp         ; garbage
8  pop rax
```

Listato 2.1: Esempio di espansione di un'istruzione

Per quanto riguarda il **permutator**, esso ha il compito di effettuare un riordinamento delle istruzioni, aumentando ulteriormente la complessità strutturale del programma mutato. Anche in questo caso, le tecniche applicabili sono molteplici: si possono invertire di posizione istruzioni tra loro indipendenti oppure effettuare un cambiamento dei registri operandi nei diversi blocchi di istruzioni (è ciò che fa il malware *Regswap*). In alternativa, anziché lavorare a livello di singola istruzione, si può alzare il grado di astrazione e considerare blocchi logici di codice, con l'obiettivo di spezzettarne la struttura disponendoli in maniera tale da sovvertire l'ordine originale, ma garantendo l'equivalenza dal punto di vista del flusso d'esecuzione, tramite l'inserimento di istruzioni di salto incondizionato. Questa tecnica è detta "*spaghetti code*" o "*code transposition*" e ne viene riportato un esempio in Figura 2.4.

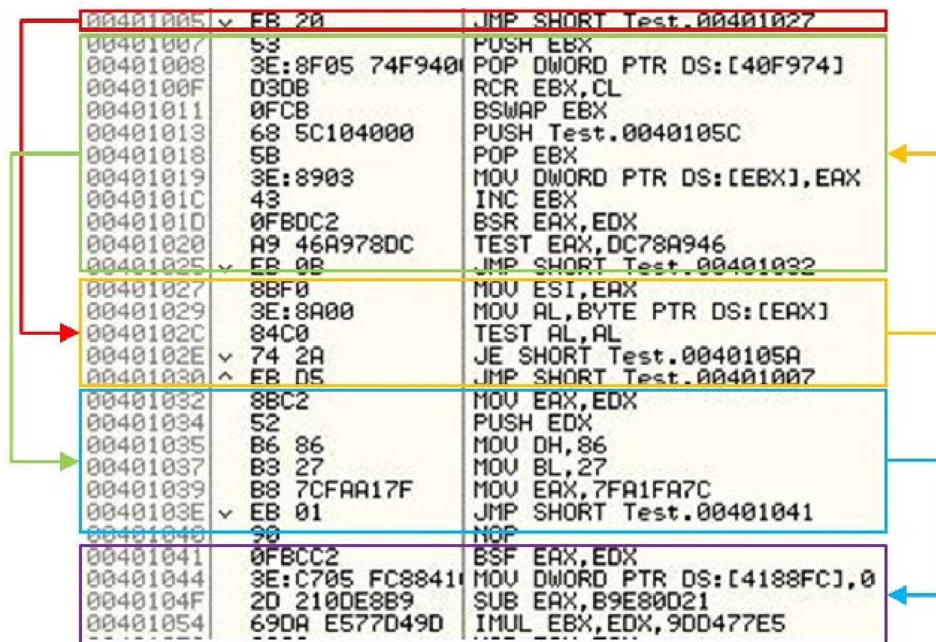

Figura 2.4: Tecnica dello "*spaghetti code*"

Infine, l'ultimo componente è lo **shrinker**. La sua funzione principale è quella di evitare che il codice della macchina virtuale cresca in maniera indefinita di generazione in generazione. Ciò accadrebbe se venisse effettuata unicamente l'espansione delle istruzioni e l'aggiunta di *garbage*. Dunque, per evitare ciò, viene effettuata una compressione di insiemi di istruzioni in insiemi

di dimensione minore, chiaramente sempre garantendo l'equivalenza semantica. Analogamente all'expander, anche qui le trasformazioni sono definite da regole. Tuttavia, considerando che lo shrinker è il componente dell'engine che esegue subito dopo il disassembler, si ritroverà ad operare su un codice strutturato in maniera complessa, specialmente a causa delle azioni compiute dal permutator nelle generazioni precedenti dell'eseguibile. Risulta essere utile, quindi, ricomporre il codice in una struttura logica lineare, invertendo la procedura con cui si ottiene lo *spaghetti code*. Per farlo, si può ricorrere all'utilizzo di *control-flow graph* astratti dal codice assembler, a cui è possibile applicare tecniche di *control-flow flattening*.

La fase iniziale di esecuzione dell'engine metamorfico, non si occupa soltanto del disassemblaggio del codice. Anzi, prima che il processo di disassembling delle istruzioni di programma abbia inizio, è necessario gettare le basi per ottenere una rappresentazione dell'eseguibile che si intende modificare ad un livello più elevato. Infatti, come già visto nella Sezione 1.4, il codice sorgente è presente soltanto in alcune sezioni del file eseguibile. È necessario, quindi, individuare tali sezioni e, più in generale, costruire una rappresentazione in memoria dell'intero programma, interpretando in maniera opportuna i metadati dell'ELF e creando strutture dati che siano di supporto sia alle fasi metamorfiche dell'engine che alla ricostruzione finale del file.

Nelle successive sezioni, verranno illustrate in maniera maggiormente dettagliata le operazioni condotte da ciascun componente dell'engine, dandone una descrizione ad alto livello con un focus particolare sulle scelte di progettazione effettuate.

2.4 Rappresentazione intermedia di un ELF

Ottenere una rappresentazione ad alto livello del file eseguibile è lo step fondamentale dell'intero processo di metamorfosi del programma: infatti, tutti i componenti intermedi dell'engine andranno ad operare su tale rappresentazione per poi, una volta apportate le trasformazioni e le correzioni necessarie,

riportare tutto sul file di output, sovrascrivendo l'eseguibile di partenza.

2.4.1 Sezioni, segmenti ed entry point

Come primo passo per ottenere la rappresentazione dell'ELF, si cerca innanzitutto di estrarre le informazioni ottenibili dagli header: analizzando il contenuto dell'executable header, della program header table e della section header table, si riescono ad individuare le diverse **sezioni** che compongono il file, i **segmenti** che costituiscono l'immagine in memoria del programma e si è in grado di costruire un mapping tra sezioni e segmenti. Ciò permette, in primo luogo, di differenziare i contenuti del file a seconda della tipologia di sezione cui appartengono, dando la possibiltà di individuare quali byte interpretare come codice da disassemblare, quali come dati e quali come metadati. In secondo luogo, durante il processo di mutazione, è necessario, a seguito delle modifiche a cui il contentuo dell'eseguibile viene sottoposto, modificare i metadati contenuti negli header del file (ad esempio, la dimensione e l'offset di inizio di una sezione); per questo motivo, la rappresentazione degli header è anch'essa necessaria. Inoltre, un'altra informazione di cui è importante tenere traccia è l'indirizzo dell'**entry point** del programma. Esso subisce variazioni di posizionamento con elevatissima probabilità e, per garantire il corretto funzionamento dell'ELF prodotto in output, tale riferimento deve essere opportunamente aggiornato.

2.4.2 Disassembling delle istruzioni

A questo punto, suddiviso il contenuto del file in sezioni, si può procedere con la fase di disassembling vera e propria delle istruzioni x86. Si va alla ricerca delle sezioni contenenti codice e, ottenutone il contenuto sotto forma di una sequenza di byte, lo si fornisce in input ad un disassembler che interpreta i byte come istruzioni in assembler x86 e ne restituisce una rappresentazione ad alto livello. Questa operazione è molto importante in quanto il disassemblatore fornisce tutte le informazioni ricavabili sulla singola istruzione, a partire

dall'opcode e dallo mnemonico che la identificano, fino ad arrivare ad eventuali prefissi, byte ModR/M, SIB, displacement e registri usati come operandi.

Inoltre, il disassemblatore adottato è in grado di fornire informazioni ad un alto grado di astrazione, classificando la singola istruzione tramite dei flag combinabili tra loro che, ad esempio, possono indicare che quella specifica istruzione accede in memoria in lettura piuttosto che in scrittura, è un'istruzione che altera il flusso di esecuzione (salto indiretto, condizionato, chiamata a procedura), effettua operazioni logico-aritmetiche, opera condizionatamente a dei controlli sui dati o altre tipologie di comportamenti. Ciò introduce un enorme vantaggio, permettendo di diversificare le operazioni da compiere con le diverse istruzioni, senza la necessità di andare ad effettuare controlli a basso livello.

2.4.3 Tracciamento dei riferimenti nel codice

Disassemblate le istruzioni, nei passi successivi l'engine metamorfico effettua delle operazioni che contribuiscano a guadagnare una vista sul programma orientata al flusso d'esecuzione. In questo contesto, un passo fondamentale è il **tracciamento dei riferimenti**. Se ne distinguono due tipologie:

- riferimento da istruzione ad istruzione;
- riferimento da istruzione a dati.

La prima tipologia viene rilevata in presenza di operazioni di salto o di chiamata a procedura, cioè nel momento in cui c'è un'operazione che può modificare il flusso d'esecuzione del programma passando il controllo ad un altro blocco di codice. In uno scenario simile, è chiaro come sia possibile individuare un'istruzione "sorgente" del salto (semplificando, una `JMP`) e un'altra istruzione "target". Analogamente, per il secondo tipo di riferimento, non si avrà un'istruzione bersaglio, bensì un riferimento (offset o indirizzo) che punta ad un'area presente in una sezione contenente dati. Tuttavia, in questo secondo caso, l'analisi non si limita unicamente alle istruzioni di salto e alle `CALL`, ma coinvolge tutte le istruzioni che indirizzano sezioni non contenenti

codice. Al contrario di quanto viene fatto nell'engine di MetaPHOR tramite la definizione delle *label table*, nell'engine di *MorphVM* i riferimenti non vengono mantenuti all'interno di specifiche tabelle, bensì sono integrati direttamente nella rappresentazione delle singole istruzioni.

L'utilità del tenere traccia dei riferimenti ha un duplice obiettivo: permettere la corretta ricostruzione delle istruzioni di salto in fase di riassemblaggio dell'eseguibile e, allo stesso tempo, marcare le istruzioni target, in maniera analoga a ciò che in MetaPHOR viene fatto tramite il byte "*label mark*".

Per fornire un esempio della necessità di correggere le istruzioni di salto in fase di riassemblaggio dell'ELF, viene riportata nel Listato 2.2 una situazione in cui il processo di espansione porta a dover incrementare l'offset che la `JMP` deve specificare per poter saltare sempre alla stessa istruzione target, così da non modificare la semantica del programma.

```
1  ; codice originale
2  0:  eb 07                  jmp    9 <label>
3  2:  48 89 d8               mov    rax,rbx
4  5:  90                     nop
5  6:  48 85 f6               test   rsi,rsi
6  0000000000000009 <label>:
7  9:  48 89 ca               mov    rdx,rcx
8
9  ; codice dopo la correzione della jmp
10 0:  eb 09                  jmp    b <label>
11 2:  53                     push   rbx      ; mov -> push/pop
12 3:  58                     pop    rax
13 4:  90                     nop
14 5:  48 89 c9               mov    rcx,rcx  ; garbage
15 8:  48 85 f6               test   rsi,rsi
16 000000000000000b <label>:
17 b:  48 89 ca               mov    rdx,rcx
```

Listato 2.2: Correzione di un'istruzione di salto

Notare come l'istruzione di `JMP`, che ha come target la `MOV rdx,rcx`, abbia dovuto incrementare l'offset da `0x07` a `0x09` (specificato come singolo byte dopo l'opcode `0xeb`) per garantire la correttezza dell'eseguibile a seguito della trasformazione di un'istruzione e dell'inserimento di un'istruzione spuria (`MOV rcx,rcx`).

Alla tematica della correzione dei salti è legato un caso particolare molto interessante: nell'esempio presentato (2.2), l'istruzione di `JMP` è una cosiddetta "*short jump*", ovvero un'istruzione di salto che, oltre all'opcode, presenta un solo ulteriore byte per specificare l'offset del salto. Questo tipo di istruzioni sono molto utilizzate dai compilatori perché, nel caso in cui il salto da effettuare abbia come bersaglio porzioni di codice vicine, permette di risparimare sulla dimensione dell'istruzione. Chiaramente, avendo soltanto un byte a disposizione, l'offset specificabile è limitato nel range $[-128, 127]$. Qualora, a seguito delle modifiche apportate dall'engine metamorfico, l'offset necessario dovesse sforare i limiti di tale intervallo, sarebbe necessario sostituire la *short jump* con un'istruzione di salto adeguata, che è completamente differente, anche in termini di opcode. Questo aspetto è trattato nel dettaglio nella Sezione 3.7.3.

2.4.4 Funzioni e simboli

Un concetto di elevata importanza per l'engine di *MorphVM* è quello di **funzione**. La sua accezione è assolutamente equivalente a quella comune nei linguaggi di programmazione e indica una procedura funzionale di codice. L'individuazione delle funzioni all'interno del codice disassemblato rappresenta il vero punto di forza che contraddistingue l'engine metamorfico di *MorphVM* da motori metamorfici usati, ad esempio, nei malware. Quest'ultimi, infatti, sono soliti applicare le trasformazioni direttamente all'intero blocco di istruzioni macchina che compone il programma. Invece, tramite la definizione di funzioni, è possibile applicare politiche più specifiche e, ad esempio, selezionare un pool di funzioni che è vietato modificare. Quest'ultimo aspetto ritornerà essere molto utile per funzioni in cui si fa uso di tabelle dei salti, come descritto in dettaglio nella Sezione 2.4.5. Inoltre, il concetto di funzione permette

di elevare ulteriormente il livello di astrazione, facilitando la progettazione e l'implementazione dell'expander, del permutator e dello shrinker.

L'individuazione delle funzioni avviene grazie all'analisi delle **tabelle dei simboli** del file ELF. Esse contengono informazioni sui simboli definiti ed utilizzati nel file eseguibile, come variabili globali, funzioni e label. Ad ogni simbolo è associato un nome, una tipologia, il valore (indirizzo o offset) ed altre informazioni specifiche. Nel caso particolare delle funzioni, permettono di individuare la posizione iniziale del blocco di codice corrispondente all'interno del file e la sua dimensione in byte. Grazie a queste informazioni, si riesce a rappresentare ogni funzione presente nel file e ad associarle le istruzioni che le appartengono.

Inoltre, l'engine tiene traccia anche degli altri simboli presenti nelle tabelle, arricchendo la rappresentazione complessiva che mantiene dell'eseguibile e permettendo di correggere il contenuto delle *symbol table* in maniera opportuna, qualora lo shifting di istruzioni e sezioni lo renda necessario.

2.4.5 Le jump table

Infine, l'ultimo passo che l'engine compie per completare la rappresentazione intermedia dell'ELF è l'individuazione delle **jump table** (o *branch table*). Le tabelle dei salti sono strutture che vengono automaticamente introdotte da molti compilatori per garantire una migliore efficienza nell'esecuzione del codice, specialmente per la gestione di *switch/case statement* con un elevato numero di *case* corrispondenti a valori molto vicini tra loro. Un esempio di codice sorgente in linguaggio C che dà vita ad una branch table è riportato nel Listato 2.3.

In una struttura *switch/case* standard, ogni caso viene confrontato sequenzialmente con il valore di controllo e, quando viene trovata una corrispondenza, si esegue il blocco di codice associato. Tuttavia, quando il numero di casi diventa elevato, questo approccio è inefficiente, poiché richiede un numero di confronti proporzionale al numero di casi.

```c
void jt_trial(long l){
    switch(l){
      case 0:
        printf("This is case 0\n");
        break;
      case 2:
        printf("This is case 2\n");
        break;
      case 4:
      case 6:
        printf("This is case 4 and 6\n");
        break;
      case 8:
        printf("This is case 8\n");
        break;
      case 10:
        printf("This is case 10\n");
        break;
      case 12:
        printf("This is case 12\n");
        break;
      case 14:
        printf("This is case 14\n");
        break;
      case 16:
        printf("This is case 16\n");
        break;
      case 18:
        printf("This is case 18\n");
        break;
      default:
        printf("Doubling the value this is the result: %ld\n", l+l);
        printf("This is the default case\n");
        break;
    }
}
```

Listato 2.3: Esempio di costrutto *switch/case* che genera una jump table

La branch table risolve questo problema, creando in memoria una tabella di salti, spesso strutturata come un array, in cui sono contenuti gli indirizzi o gli offset da utilizzare in istruzioni di salto per passare il controllo alla porzione di codice associata al *case* corrispondente. Il vantaggio sta nell'utilizzare il valore da confrontare, opportunamente manipolato con shifting o moltiplicazioni per tenere conto della taglia di ogni elemento della tabella, come indice per accedere alla jump table e individuare immediatamente la porzione di codice a cui saltare.

L'individuazione di questi tipi di costrutti, ha una finalità ben precisa: è necessario individuare le tabelle dei salti per poterne modificare opportunamente il contenuto in maniera concorde agli spostamenti delle istruzioni all'interno del programma e garantire così il corretto funzionamento dell'eseguibile di output. Bisogna considerare che la detection delle jump table è assolutamente non banale, dal momento che la loro codifica in istruzioni macchina differisce a seconda del compilatore e, inoltre, assume forme differenti anche in base alle direttive di ottimizzazione specificate in fase di compilazione. Di conseguenza, l'unico modo per rilevarne la presenza è tramite l'utilizzo di **euristiche**.

L'engine di *MorphVM* utilizza due euristiche basate su un approccio di scansione sequenziale delle istruzioni, che effettuano la ricerca di pattern che caratterizzano l'utilizzo di una branch table, individuati grazie alla presenza di determinate istruzioni in uno specifico ordine. Sono state realizzate basandosi su un'analisi del codice prodotto in output dal *GNU C compiler* su sorgenti di prova, utilizzando diverse direttive di ottimizzazione.

Porzioni di codice che fanno uso delle jump table non devono assolutamente essere coinvolte in maniera diretta nel processo di trasformazione dell'ELF, altrimenti, l'engine metamorfico non sarebbe più in grado di riconoscerle in generazioni successive dell'eseguibile. A tal proposito, le funzioni ricoprono un ruolo chiave. Le euristiche di detection vengono eseguite per ogni singola funzione del file così che, qualora venga individuato l'utilizzo di una tabella di salti, ci si possa limitare ad escludere la singola funzione dal processo di metamorfosi, potendo continuare ad agire sulle altre porzioni di codice dell'eseguibile.

Una descrizione approfondita delle jump table e delle euristiche realizzate per individuarle è fornita nella Sezione 3.6.

2.5 Expander

Ottenuta la rappresentazione intermedia dell'eseguibile, si può procedere con la metamorfosi del codice. Per la fase di espansione, si considerano sia la trasformazione di istruzioni che l'aggiunta di istruzioni spurie. Anche in questo caso, l'engine esegue le trasformazioni operando sulle singole funzioni; ciò comporta il grande vantaggio di evitare di dover considerare dei corner case al momento dell'applicazione di determinate regole. Ad esempio, si supponga che esista la seguente regola:

$$
\begin{aligned}
&\text{XOR} \quad \text{Reg}, \text{Reg} \\
&\text{ADD} \quad \text{Reg}, \text{Imm} \\
&\downarrow \\
&\text{PUSH} \quad 0 \\
&\text{POP} \quad \text{Reg} \\
&\text{MOV} \quad \text{Reg}, \text{Imm}
\end{aligned}
$$

Per poter essere applicata, le istruzioni di `XOR` e `ADD` devono essere sequenziali. Tuttavia, se fossero contigue ma appartenenti a funzioni differenti, l'applicazione della trasformazione romperebbe la semantica del programma. Lavorare direttamente con le funzioni, permette di evitare di effettuare queste tipologie di controlli e avere maggiori garanzie riguardo la correttezza del processo di metamorfosi dell'eseguibile.

2.5.1 Regole di trasformazione

In maniera analoga a quanto fatto in MetaPHOR (vedere Appendice A), anche in *MorphVM* vengono definite delle regole di trasformazione delle istru-

zioni ben precise. Esse possono partire da una singola o da molteplici istruzioni. A seconda della tipologia di istruzione da modificare, viene considerato un determinato sottoinsieme di possibili regole e, in maniera casuale, ne viene selezionata una da applicare. Chiaramente, non è detto che tale regola sia effettivamente applicabile, quindi è necessario valutarne di volta in volta la fattibilità.

In seguito al processamento di ogni istruzione, con una piccola probabilità (all'incirca il 5% delle volte) viene deciso di aggiungere un'istruzione *garbage*, per creare ulteriore rumore nel file.

3. Implementazione di riferimento

3.1 Metodologie di sviluppo

L'engine metamorfico di *MorphVM* è stato interamente sviluppato in linguaggio C, adottando un metodologia di sviluppo iterativa e *test-driven*, facendo corrispondere ad ogni iterazione un obiettivo ben preciso da raggiungere. La correttezza e l'efficacia del software sviluppato sono state provate utilizzando file eseguibili in formato ELF esterni, anziché direttamente la macchina virtuale stessa, per agevolare i processi di testing e di debugging. Infatti, proprio il debugging, in un contesto in cui si modificano istruzioni e strutture interne dei file a basso livello, è di vitale importanza ma, allo stesso tempo, risulta essere molto più complesso che in scenari comuni. Per questo motivo, lavorare su file di dimensioni ridotte può permettere di accelerare le fasi di testing, focalizzandosi sugli obiettivi specifici fissati per ciascuna iterazione di sviluppo.

3.2 Parsing di un file ELF

Nella prima iterazione, ci si è concentrati sulla lettura e sul parsing di un file ELF, al fine di ottenerne una rappresentazione logica a runtime. Inoltre, partendo dalla rappresentazione ottenuta, si è provato a riscrivere il contenuto del file senza applicarvi alcuna modifica, con l'obiettivo di ottenere nuovamente un ELF che fosse ancora eseguibile. Sebbene tale procedimento possa

sembrare banale, in realtà non è così scontato: esistono molteplici librerie che permettono di operare su file ELF, molte delle quali si appoggiano sull'utilizzo di strutture dati, macro e interfacce definite nel noto file di intestazione `elf.h`, componente essenziale per il supporto ELF nel kernel Linux. Tuttavia, la maggior parte di esse non sono di facile utilizzo, come *libbfd*, o non permettono di ottenere le capability di riscrittura dell'eseguibile necessarie al caso presentato.

3.2.1 *libelf*

Una delle librerie C maggiormente adoperate per la gestione degli ELF è *libelf*, la quale fornisce delle interfacce per la lettura e la scrittura di file ELF. Il vantaggio maggiore offerto da questa libreria è il livello di astrazione che si riesce ad ottenere tramite l'uso delle API **GELF**, le quali permettono di operare su strutture dati universali utilizzabili sia per eseguibili a 32 bit che a 64 bit.

Tuttavia, il supporto che la libreria fornisce nella riscrittura di un file ELF tradisce quanto affermato nella documentazione stessa di *libelf*, non riuscendo a produrre in output un file realmente eseguibile. Questo non permette di affidarsi completamente all'uso della libreria per generare il file di output, ma è necessaria una gestione custom, descritta in maniera più dettagliata nella Sezione 3.4.

Nonostante ciò, l'utilizzo di *libelf* ha permesso di ottenere in maniera agevole una rappresentazione strutturata degli elementi logici che compongono un file ELF eseguibile, permettendo di definire delle strutture dati fondamentali per l'esecuzione delle funzionalità dell'engine metamorfico.
È da sottolineare come l'utilizzo di una guida all'uso della libreria, scritta da J. Koshy [23], abbia fornito un contributo notevole non solo in fase di implementazione, ma anche per la comprensione di dettagli a basso livello sui diversi componenti presenti all'interno di un file ELF.

3.2.2 Le strutture dati

Come ampiamente descritto nella Sezione 2.4, la prima operazione che l'engine metamorfico effettua è la **lettura del file eseguibile** e il conseguente popolamento delle strutture dati che ne forniscono una rappresentazione logica intermedia, su cui poter poi applicare le tecniche di offuscamento e metamorfismo. Si comincia ad estrarre tutte le informazioni ottenibili analizzando l'eseguibile, in particolare dagli header, seguendo una procedura logica e intuitiva:

1. Inizializzare la libreria *libelf* e ottenere un handle al file
2. Leggere l'executable header
3. Analizzare il contenuto della program header table
4. Recuperare le informazioni necessarie dalla section header table per rappresentare le sezioni
5. Costruire il mapping tra sezioni e segmenti
6. Disassemblare il contenuto delle sezioni eseguibili di tipo `SHT_PROGBITS`
7. Ottenere le stringhe dell'eseguibile
8. Rappresentare la tabella dei simboli

3.2.3 Gli header

Il primo passo della procedura è facilmente ottenibile tramite l'uso dell'API `elf_begin`, invocata specificando di aprire il file ELF sia con permessi di lettura che di scrittura, per poterlo poi sovrascrivere. Analogamente, l'**ELF header** (abbreviato con il termine *ehdr*), si può ottenere con una chiamata a `gelf_getehdr`. Come preannunciato nella Sezione 1.4, tale header viene utilizzato sia per accertarsi che l'ELF sia effettivamente eseguibile e compilato per architetture x86, sia per ottenere informazioni sui successivi metadati da

estrapolare dal file, come la posizione della program header table e della section header table. Inoltre, un ulteriore metadato di grande importanza di cui viene tenuta traccia è l'offset originale all'interno del file a cui è locato l'**entry point** del programma.

Ottenute tali informazioni, si può iniziare ad effettuare un parsing di tutte le componenti del file per poterlo rappresentare in memoria. Viene popolata la struttura dati di tipo `Phdr_table`, riportata nel Listato 3.1, che rappresenta la **program header table** e contiene al suo interno una lista di strutture di tipo `Gelf_Phdr`, ognuna delle quali rappresenta una entry della tabella. Ogni entry racchiude tutti i metadati relativi ad uno specifico segmento del file. Il numero delle entry presenti viene letto dal campo `e_phnum` dell'ELF header e ciascuna di esse è ottenuta invocando l'API `gelf_getphdr` della libreria *libelf*.

```
1  typedef struct Phdr_table{
2      size_t num_entries;            // number of program headers
3      linked_list *list;             // list of GElf_Phdr structures
4      linked_list *segments;         // list of Segment structures
5  } Phdr_table;
```

Listato 3.1: Struttura dati `Phdr_table`

Il passo numero 4, in maniera analoga al precedente, recupera la **section header table** e costruisce una rappresentazione della **tabella delle sezioni**. Le sezioni sono organizzate per indice; dunque, viene letto il numero di elementi della section header table dal campo `e_shnum` dell'ELF header e, effettuando un ciclo a partire dall'indice 0, tutti gli header di sezione vengono recuperati. In aggiunta, vengono costruite anche delle strutture dati che rappresentano le sezioni vere e proprie e tengono traccia del loro contenuto.

3.2.4 Sezioni

Recuperare i metadati delle sezioni e il loro contenuto è un passo fondamentale, senza il quale non sarebbe possibile individuare le porzioni del file che contengono codice da disassemblare e, quindi, non si avrebbero a disposizione le istruzioni x86 a cui applicare le trasformazioni metamorfiche.

Le sezioni di un file ELF hanno un'organizzazione piuttosto particolare: innanzitutto, la section header table è tipicamente posta alla fine del file ELF, mentre le sezioni la precedono. Tale layout è ben mostrato in Figura 3.1.

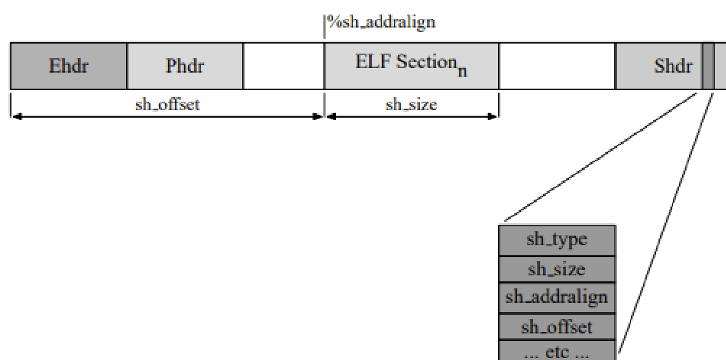

Figura 3.1: Collocazione di una sezione in un file ELF

Ad ogni sezione è associato un descrittore di tipo `Elf_Scn`, attraverso il quale è possibile recuperare i dati che la sezione stessa contiene, sotto forma di una lista di descrittori `Elf_Data`. Ogni sezione può avere associati zero o più di questi descrittori, anche se, in pratica, è molto raro avere a che fare con una sezione che ne abbia più di uno. Ogni `Elf_Data` è una struttura composta da diversi campi: `d_buf` è il buffer che contiene i dati, `d_size` indica la taglia del buffer, `d_off` specifica l'offset al quale si trovano i dati contenuti nel buffer rispetto all'inizio della sezione e, infine, `d_align` fornisce informazioni riguardo l'allineamento richiesto dal buffer (ad esempio ai 4 bytes). Quanto appena descritto, è visibile graficamente nelle Figure 3.2 e 3.3.

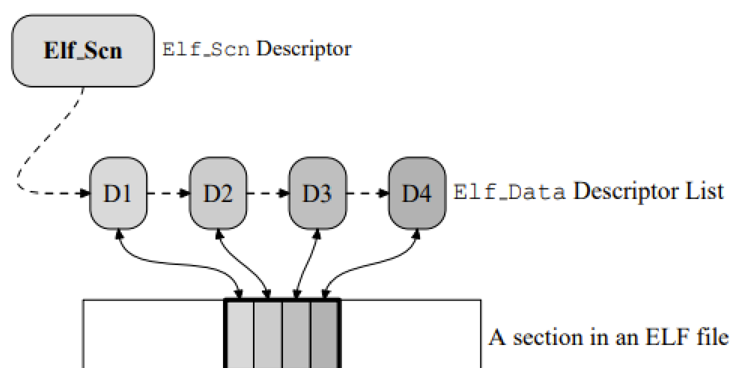

Figura 3.2: Organizzazione dei dati in una sezione

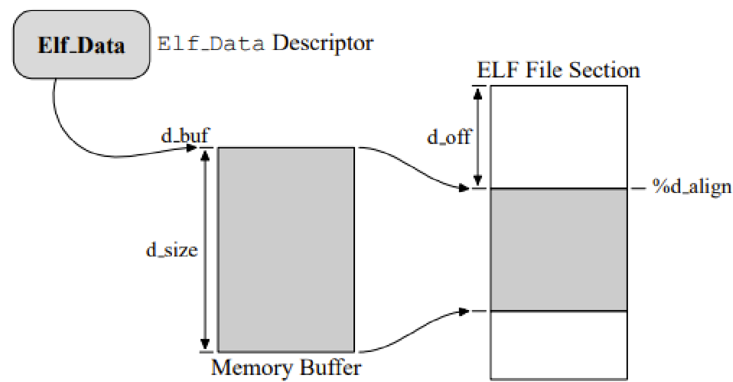

Figura 3.3: Layout di un descrittore `Elf_Data`

Dunque, per rappresentare la tabella delle sezioni, viene popolata un'apposita struttura dati di tipo `Sec_table`, la quale contiene una lista collegata di elementi di tipo `Section` che mantengono un riferimento all'header della sezione, a metadati aggiuntivi, come, ad esempio, il nome della sezione stessa, e una lista di `Section_data`, ulteriori strutture che hanno il compito di rappresentare un singolo descrittore di tipo `Elf_Data`, quindi una porzione del contenuto della sezione. Inoltre, sono presenti campi addizionali che svolgono un ruolo cruciale in fase di modifica o riscrittura dell'ELF: ad esempio,viene tenuta traccia di una lista delle istruzioni disassemblate per quelle sezioni che contengono codice.

I nome delle sezioni in un file ELF sono presenti in un'apposita sezione denominata "*.shstrtab*", in cui appaiono come una sequenza di stringhe *null-terminated* (Figura 3.4). Ogni stringa è puntata da un indice, o, per meglio dire, un offset, contenuto nel campo `sh_name` del section header della sezione a cui il nome è associato. Le definizioni delle strutture dati menzionate per la gestione delle sezioni sono riportate in linguaggio C nel Listato 3.2.

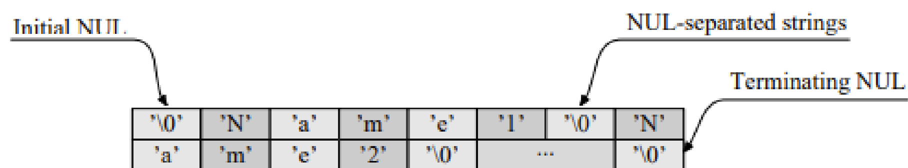

Figura 3.4: Struttura della tabella dei nomi delle sezioni in `.shstrtab`

```
typedef struct Section_data{
  Elf_Data *data;                 // content of a section
  linked_list *instructions;      // list of disassembled instructions
  size_t displacement;            // byte shift from the original position
  size_t old_size;                // original buffer size
  size_t padding;                 // needed padding
}Section_data;

typedef struct Section{
  Elf_Scn *scn;                   // the section handler
  GElf_Shdr *shdr;                // the section header
  char *name;                     // the section name
  linked_list *data;              // list of Section\_data
  size_t num_data;                // number of Section_\Data
  int64_t section_displacement;   // byte shift from the original position
  size_t old_size;                // original section size
}Section;

typedef struct Sec_table{
  linked_list *secs;              // List of Section structures
  size_t num_entries;             // Number of sections
}Sec_table;
```

Listato 3.2: Strutture dati per la gestione delle sezioni

3.2.5 Segmenti

Lo step numero 5 della procedura di lettura e parsing del file eseguibile prevede la costruzione di un **mapping tra sezioni e segmenti**. Questa mappa individua quali sezioni appartengono ai diversi segmenti del file eseguibile, andando a confrontare gli offset e le dimensioni di ognuno di essi: una sezione è ritenuta appartenere ad un segmento se è completamente contenuta tra gli estremi del segmento, definiti nel seguente modo:

- estremo inferiore: offset rispetto all'inizio del file ELF a cui il segmento inizia; è specificato nel campo `p_offset` del rispettivo program header;
- estremo superiore: determinato dalla somma tra l'estremo inferiore e la taglia del segmento, indicata dal campo `p_filesz` nel program header.

Il mapping viene salvato popolando delle strutture dati di tipo `Segment`, le quali contengono le informazioni necessarie per la gestione dei segmenti, tra cui la lista delle sezioni che appartengono a quel segmento. La definizione della struttura dati è presentata nel Listato 3.3. La procedura appena descritta è necessaria per ricostruire correttamente l'eseguibile dopo il processo di metamorfosi: verosimilmente, le sezioni varieranno di posizione e dimensione

rispetto al loro status originale e questo cambiamento deve essere riportato anche per quanto riguarda i segmenti che ospitano a runtime il contenuto delle sezioni, modificando in maniera opportuna i metadati nei program header.

```
typedef struct Segment{
  int index;                  // index of the segment, starting from 0
  uint64_t start_index;       // segment start offset
  uint64_t end_index;         // segment end offset
  uint64_t size;              // size in bytes of the segment
  linked_list *sections;      // list of sections belonging to the segment
}Segment;
```

Listato 3.3: Definizione della struttura dati `Segment`

3.2.6 Istruzioni

Giunti al passo 6 della procedura, si può eseguire il disassemblaggio del codice del programma. Vengono individuate le **sezioni** che contengono codice eseguibile e i loro byte vengono analizzati da un disassemblatore di istruzioni x86 che fornisce in output una lista ordinata di strutture dati (di tipo `insn_info_x86`), ciascuna delle quali fornisce una descrizione ad alto livello di una singola istruzione assembly. La definizione completa della struttura è consultabile nel Listato 3.4.

Queste strutture dati rappresentano, di fatto, gli elementi principali su cui l'intero engine metamorfico opera, applicando le regole di trasformazione, le permutazioni e le diverse tecniche di offuscamento precedentemente descritte.

Come già accennato nelle Sezioni 2.4.2 e 2.4.3, questa fase non prevede unicamente il disassembling del codice, ma vengono effettuate molteplici operazioni, come il tracciamento dei riferimenti tra le istruzioni stesse, che contribuiscono ad arricchire la rappresentazione intermedia del file eseguibile necessaria all'engine. Tuttavia, dal momento che è necessario presentare prima ulteriori strutture dati per permettere una piena comprensione delle operazioni svolte nel processo di disassemblaggio, se ne rimanda una descrizione dettagliata, riportata in Sezione 3.3.

```c
typedef struct insn_info_x86 {
  unsigned long flags;
  unsigned char insn[15];
  unsigned char opcode[2];
  char mnemonic[16];
  unsigned long initial;
  unsigned long insn_size;
  unsigned long addr;
  unsigned long span;
  bool has_index_register;
  unsigned char ireg;
  char ireg_mnem[8];
  bool has_base_register;
  unsigned char breg;
  char breg_mnem[8];
  bool has_scale;
  unsigned long scale;
  unsigned long disp_offset;
  int disp_size;
  long long disp;
  unsigned long immed_offset;
  int immed_size;
  unsigned long long immed;
  unsigned int opcode_size;
  int32_t jump_dest;
  bool uses_rip;
  unsigned char rex;
  unsigned char modrm;
  unsigned char sib;
  unsigned char sse_prefix;
  unsigned char prefix[4];
  bool dest_is_reg;
  unsigned char reg_dest;
  int opd_size;
  int op[3];

  // Additional info wrt disassembler
  unsigned long long orig_addr;
  unsigned long long new_address;
  int64_t displacement;
  struct insn_info_x86* jump_target;
  linked_list *target_of;
  int jump_op_size;
  unsigned long jump_op_start;
  Section_data *sec_data;
  Asm_function *function;
  bool uses_jt;
  void *jt;
} insn_info_x86;
```

Listato 3.4: Struttura dati insn_info_x86

3.2.7 Simboli e stringhe

La procedura continua con il recupero delle **stringhe del programma**. Si va quindi alla ricerca, tra tutte le sezioni individuate, della sezione di tipo `SHT_STRTAB` e di nome "*.strtab*". Infatti, la maggior parte dei compilatori e assemblatori comunemente usati nominano in tal modo la sezione contenente la tabella delle stringhe del programma. Se tale sezione viene trovata, si procede ad estrarre le stringhe dai buffer delle strutture `Elf_Data` associate alla sezione e ad organizzarle in una lista collegata. Anche in questo caso, la tabella delle stringhe contenuta in `.strtab` è strutturata in maniera analoga a quella contenente i nomi delle sezioni, mostrata in Figura 3.4.

Infine, nell'ultimo step della procedura, come anticipato nella Sezione 2.4.4, si controlla l'esistenza di una sezione di tipo `SHT_SYMTAB`, contenente cioè una **tabella dei simboli**. Di solito, tale sezione ha nome "*.symtab*". Se esiste, viene calcolato il numero di entry contenute nella tabella, dividendo la taglia della sezione (`sh_size`) per la dimensione di ciascuna entry, riportata nel campo `sh_entsize` dell'executable header. Ogni elemento della tabella rappresenta un simbolo, ottenuto invocando l'API `gelf_getsym`, la quale restituisce un riferimento ad una struttura di tipo `Elf64_Sym`. Tale struttura ospita i metadati che descrivono la tipologia e la visibilità del simbolo e il valore che ha associato.

A questo punto, dato che il valore del simbolo è tipicamente un puntatore ad altri elementi presenti nel file ELF, si va alla ricerca del blocco dati appartenente alla specifica sezione a cui il simbolo fa riferimento, per tenerne traccia. Bisogna considerare che non tutti i simboli riferiscono elementi locati in sezioni: ad esempio, simboli con campo `st_shndx` pari a `SHN_UNDEF, SHN_ABS,` e `SHN_COMMON` non lo fanno. Vengono dunque istanziate delle strutture dati di tipo `Symbol` che mantengono un riferimento sia all'`Elf64_Sym` che alla `Section_data` contenente quanto riferito dal simbolo (vedere il Listato 3.5). Dettagli aggiuntivi sulla struttura delle tabelle dei simboli sono consultabili nella documentazione ufficiale [24].

```
typedef struct {
  Elf64_Word  st_name;
  unsigned char st_info;
  unsigned char st_other;
  Elf64_Half  st_shndx;
  Elf64_Addr  st_value;
  Elf64_Xword st_size;
} Elf64_Sym;

typedef struct Symbol{
  GElf_Sym sym;              // the symbol
  Section_data *sec_data;    // where the content referred by the symbol is
}Symbol;
```

Listato 3.5: Strutture dati per la gestione dei simboli

3.2.8 Funzioni

Di particolare interesse sono i simboli di tipo `ST_FUNC`, i quali indicano funzioni. Come ampiamente descritto in Sezione 2.4.4, tenere traccia delle funzioni nella rappresentazione dell'ELF offre numerosi vantaggi e permette di elevare il livello di astrazione a cui l'engine metamorfico opera.

Per facilitare la comprensione delle operazioni effettuate per definire e rappresentare le funzioni del file eseguibile, viene fornita, nel Listato 3.6, una visione logica della struttura dei simboli di un ELF, mostrando l'output parziale del comando `readelf -sW`, il quale permette di esaminare il contenuto della sezione *".symtab"*, eseguito su un file ELF di esempio.

Ogni simbolo ha associato un valore e una dimensione. Nel caso di simboli indicanti funzioni, il valore è da interpretare come l'indirizzo a cui la funzione ha inizio all'interno del file eseguibile, specificato come offset dall'inizio del file; la dimensione ne indica la taglia, in byte.

Queste sono informazioni chiave che permettono di individuare le istruzioni che appartengono a ciascuna funzione, confrontando la posizione nel file delle istruzioni disassemblate con l'intervallo di indirizzi occupati dalla funzione stessa.

Ogni funzione presente nell'eseguibile è rappresentata da una struttura dati di tipo `Asm_function` (Listato 3.7), a cui viene associata la lista delle sue istruzioni, ma non solo: viene effettuato anche il collegamento inverso, mantenendo in ogni istruzione `insn_info_x86` un riferimento alla propria `Asm_function`.

```
Symbol table '.symtab' contains 50 entries:
   Num:    Valore          Dim  Tipo    Assoc  Vis      Ind Nome
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS Scrt1.o

     [...]

    27: 0000000000001290    12 FUNC    GLOBAL DEFAULT   15 foo_3p
    28: 00000000000015d0   236 FUNC    GLOBAL DEFAULT   15 foo4
    29: 00000000000012e0   384 FUNC    GLOBAL DEFAULT   15 foo2
    30: 0000000000004020     0 NOTYPE  GLOBAL DEFAULT   25 __data_start
    31: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND strcmp@GLIBC_2.2.5
    32: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    33: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND strtol@GLIBC_2.2.5
    34: 0000000000004028     0 OBJECT  GLOBAL HIDDEN    25 __dso_handle
    35: 00000000000012a0    16 FUNC    GLOBAL DEFAULT   15 print_fun
    36: 0000000000002000     4 OBJECT  GLOBAL DEFAULT   17 _IO_stdin_used
    37: 00000000000012b0    44 FUNC    GLOBAL DEFAULT   15 foo
    38: 0000000000004060     0 NOTYPE  GLOBAL DEFAULT   26 _end
    39: 0000000000004058     8 OBJECT  GLOBAL DEFAULT   26 pointer
    40: 00000000000011a0    34 FUNC    GLOBAL DEFAULT   15 _start
    41: 0000000000001480   220 FUNC    GLOBAL DEFAULT   15 jt_trial
    42: 0000000000004030     0 NOTYPE  GLOBAL DEFAULT   26 __bss_start
    43: 0000000000001080   286 FUNC    GLOBAL DEFAULT   15 main
    44: 0000000000001560   105 FUNC    GLOBAL DEFAULT   15 foo3

     [...]
```

Listato 3.6: Struttura dei simboli di un file ELF

```c
typedef struct Asm_function{
    Symbol *sym;                    // symbol associated with the function
    linked_list *instructions;      // list of disassembled instructions
    char* name;                     // name of the function
    uint64_t size;                  // size in bytes of the function
    uint64_t start_addr;            // initial virtual address of the function
    uint64_t end_addr;              // vaddr of the end of the function
    ll_node *fi;                    // first instruction or NULL
    size_t num_instructions;        // number of instructions of the function
    bool uses_jt;                   // true if contains a jt
}Asm_function;
```

Listato 3.7: Definizione della struttura dati `Asm_function`

È evidente come poter contare su una simile rappresentazione delle funzioni permetta, nel momento in cui si effettua l'aggiunta, la modifica o la rimozione di una o più istruzioni, di tenere traccia dei cambiamenti in termini di size e/o di posizione delle funzioni stesse, consentendone poi una gestione agevole in fase di ricostruzione dell'eseguibile da generare in output.

Oltre ai simboli di tipo `ST_FUNC`, in questa fase vengono gestiti anche simboli di tipo `ST_OBJECT` (fatta eccezione per quelli che hanno visibilità `STV_HIDDEN`), mantenendo un riferimento anche alla specifica `Section_data` cui fanno riferimento. Tali simboli, tipicamente, sono usati per riferire elementi presenti nelle sezioni dati. Il loro valore dovrà essere opportunamente aggiornato in fase di riassemblaggio dell'eseguibile tenendo conto degli spostamenti subiti dalle sezioni dati.

3.2.9 `Elf_program` e `Elf_state`

La procedura di parsing dell'ELF appena descritta, permette di ottenere una **rappresentazione intermedia** in memoria dell'eseguibile tramite le strutture dati presentate, su cui poter successivamente applicare le tecniche di metamorfismo.

Per mantenere un unico punto d'accesso a tale descrizione dell'ELF, sono state definite due ulteriori strutture dati, di tipo `Elf_program` ed `Elf_state` (Listato 3.8). La prima mantiene riferimenti all'ELF header, alla program header table e alla tabella delle sezioni; inoltre, viene utilizzata per mantenere la lista collegata contenente tutte le istruzioni disassemblate del programma. La seconda, invece, mantiene a sua volta un riferimento alla struttura `Elf_program`, ma, in aggiunta, ha dei dei puntatori alle diverse liste di elementi che rappresentano le altre componenti dell'eseguibile:

- *strings*: lista delle stringhe contenute nell'eseguibile;
- *symbols*: elenco dei symboli contenuti nella sezione `.symtab`;
- *functions*: elenco delle funzioni che compaiono nell'ELF;

- *j_tables*: lista completa delle *jump tables* di cui è stata rilevata la presenza all'interno del codice del file eseguibile.

```
typedef struct Elf_program{
  Elf *elf;                          // ELF handle
  GElf_Ehdr *ehdr;                   // Executable header
  Phdr_table *phdrt;                 // Program Header Table
  Sec_table *sect;                   // Section Header Table
  linked_list *unified_instructions; // Disassembled instructions of the ELF
}Elf_program;

typedef struct Elf_state{
  Elf_program *prog;                 // Elf program descriptor
  linked_list *symbols;              // A list of all symbols
  linked_list *functions;            // A list of all functions in the elf
  linked_list *strings;              // A list of all strings in the elf
  linked_list *j_tables;             // A list of all jump tables in the elf
}Elf_state;
```

Listato 3.8: Definizione delle strutture dati `Elf_program` ed `Elf_state`

L'ultima lista, relativa alle tabelle dei salti, è stata introdotta in iterazioni di sviluppo successive.

Dal momento che si assume che l'engine metamorfico, ad ogni esecuzione, debba agire su un unico file ELF (quello di *MorphVM*, in cui esso stesso è contenuto), viene dichiarata una variabile statica globale `state` di tipo `Elf_state`, che funge da unico *access point* alla rappresentazione intermedia dell'eseguibile, rendendo sempre facilmente accessibile ogni metadato necessario.

3.3 Il disassembler

In questa sezione viene descritto in maniera dettagliata il procedimento di disassemblaggio delle istruzioni x86, illustrando i metadati che il disassembler fornisce per ogni struttura `insn_info_x86` e le informazioni che vengono ricavate dalle procedure aggiuntive eseguite in questa fase e a cui è già stato fatto riferimento nelle Sezioni 2.4.2 e 2.4.3.

Per interpretare i byte di codice del programma, è stato riadattato alle esigenze del progetto di *MorphVM* un **disassembler** di istruzioni x86 preesistente e apertamente disponibile, in quanto sviluppato nell'ambito di un

progetto open-source [25, 26]. Esso offre un'interfaccia per l'utilizzo tramite l'API `x86_disassemble_instructions`, alla quale fornire un array di byte che vengono interpretati come una sequenza di istruzioni x86. Viene restituita in output una lista collegata i cui nodi sono di tipo `insn_info_x86` e rappresentano le singole istruzioni.

3.3.1 La struttura `insn_info_x86`

La struttura `insn_info_x86`, introdotta precedentemente nel Listato 3.4, è la struttura dati chiave attorno a cui ruota tutto il funzionamento dell'engine metamorfico.

Il disassembler fornisce numerosissime informazioni utili per descrivere al meglio l'istruzione rappresentata, popolando opportunamente i campi della struttura. Tra essi, quelli di maggior rilevanza sono: i byte che compongono l'istruzione (campo `insn`), l'opcode, lo mnemonico associato all'istruzione (una stringa, e.g. "mov", "jmp", "call"); molteplici membri della struttura forniscono informazioni sugli operandi utilizzati, come ad esempio il valore dei byte `modrm` e `sib`, nonché il valore di eventuali immediati (`immed`) o informazioni sui registri e i parametri di indirizzamento usati (`scale, ireg, disp`). Molto importanti sono il campo `uses_rip`, il quale è settato a *true* quando l'istruzione utilizza indirizzamento *RIP-relative*, e i campi `op` e `opd_size` che permettono, in combinazione coi prefissi, di derivare la size degli operandi utilizzati, in particolar modo dei registri (ad esempio, `ax` piuttosto che `eax` o `rax`).

Una menzione particolare merita il campo `flags`, che permette di inferire informazioni addizionali sul comportamento o sulla natura delle istruzioni, contribuendo a classificarle in macro-categorie e, di conseguenza, ad elevare il livello di astrazione con cui le istruzioni vengono processate. Viene fornita una descrizione completa dei possibili valori dei flag, combinabili tra loro, nel Listato 3.9.

Oltre alle informazioni fornite dal disassembler, la struttura `insn_info_x86` è stata arricchita di ulteriori campi che hanno lo scopo di collocare logica-

```
#define I_MEMRD       0x1     // Legge dalla memoria
#define I_MEMWR       0x2     // Scrive sulla memoria
#define I_CTRL        0x4     // Istruzioni della famiglia "test" e "control"
#define I_JUMP        0x8     // Istruzioni della famiglia "jump"
#define I_CALL        0x10    // Istruzione del tipo "call"
#define I_RET         0x20    // Istruzione del tipo "ret"
#define I_CONDITIONAL 0x40    // Istruzione eseguita solo se condizione
#define I_STRING      0x80    // Opera su stringhe (e.g. movs, stos, cmps...)
#define I_ALU         0x100   // Esegue operazioni di tipo logico-aritmetico
#define I_FPU         0x200   // Utilizza la Floating Point Unit (FPU)
#define I_MMX         0x400   // Istruzione che utilizza i registri MMX
#define I_XMM         0x800   // Istruzione che utilizza i registri XMM
#define I_SSE         0x1000  // Istruzione SSE
#define I_SSE2        0x2000  // Istruzione SSE2
#define I_PUSHPOP     0x4000  // Istruzione di tipo "push" o di tipo "pop"
#define I_STACK       0x8000  // Se l'istruzione opera nello stack
#define I_JUMPIND     0x10000 // Indirect Branch
#define I_CALLIND     0x20000 // Indirect Call
#define I_MEMIND      0x40000 // Indirect memory address load (LEA)
```

Listato 3.9: Possibili flag per un'istruzione `insn_info_x86`

mente l'istruzione all'interno del file ELF, mettendola in relazione con le altre componenti della rappresentazione intermedia dell'eseguibile e agevolare così l'applicazione delle strategie di metamorfismo. Nel dettaglio, i campi aggiunti sono:

- *orig_addr*: l'offset orginale in cui l'istruzione è posizionata all'interno del file ELF di input, prima di avviare il processo di metamorfosi;
- *new_address*: l'offset a cui l'istruzione dovrà essere riscritta nell'ELF di output;
- *displacement*: lo shifting subito dall'istruzione a causa delle modifiche apportate all'ELF; è inteso come la differenza tra *new_address* e *orig_addr*;
- *jump_target*: se l'istruzione appartiene alla famiglia delle istruzioni di salto, tale campo mantiene un riferimento all'istruzione target del salto;
- *target_of*: se l'istruzione è target di qualche istruzione di salto, il campo è una lista collegata contenente riferimenti a tutte le istruzioni di salto che la hanno come target;
- *jump_op_size*: numero di byte utilizzati per specificare il target del salto;

- *jump_op_start*: numero di byte dopo il quale, all'interno dei byte dell'istruzione, inizia ad essere specificato il target del salto;
- *sec_data*: riferimento alla `Section_data` a cui l'istruzione appartiene;
- *function*: riferimento alla funzione di cui l'istruzione fa parte;
- *uses_jt*: assume valore *true* se la funzione a cui l'istruzione appartiene fa utilizzo di una *jump table*;
- *jt*: riferimento alla *jump table*; altrimenti, `NULL`.

Va sottolineata l'importanza ricoperta dal campo `target_of`: esso, nel caso in cui la lista sia non vuota, funge di fatto da **label mark**, segnalando che l'istruzione è target di qualche istruzione di salto.

3.3.2 Processo di disassemblaggio

Per poter disassemblare il codice del programma, è necessario discriminare quali siano le cosiddette "sezioni testo" che lo contengono. La loro individuazione è effettuata considerando la tipologia della sezione e i flag ad essa associati, specificati nel corrispondente section header; in particolare, una sezione è considerata da disassemblare se valgono entrambe le seguenti due condizioni:

- il campo `sh_type` del section header ha valore `SHT_PROGBITS`;
- il bit corrispondente al flag `SHF_EXECINSTR` è settato ad 1 nel campo `sh_flags` del section header.

Tipicamente, in eseguibili compilati in maniera convenzionale, la maggior parte del codice è contenuta in una sezione denominata "*.text*". Individuate le sezioni di interesse, grazie alla rappresentazione intermedia dell'eseguibile, è possibile iterare sui blocchi dati che ne rappresentano il contenuto, accedendo alle strutture `Elf_Data` associate alla sezione. Si disassemblano i byte di ciscun blocco, costruendo una lista di istruzioni di cui viene salvato un riferimento nella corrispondente struttura dati `Section_data`.

Oltre al puro disassemblaggio delle istruzioni, vengono eseguite operazioni di analisi aggiuntive: per istruzioni che fanno uso di indirizzamento *RIP-relative*, ma non appartengono alla classe delle JUMP (guardando il valore dei flag), nella struttura `insn_info_x86` si mantengono informazioni sull'offset specificato dall'istruzione rispetto al valore dell'`instruction pointer` per indirizzare la memoria. Ciò risulta essere molto utile nella fase di riassemblaggio dell'eseguibile, successivamente all'applicazione delle modifiche. Ad esempio, si supponga che siano state introdotte nuove istruzioni di *garbage* nel codice, che corrispondono a byte posizionati presumibilmente in una posizione intermedia rispetto all'intero insieme di istruzioni di partenza. Conoscere l'offset dell'indirizzamento *RIP-relative* utilizzato originariamente da ogni istruzione è fondamentale per poterlo riaggiustare opportunamente nel caso in cui l'inserimento di nuovi byte nel corpo del codice lo renda necessario.

Il motivo per cui tale procedura non è applicata alle istruzioni di classe JUMP è che per esse viene già effettuata in automatico una gestione di questo tipo dal disassemblatore. Di fatto, ciò che avviene in questa fase è semplicemente una procedura di *flagging* delle istruzioni, che porterà, come descritto in Sezione 3.3.3, a popolare dei metadati che tracciano i riferimenti tra le istruzioni e tra istruzioni e porzioni dell'eseguibile che non contengono codice.

3.3.3 Riferimenti tra istruzioni

Dopo aver popolato tutte le strutture dati precedentemente descritte, si procede all'individuazione di collegamenti tra istruzioni: per ogni istruzione che sia una `JMP`, una `CALL` o una istruzione di salto condizionale, si individua l'istruzione target del salto e ne viene salvato un riferimento nel campo `jump_target` della struttura `insn_info_x86` relativa alla istruzione di salto.

Viceversa, ogni istruzione che è target di un salto, può essere interpretata come se avesse una label associata, a cui molteplici istruzioni possono saltare. Di conseguenza, viene mantenuto anche il riferimento inverso, ma in una lista (campo `target_of`): di fatto, ogni istruzione che è target di salti, mantiene anch'essa dei riferimenti a tutte le istruzioni di salto che la bersagliano. Come

già anticipato al termine della Sezione 3.3.1, i riferimenti mantenuti nella lista `target_of` fungono anche da *label mark* per l'istruzione bersaglio dei salti. Un esempio dei collegamenti tra le istruzioni appena descritti è raffigurato in Figura 3.5.

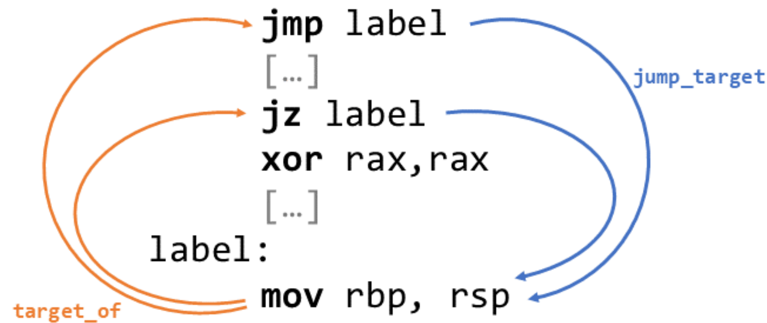

Figura 3.5: Tracciamento dei riferimenti tra istruzioni

In aggiunta alle istruzioni di salto, si creano collegamenti anche per tutte le istruzioni precedentemente flaggate, che fanno uso di indirizzamento *RIP-relative*. Se l'indirizzo cui fanno riferimento non corrisponde a sezioni contenenti codice disassemblato, molto probabilmente si tratta di un indirizzamento ai dati del programma (e.g. all'interno della sezione `.data`). In tal caso, si procede ad individuare la `Section_data` corrispondente al dato target.

Arricchire la rappresentazione intermedia del file eseguibile mantenendo tutti questi riferimenti permette di riaggiustare opportunamente i displacement sia delle istruzioni di salto che degli indirizzamenti *RIP-relative* in fase di ricostruzione dell'ELF, in accordo alle modifiche e ai cambiamenti di disposizione delle istruzioni.

Il componente dell'engine metamorfico responsabile del parsing del file eseguibile e del processo di disassemblaggio e interpretazione del codice è, a questo punto dello sviluppo, sufficiente per poter raggiungere l'obiettivo fissato nel primo passo di sviluppo. In realtà, esso è stato già sviluppato in maniera tale da tener conto di tutte le informazioni aggiuntive necessarie nel momento in cui l'ELF dovrà essere modificato. Per poter testare il corretto funziona-

mento di quanto sviluppato, è necessario realizzare la parte finale dell'assembler dell'engine, ovvero quella che si occupa di riscrivere il file a partire dalla rappresentazione intermedia.

3.4 Riscrittura di un ELF

Sebbene in questa fase non fosse strettamente necessario tenere conto di eventuali spostamenti di sezioni ed header dell'ELF, poiché, per il momento, non si apporta alcuna modifica al contenuto del file, il componente che realizza la riscrittura dell'eseguibile è stato comunque sviluppato in previsione di tale scenario.

Tuttavia, non è stato possibile realizzare la procedura di overwriting del file affidandosi completamente alla libreria *libelf*, al contrario di quanto affermato dalla documentazione della libreria stessa: infatti, lasciandola gestire direttamente a *libelf*, si ottiene in output un file che risulta essere un *dynamically loadable object*, simile di fatto ad una libreria condivisa, non più un *executable*.

Per poter ottenere in output un formato ELF effettivamente eseguibile, è stato necessario ridurre il controllo che la libreria *libelf* aveva sulla riscrittura del file, settando il flag `ELF_F_LAYOUT` sul descrittore dell'ELF, tramite una chiamata all'API `elf_flagelf`. In questo modo, come ben descritto anche dalla documentazione online [27], la gestione di diversi membri delle strutture dati rappresentanti l'ELF non è più demandata alla libreria, ma diviene responsabilità dell'applicazione. In particolare, ciò che non è più gestito automaticamente comprende:

- I campi `e_phoff` ed `e_shoff` dell'executable header, i quali determinano il posizionamento nel file della program header table e della section header table.
- Per ogni sezione, i membri `sh_addralign,` `sh_offset` e `sh_size` dei section header.

- Il campo `d_off` dei descrittori dati delle sezioni, il quale specifica l'offset del buffer di dati all'interno della sezione di riferimento.
- L'intera gestione delle entry della program header table è demandata all'applicazione.

Di conseguenza, è stato adottato un approccio più complesso, ma che permettesse di avere un controllo completo sulla riscrittura del file ELF: **riscrivere l'eseguibile "a mano"**. Nel farlo, sono stati presi in considerazione diversi aspetti: in primis, in base al campo `e_ident[EI_CLASS]` dell'executable header, si comprende se l'ELF sia a 32 bit o a 64 bit. Per ognuno dei due casi, sono state sviluppate delle API apposite che si occupano della riscrittura delle diverse componenti del file ELF, in particolare degli header. A seconda dello scenario in cui ci si trova, si fa uso delle funzioni appropriate, che organizzano i campi rispettando il formato dei metadati adeguato. Inoltre, si tiene conto anche dell'endianess del file, dedotta dal campo `e_ident[EI_DATA]` dell'ELF header.

Il file viene riscritto seguendo il layout canonico di un ELF, che prevede di iniziare dall'executable header, seguito dalla program header table, dalle sezioni e, infine, dalla section header table. Durante questo processo, ci si accerta che l'allineamento richiesto dai vari membri del file sia rispettato, inserendo, ove necessario, dei byte di **padding** che, nel caso di sezioni contenenti codice hanno valore `0x90` (opcode dell'istruzione `NOP`), altrimenti hanno valore `0x00`.

Il vantaggio apportato dall'adozione di questa soluzione *handcrafted* è che si ha un controllo completo sul risultato del processo metamorfico ed è possibile mappare direttamente sul file di output le modifiche effettuate sulla rappresentazione intermedia dell'ELF che è stata realizzata, garantendo un guadagno in termini di flessibilità.

In questo modo, si è riusciti ad ottenere un file di output identico a quello di input e che risultasse ancora correttamente eseguibile.

3.5 Aggiunta di una NOP

Nella fase di espansione dell'engine metamorfico, come descritto nella Sezione 2.5, una delle operazioni effettuate è l'inserimento di istruzioni di *garbage*, ovvero istruzioni spurie che non hanno impatto semantico sul codice, ma contribuiscono a variarne la struttura e, in particolare, la posizione delle altre istruzioni, aumentando il "rumore" all'interno dell'eseguibile. La più semplice è una istruzione di `NOP` (*no operation*), costituita da un unico byte di opcode pari a `90`, in esadecimale.

L'aggiunta dell'istruzione viene realizzata tramite l'inserimento di una corrispondente struttura `insn_info_x86` sia nella lista collegata di istruzioni della `Section_data` in cui si intende effettuare l'inserimento, sia nella lista di istruzioni della `Asm_function` target. In questo modo, l'istruzione risulterà essere inserita nella rappresentazione intermedia dell'ELF mantenuta in memoria durante il processo di metamorfosi e verrà poi effettivamente scritta sull'eseguibile generato come risultato dell'intero processo.

Chiaramente, il semplice inserimento di un nodo nelle liste collegate non è sufficiente a garantire la correttezza dell'output, ma è necessario **tenere traccia anche degli effetti** che tale operazione comporta nei confronti sia delle altre istruzioni che dei metadati dell'ELF. Infatti, bisogna tener conto di:

- spiazzamenti delle istruzioni *RIP-relative* e dei salti;
- incremento della taglia delle funzioni; il processo, a cascata, può comportare anche lo shifting in avanti dell'indirizzo d'inizio delle funzioni posizionate dopo quella in cui l'istruzione è stata aggiunta;
- cambiamento della size delle sezioni e, eventualmente, necessità di specificare taglie e indirizzi virtuali differenti anche per i segmenti; ciò può comportare, anche in funzione del padding necessario per garantire l'allineamento richiesto dai vari componenti dell'ELF, uno spostamento in avanti della posizione della section header table all'interno del file;

- necessità di sostituire opportunamente i valori dei simboli, per garantire la correttezza dell'eseguibile;
- eventuale spostamento dell'*entry point* del programma.

3.6 Jump table detection

Prima di analizzare nel dettaglio le operazioni svolte dall'assembler dell'engine per ricostruire correttamente l'eseguibile modificato, bisogna fare un passo indietro e trattare il topic della detection delle tabelle dei salti.

Come già introdotto nella Sezione 2.4.5, è necessario tener conto anche dell'impatto che le modifiche applicate al contenuto del file hanno sulle eventuali jump table utilizzate in alcune funzioni. La loro individuazione è basata sull'utilizzo di **euristiche** realizzate prendendo come riferimento l'output del compilatore GCC su file di esempio, compilando sia con direttive di ottimizzazione (`-O2` e `-O3`) che senza. Gli output nei due casi sono leggermente differenti e un'euristica per ognuno dei due scenari è adottata. In particolare, nella fase di creazione della rappresentazione dell'ELF, dopo aver definito le funzioni e averle associate con le istruzioni, ciascuna funzione viene sottoposta al processo di detection di branch table eseguendo l'euristica di base. Se non viene individuata alcuna tabella dei salti, viene eseguita anche l'euristica per la detection in caso di compilazione ottimizzata.

3.6.1 Strutture dati di supporto

Sono state definite delle strutture dati di supporto alle euristiche che hanno lo scopo di mantenere lo stato della detection e permettere di individuare i pattern e le informazioni per localizzare poi la tabella in memoria. Esse sono presentate nel Listato 3.10.

```c
#define N_REG 16
typedef enum Register_values{
  UND=0,                // undefined
  OFF=1,                // offset in the jump table
  BASE=2,               // base address of the jump table
  JMP_DST=3             // destination address of the indirect jump
}Register_value;

typedef struct Register{
  unsigned char ndx;     // register number
  Register_value value;  // register role
}Register;

struct Jump_table_lookup_state{
  unsigned long base_off;
  unsigned long base_addr;
  insn_info_x86* indirect_jump;
  Register registers[N_REG];
}lookup_state;
```

Listato 3.10: Strutture di supporto alla jump table detection

3.6.2 Euristiche di detection

Per poter comprendere il pattern attraverso il quale si rileva la presenza di una branch table, viene riportato nel Listato 3.11 il codice x86 di una funzione che fa utilizzo di una jump table, il cui sorgente è stato compilato senza direttive di ottimizzazione.

```
[...]
12f4: mov    rax, QWORD PTR [rbp-0x8]
12f8: lea    rdx,[rax*4+0x0]
1300: lea    rax,[rip+0xe91]      # 2198 <_IO_stdin_used+0x198>
1307: mov    eax, DWORD PTR [rdx+rax*1]
130a: cdqe
130c: lea    rdx,[rip+0xe85]      # 2198 <_IO_stdin_used+0x198>
1313: add    rax,rdx
1316: jmp    rax
1318: lea    rax,[rip+0xd98]      # 20b7 <_IO_stdin_used+0xb7>
131f: mov    rdi,rax
1322: call   1030 <puts@plt>
1327: jmp    13eb <jt_trial+0x10e>
132c: lea    rax,[rip+0xd93]      # 20c6 <_IO_stdin_used+0xc6>
1333: mov    rdi,rax
1336: call   1030 <puts@plt>
133b: jmp    13eb <jt_trial+0x10e>
[...]
```

Listato 3.11: Codice assembly x86 che mostra l'utilizzo di una jump table

Questo snippet di codice rappresenta una porzione di uno *switch/case statement* che utilizza un valore numerico per discriminare i vari casi e mostra il pattern che l'euristica tenta di individuare. La prima istruzione di `MOV`, carica nel registro `rax` il valore che determina il case da seguire nel flusso d'esecuzione. Tale valore viene utilizzato come **indice per l'accesso alla jump table**. Infatti, nella successiva `LEA` viene salvato in `rdx` il valore di `rax` moltiplicato per quattro: ogni entry della tabella occupa 4 byte e rappresenta un offset da utilizzare in un'istruzione di salto indiretto per seguire il flusso di esecuzione opportuno.

La successiva `LEA` utilizza indirizzamento *RIP-relative* per caricare in `rax` l'**indirizzo base della jump table**, di solito contenuto della sezione `.rodata`. Dopodiché, si legge il contenuto dell'entry della tabella corrispondente all'offset calcolato in precedenza e si salva nella parte bassa del registro `rax`, estendendolo in segno (istruzioni `MOV` e `CDQE`). La successiva istruzione `LEA`, riscrive nuovamente in `rdx` l'indirizzo della branch table.

A questo punto, in `rdx` si ha l'indirizzo base della tabella, mentre in `rax` è contenuto un offset da sommare a tale base per ottenere l'indirizzo a cui saltare. Di fatti, si procede con l'operazione di `ADD rax,rdx` e infine con il salto indiretto `JMP rax`.

Le successive istruzioni visibili nel Listato 3.11 rappresentano i blocchi case dello *switch statement* e ognuno di essi non fa altro che stampare una stringa su standard output ed effettuare il `break`.

L'euristica tenta di individuare la presenza del pattern appena descritto, analizzando le istruzioni della singola funzione in maniera sequenziale. Le prime due istruzioni di `MOV` e `LEA` non vengono considerate per motivi di efficienza, ma si parte direttamente con l'individuazione di una possibile `LEA` che scriva in un registro l'indirizzo base della tabella dei salti. La procedura è descritta in pseudo-codice dall'Algoritmo 1.

Come si evince dall'algoritmo, una volta che viene rilevata la presenza di una branch table, bisogna rappresentarla tramite una struttura dati e salvarne un riferimento. Per poterlo fare, manca ancora un passaggio: **individuare**

Algoritmo 1: Basic JT Heuristic

```
Data: function
found ← 0;
forall ins in function do
    if (ins is LEA and ins addresses ".rodata") then
        State.registers[ins.dest_reg] ← BASE;
        mov ← findMov(ins.next, ins.dest_reg);
        if (mov ≠ NULL) then
            State.registers[mov.base_reg] ← OFF;
            lea ← findLeaWithSameAddr(mov.next);
            if (lea ≠ NULL) then
                add = findAddBasePlusOff(lea.next);
                if (add ≠ NULL) then
                    State.registers[add.dest_reg] ← JMP_DST;
                    jmp ← findIndirectJmp(add.next);
                    if (jmp ≠ NULL) then
                        found ← 1;
                    end
                end
            end
        end
    end
end
if found then
    jt ← buildJumpTable();
    return jt;
else
    return NULL;
end
```

quante sono le entry che fanno parte della tabella stessa.

Se si riesce ad ottenere tale valore, si può andare a leggere il contenuto delle diverse entry della tabella nella sezione *.rodata*, ricostruire gli indirizzi target e, di conseguenza, le istruzioni a cui l'istruzione di salto indiretto può saltare. In questo modo, è possibile tracciare, analogamente a quanto fatto per tutte le istruzioni di salto, i riferimenti tra le istruzioni (sempre tramite i campi `jump_target` e `target_of` delle strutture `insn_info_x86`), così da ricostruire correttamente la jump table nella fase di riassemblaggio dell'eseguibile. La tecnica con cui si effettua l'individuazione del numero di entry presenti nella tabella è presentata nella Sezione 3.6.3.

Per quanto riguarda l'euristica utilizzata nello scenario di codice compilato con direttive di ottimizzazione, si segue lo stesso procedimento logico dell'euristica precedente. Tuttavia, il pattern da individuare è leggermente differente. Come si può vedere dal Listato 3.12, si parte direttamente caricando in un registro la base della branch table, sempre tramite una `LEA` che indirizza dati nella sezione *.rodata*. Stavolta, il contenuto dell'entry della tabella adeguata viene salvato in un registro (`rax` nell'esempio) utilizzando una singola istruzione `MOVSXD`, che ha lo stesso comportamento di una `MOV`, ma effettua già l'estensione in segno. L'entry della tabella viene indirizzata con il modo canonico che fa uso di *base, scale* e *index*, dove la base è il registro contenente la base della branch table, la scala è pari a 4 e l'indice è contenuto nel registro che ospita il parametro dello *switch statement*:

```
MOVSXD Reg, [ jt_base + parameter·4 ]
```

```
1480: cmp    rdi,0x12
1484: ja     14a0 <jt_trial+0x20>
1486: lea    rdx,[rip+0xd03]    # 2190 <_IO_stdin_used+0x190>
148d: movsxd rax,DWORD PTR [rdx+rdi*4]
1491: add    rax,rdx
1494: jmp    rax
```

Listato 3.12: Codice assembly x86 che mostra il pattern per individuare una jump table nel caso di compilazione con ottimizzazione

Dopodiché, le altre istruzioni del pattern sono analoghe al caso base: viene effettuata una `ADD` dell'indirizzo base della tabella con l'offset ottenuto tramite la `MOVSXD` e si effettua un salto indiretto. Dunque, la seconda euristica tenta di individuare, in maniera del tutto similare a quella precedentemente presentata, il pattern formato dalla sequenza ordinata di istruzioni `LEA, MOVSXD, ADD, JMP`, non necessariamente consecutive, assegnando un ruolo ai registri che esse utilizzano, così da verificare una corrispondenza semantica con quanto ci si aspetta sia fatto per utilizzare una branch table.

3.6.3 Numero di entry di una branch table

Per individuare il numero di elementi che compongono una jump table in memoria, si sarebbe potuto fare affidamento su eventuali istruzioni di `CMP` o simili che, di solito, si trovano prima del caricamento dell'indirizzo base della tabella in un registro. Spesso, infatti, viene effettuato un confronto del valore del parametro dello *switch statement* con il valore più elevato tra quelli associati ai case dello switch stesso, il quale appare come immediato della *CMP*. In riferimento al listato 3.12, tale valore è `0x12`, ovvero `18` in decimale. Tuttavia, questo approccio può portare ad errori, in quanto non è detto che i valori associati ai case seguano necessariamente un pattern numerico. Inoltre, può essere presente un caso di default aggiuntivo non catturato dall'immediato.

L'approccio adottato in *MorphVM* è ancora una volta basato su un'**euristica**, che si fonda sull'idea che, verosimilmente, gli indirizzi ottenibili sommando l'indirizzo base della jump table e i valori delle entry ricadano tutti all'interno della funzione che si sta analizzando (di fatto, devono indirizzare i case dello *switch statement*).

Da ciò deriva che i valori contenuti nelle entry della tabella non possano, in valore assoluto, differire di molto, in maniera particolare più della size della funzione stessa. Ciò è visibile nella Figura 3.6, che mostra i byte contenuti nella sezione *.rodata* corrispondenti alla tabella dei salti utilizzata dal codice del Listato 3.11. Notare come le entry ad indice dispari, presentino tutte l'offset `0xfffff225`. Questo offset, sommato all'indirizzo base della tabella,

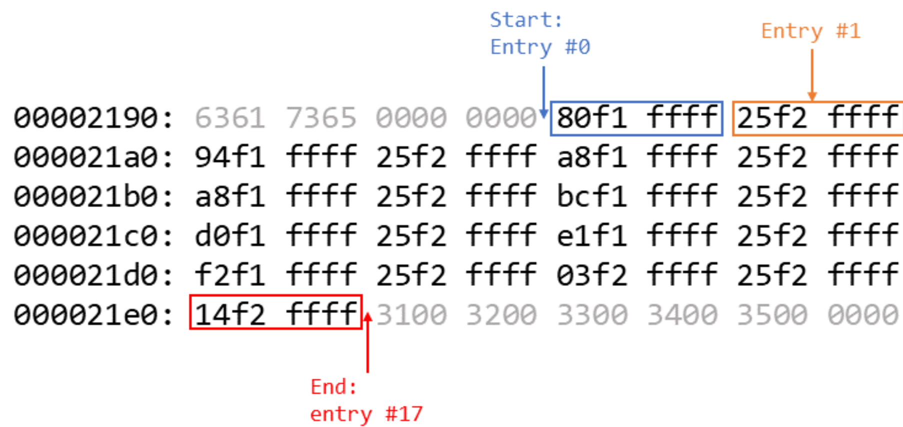

Figura 3.6: Byte che rappresentano una branch table nella sezione dati

permette di raggiungere il caso di default dello *switch statement*: infatti, i valori associati ai diversi case dello switch sono i numeri pari da 0 a 18 e, quindi, i dispari sono ridirezionati tutti sul default case. Il codice sorgente in linguaggio ad alto livello era stato già presentato nel Listato 2.3.

Dunque, per **individuare il numero di entry**, si parte dalle informazioni recuperate durante la detection della branch table, analizzando il contenuto della sezione *.rodata* a partire dall'indirizzo base della tabella. Si assume che l'entry ad indice 0 appartenga sicuramente alla tabella e si calcola il valore assoluto della differenza tra tale entry e la successiva, verificando che sia minore di una determinata soglia. La threshold viene inizialmente impostata al valore della dimensione della funzione diviso 4. Si va avanti fin quando il valore assoluto della differenza tra due entry consecutive supera la soglia.

Se il numero di elementi individuati è almeno pari a 7 (in pratica, è il minimo numero di entry necessarie per portare il compilatore a considerare di utilizzare una branch table), la ricerca è ritenuta conclusa; altrimenti si ripete il procedimento descritto da capo, raddoppiando il valore della soglia, per un massimo di 2 iterazioni. La threshold è comunque limitata superiormente dalla dimensione della funzione.

Avendo a disposizione il numero di elementi della tabella, si può procedere ad effettuare il collegamento tra l'istruzione di salto indiretto e tutte le istruzioni corrispondenti ai vari casi, calcolando i target dei salti tramite la base

della branch table e gli offset presenti in ciascuna entry.

Infine, per ogni jump table, viene popolata una struttura che la rappresenta, la cui definizione è riportata nel Listato 3.13.

```
typedef struct Jump_table {
  Asm_function *fun;
  unsigned long base_addr;
  unsigned long base_off;
  insn_info_x86 *rel_jmp;
  size_t n_entries;
  int32_t *first_entry;
  linked_list *targets;
} Jump_table;
```

Listato 3.13: Definizione della struttura `Jump_table`

3.7 L'assembler

La rappresentazione intermedia dell'eseguibile permette di tenere traccia delle mutazioni applicate e degli effetti che esse causano ai diversi componenti dell'ELF. L'**assembler** dell'engine metamorfico ha il compito di ricostruire l'eseguibile di output, andando a correggere le dimensioni e le posizioni di sezioni, segmenti, header. Inoltre, deve occuparsi di sistemare tutte le istruzioni di salto e salto condizionato, nonché quelle relative alle branch table. Vanno corretti anche i valori dei simboli e delle funzioni, per garantire il corretto funzionamento dell'eseguibile finale.

3.7.1 Effetti delle modifiche su funzioni ed *entry point*

Nel momento in cui un'istruzione viene aggiunta, rimossa o modificata all'interno di una funzione, viene utilizzata la struttura `Asm_function` per tenere traccia degli effetti che tale operazione ha sulla funzione stessa. In particolare, nel caso di un'aggiunta, la dimensione della funzione viene incrementata del numero di byte di cui l'istruzione aggiunta è composta; in modo analogo, si aggiorna l'indirizzo che determina la fine della funzione. Per quanto riguarda l'*entry point* dell'eseguibile, si considera se l'istruzione aggiunta è stata inserita in un punto del file che precede l'indirizzo dell'*entry point*. In tal caso,

anche a quest'ultimo viene applicato uno spostamento in avanti del numero di byte pari alla taglia dell'istruzione. Aggiornare la posizione del punto di ingresso del programma è strettamente necessario per la correttezza dell'output, dal momento che tale valore va specificato nel campo `e_entry` dell'executable header del file ELF. Uno snippet di codice che mostra brevemente le procedure descritte è visualizzabile nel Listato 3.14.

```
1  insn_info_x86 *ins;
2  Asm_function *fun;
3
4  /* ... */
5
6  // increase the size of the function and its end address
7  fun->size+=ins->insn_size;
8  fun->end_addr+=ins->insn_size;
9  fun->num_instructions++;
10
11 /* ... */
12
13 // update the entry point position, if necessary
14 if (PRECEDES_ENTRY_POINT(ins->orig_addr)){
15     UPDATE_ENTRY_POINT(ins->insn_size);
16 }
```

Listato 3.14: Tracciamento degli effetti che l'aggiunta di un'istruzione può avere su funzioni ed *entry point*

Questa tipologia di meccanismo in cui si tiene traccia dei displacement tra la posizione iniziale di un elemento nel file e quella finale, viene utilizzato anche per le singole istruzioni e per le sezioni, agevolando lo sviluppo e l'implementazione delle funzionalità di responsabilità dell'assembler dell'engine.

3.7.2 Algoritmo di reassembling

L'assembler è strutturato in maniera tale da seguire una procedura ben precisa prima di poter riscrivere il file di output. I diversi step che compie sono presentati in pseudo-codice nell'Algoritmo 2.

Il primo passo da compiere è quello di effettuare le computazioni riguardanti i metadati dei vari componenti dell'ELF: vengono calcolate le nuove dimensioni delle sezioni e, di conseguenza, le loro nuove posizioni; eventuali spostamenti delle sezioni possono provocare uno shifting in avanti della section header table

Algoritmo 2: Reassemble ELF

```
while no instruction must be modified do
    computeSectionSize();
    shiftSectionHeaderTable();
    adjustEntryPoint();
    computeInstructionAddressAndFixJmp();
end
fixPhdr();
fixJumpTables();
rewriteSections();
rewriteSymbols();
updateELF();
```

e, quindi, è necessario ricalcolare la sua posizione di inizio. Lo stesso può accadere per l'entry point del programma e, dunque, si provvede ad assegnarne il nuovo indirizzo nell'apposito campo dell'executable header mantenuto in memoria (`e_entry`).

Note le posizioni delle sezioni, si procede a calcolare i displacement subiti dalle istruzioni e, di conseguenza, le loro nuove posizioni nel layout del file. Inoltre, si aggiustano le istruzioni di salto, salto condizionato e quelle che riferiscono dati in sezioni diverse da quelle contenenti codice. Tuttavia, come preannunciato in Sezione 2.4.3, potrebbe essere necessario dover **trasformare degli *short jump* in *long jump***. Questa operazione richiede di andare ad operare direttamente sui byte dell'istruzione di salto, in quanto persino l'opcode dell'istruzione differisce nei due scenari. Il problema che ciò comporta è che trasformare uno *short jump* in un *long jump* causa un aumento della dimensione dell'istruzione, che si ripercuote sulle posizioni delle altre istruzioni e sulle dimensioni di funzioni e sezioni. Ciò implica la necessità di rieseguire i passi precedenti per ricalcolare i nuovi metadati aggiornati fin quando tutte le istruzioni non siano state correttamente trasformate. È questo il motivo per cui figura un *while loop* all'inizio dell'Algoritmo 2.

I passaggi principali della procedura presentata sono descritti con maggior dettaglio nelle sezioni successive, in cui vengono evidenziate le principali problematiche che è stato necessario affrontare.

3.7.3 Fixing delle istruzioni di salto e short jump

Le istruzioni di salto e salto condizionato vengono corrette modificando il valore del **displacement** dell'istruzione x86. Tale operazione è resa possibile grazie al tracciamento dei riferimenti tra le istruzioni, descritto nelle Sezioni 2.4.3 e 3.3.3. Infatti, grazie a tale procedura svolta dal disassembler dell'engine, è possibile processare ogni istruzione di salto essendo a conoscenza dell'istruzione target (tramite il campo `jump_target` nella struttura `insn_info_x86`). In questo modo, avendo già calcolato gli indirizzi aggiornati che le istruzioni avranno nel file di output, è possibile calcolare il nuovo spiazzamento che la `JMP` deve specificare per far sì che il salto rimanga logicamente consistente. Infatti, con indirizzamento di tipo *RIP-relative*, tale offset è da calcolarsi come il numero di byte che intercorrono tra l'inizio dell'istruzione target e l'inizio dell'istruzione successiva a quella di salto:

$$\text{displacement} = \text{newAddressOfTarget} - (\text{newAddressOfJmp} + \text{sizeOfJmp})$$

In maniera analoga, viene calcolato anche lo spiazzamento per le istruzioni che indirizzano memoria in sezioni dati. Esso viene utilizzato per sostituire quello originariamente presente nell'istruzione, modificandone direttamente i byte corrispondenti nel campo `insn` della struttura `insn_info_x86`.

Come anticipato, va posta particolare attenzione agli **short jump**, ovvero quelle istruzioni di salto che fanno uso di 1 singolo byte per indicare il displacement. Essendo il byte da interpretare con segno, l'offset specificabile può unicamente essere nel range $[-128; 127]$. Si supponga che l'offset originariamente specificato per la `JMP` fosse pari a 124 e che l'aggiunta di altre istruzioni in posizione intermedia tra quella di salto e quella target abbia portato la distanza in byte tra jump e target a crescere fino al valore di 130 byte. Chiaramente, tale valore non rientra più nel range e, quindi, risulta essere necessario modificare l'istruzione di salto, sostituendola con una codifica che permetta più byte di displacement.

In questo caso, l'engine differenzia la gestione di istruzioni di salto con-

dizionato e salto incondizionato. Per quanto riguarda quest'ultima tipologia, l'unico opcode con cui si può usare un singolo byte di displacement relativo è `0xEB`. Ogni salto di questo tipo viene trasformato in un'istruzione `JMP` con opcode `0xE9`, che permette, invece, di utilizzare tutti e 4 i possibili byte di spiazzamento. Un esempio, è riportato nel Listato 3.15.

```
; before - displacement is 124
eb 7c                   jmp    0x7e

; after - displacement is 130
e9 82 00 00 00          jmp    0x87
```

Listato 3.15: Esempio di trasformazione da short jump a long jump

Per i salti condizionati, la conversione è leggermente più complessa. Infatti, nella loro forma *short*, essi hanno opcode ad 1 byte, mentre nella forma *long* hanno opcode a 2 byte, facendo uso dell'*escape* `0x0F`. Ciò è ben visibile dalla Figura 3.7.

Figura 3.7: Opcode a 1 e a 2 byte delle istruzioni x86

Gli opcode a singolo byte dei salti condizionati corrispondono a tutti quelli che vanno da `0x70` a `0x7F`. Le corrispondenti versioni *long*, hanno invece sempre `0x0F` come primo byte di opcode e come secondo byte un valore tra `0x80` a `0x8F`.

Dunque, per passare alla versione `long` basta sommare `0x10` all'opcode della versione *short* e utilizzarlo come secondo byte di opcode, preceduto dall'*escape byte*. Per quanto riguarda lo spiazzamento, si procede come nel caso dei salti incondizionati.

Come già fatto notare in precedenza, la sostituzione di un salto *short* con uno *long* implica un cambiamento di size e ciò provoca la necessità di ricalcolare i metadati relativi alla taglia, alla posizione e al padding delle sezioni. Inoltre, anche i salti precedentemente corretti vanno rivalutati.

3.7.4 Fixing dei program header e delle jump table

Per quanto riguarda i **program header**, avendo a disposizione il mapping tra segmenti e sezioni, è relativamente semplice ricostruirli correttamente. In base alle dimensioni e alle nuove posizioni delle sezioni contenute in ogni segmento, se ne calcolano anche in questo caso gli spiazzamenti e le variazioni di taglia da riportare nel rispettivo program header, nei campi `p_offset, p_filesz, p_memsz, p_vaddr` e `p_paddr`.

Invece, per le **jump table** si ha una gestione simile al fixing dei salti: per ogni entry della tabella, si va a considerare la nuova posizione dell'istruzione target associata. Il valore aggiornato da scrivere nella entry è calcolato come:

$$\text{newOffset} = \text{target.newAddress} - \text{jt.baseAddress}$$

cioè come differenza tra il nuovo indirizzo dell'istruzione target e l'indirizzo iniziale della branch table.

3.7.5 Riscrittura delle sezioni e dei simboli

La riscrittura delle **sezioni** viene effettuata allocando nuovi buffer per contenere i byte aggiornati associati alle strutture dati `Elf_Data`. Per quanto riguarda le sezioni che contengono codice eseguibile, si procede riempiendo il buffer con i byte delle istruzioni associate alla `Section_data` corrispondente, processando gli elementi della lista di istruzioni `insn_info_x86`. I contenuti

delle altre sezioni vengono riscritti rispecchiando il loro contenuto nella rappresentazione intermedia; la differenza è che non è necessario allocare nuovi buffer, in quanto la dimensione di queste sezioni non cambia, dal momento che non sono direttamente coinvolte nel processo di metamorfosi del codice.

Per quanto riguarda i **simboli**, quelli relativi alle funzioni vengono gestiti separatamente rispetto a quelli che indirizzano elementi presenti nelle sezioni dati. Per quest'ultima categoria, il cambiamento coinvolge unicamente il valore del simbolo, che viene ricalcolato sommando al valore originale la variazione di posizione subita dalla `Section_data` contenente il dato puntato. Ciò è sufficiente, in quanto le sezioni dati non cambiano la loro struttura interna rispetto al layout originale.

Per i simboli relativi a funzioni, invece, è necesssario sia cambiare il valore del simbolo, che corrisponde all'indirizzo iniziale della funzione, ma anche la size. L'aggiornamento è reso molto agevole grazie alla rappresentazione delle funzioni ottenuta tramite le strutture `Asm_function`, da cui è immediato riconoscere la nuova taglia della funzione. L'indirizzo iniziale, invece, è calcolato basandosi sui metadati associati alla prima istruzione che appartiene alla funzione.

3.8 L'expander

Per raggiungere l'obiettivo fissato per la seconda iterazione di sviluppo, ovvero ricostruire correttamente l'eseguibile aggiungendo un'istruzione `NOP`, non è stato necessario sviluppare l'expander dell'engine, ma è bastato, tramite dei test, aggiungere una corrispondente struttura `insn_info_x86` alle liste sia della sezione che della funzione da espandere.

Grazie al tracciamento dei cambiamenti e alla ricostruzione dell'ELF effettuata dall'assembler, il test è stato superato sia su programmi *handcrafted* che su alcuni degli eseguibili della suite *GNU Binutils* [1], come, ad esempio, il programma *strings*, usato per estrarre le stringhe da un file.

Successivamente, al fine di verificare il corretto funzionamento dei componenti dell'engine fin qui realizzati anche in scenari più complessi, con un livello di non determinismo più elevato, sono stati eseguiti numerosi test che prevedono l'aggiunta di molteplici istruzioni `NOP` in posizioni pseudo-casuali del codice.

A questo punto, realizzato lo scheletro dell'engine che permette di manipolare la rappresentazione intermedia dell'ELF e ricostruire con successo un file eseguibile in output che rispecchi le modifiche effettuate, lo sviluppo si è concentrato sull'expander dell'engine e sull'applicazione delle tecniche di metamorfismo.

Come descritto precedentemente in Sezione 2.5, per mantenere elevato il livello di astrazione e, allo stesso tempo, facilitare l'implementazione dei processi di metamorfosi, l'engine lavora direttamente sulle funzioni del file. Ognuna di esse (fatta eccezione per quelle che utilizzano *branch table*) viene sottoposta al processo di espansione, applicando, ove possibile, delle regole di trasformazione delle istruzioni e aggiungendo istruzioni *garbage* tra un'istruzione e l'altra in maniera random, con una certa probabilità.

Prima di attuare effettivamente le mutazioni, vengono associati alle singole istruzioni dei **token**, grazie ai quali è possibile classificare le istruzioni stesse rispetto alle regole di trasformazione previste.

3.8.1 Token associati delle istruzioni

Ad ogni istruzione della funzione che è attualmente processata dall'engine viene associato un token. Per ciascuna tipologia di token, è definito un sottoinsieme di regole potenzialmente applicabili, la cui fattibilità dovrà poi essere valutata in base alla semantica e alle caratteristiche dell'istruzione stessa.

L'utilizzo dei token permette di non essere necessariamente vincolati al concetto di opcode o mnemonico per rappresentare l'istruzione, consentendo di estendere l'engine definendo token che rappresentino più istruzioni o addirittura specifiche sequenze o pattern.

Per questa fase di sviluppo, essi sono semplicemente associati alle istruzioni in base al loro mnemonico. Un elenco dei tipi di token utilizzati è fornito nel Listato 3.16.

```
typedef enum Token_type{
  ADD,
  SUB,
  MOV,
  LEA,
  MUL,
  JUMP,
  CMP,
  JCC,
  CALL,
  PUSH,
  POP,
  OR,
  AND,
  XOR,
  RET,
  DEF,
  NOP
} Token_type;
```

Listato 3.16: Tipi di token associabili alle istruzioni

Dunque, le istruzioni di ogni funzione vengono processate in sequenza e, in base al loro token, viene selezionato un sottoinsieme di possibili regole di trasformazione. Se le regole applicabili sono molteplici, ne viene selezionata una in maniera casuale.

3.8.2 Regole di trasformazione

La fattibilità delle singole regole deve essere valutata dalla procedura che si occupa di effettuare la trasformazione. Infatti, le regole sono definite ad alto livello e non è detto che l'istruzione rispetti effettivamente i requisiti necessari affinché la mutazione sia applicabile (ad esempio, una regola può richiedere che i due registri operandi coincidano).

Al fine di comprendere meglio il concetto descritto, si presentano in Tabella 3.1 le regole di espansione applicabili in *MorphVM*.

Si supponga di dover processare un'istruzione con associato un token di tipo `XOR`; l'unica regola di trasformazione applicabile è quella che figura nella terz'ultima riga della Tabella 3.1, ovvero trasformare lo `XOR` di un registro con se stesso in una `MOV` di `0` nello stesso registro. In questo caso, la procedura

Istruzione originale	Istruzioni equivalenti
`MOV RegA, RegB`	`PUSH RegB / POP RegA`
`ADD Reg, Imm`	`ADD Reg, Imm / SUB Reg, Imm`
`SUB Reg, Imm`	`ADD Reg, Imm / SUB Reg, Imm`
`NOP`	`MOV Reg, Reg`
`XOR Reg, Reg`	`MOV Reg, 0`
`PUSH Reg`	`SUB rsp, 8 / MOV [rsp], Reg`
`POP Reg`	`MOV Reg, [rsp] / ADD rsp, 8`

Tabella 3.1: Regole di espansione di *MorphVM*

che si occupa di effettuare la trasformazione, controlla prima se essa sia effettivamente applicabile, andando a verificare che il registro sorgente e il registro destinazione dell'istruzione `XOR` coincidano. Se il controllo ha buon esito, si procede con la sostituzione dell'istruzione.

In questo caso, bisognerebbe generare una struttura di tipo `insn_info_x86` opportuna che rappresenti la nuova `MOV`, da inserire nella lista di istruzioni della funzione in sostituzione dello `XOR`. Tuttavia, ci sono diverse diverse modalità di codificare una simile istruzione in formato x86. Più in generale, questo discorso è valido per qualsiasi tipologia di istruzione. Considerare tutte le possibili combinazioni di opcode, registri e formati in maniera diretta sarebbe molto dispendioso, nonché sorgente molto probabile di errori.

3.8.3 *Keystone*

Per facilitare la creazione di nuove istruzioni mantenendo un livello di astrazione leggermente più alto del lavorare direttamente con i byte e gli opcode, si è deciso di fare uso di una libreria di assemblaggio di istruzioni: *Keystone* [28].

In questo modo, basta costruire una stringa che rappresenti l'istruzione che si intende generare e, dandola in pasto alla libreria, se ne otterranno i byte della codifica in x86. Disassemblando i byte con lo stesso disassembler descritto in Sezione 3.3, si ottiene la struttura `insn_info_x86` corrispondente.

Di conseguenza, tutto il lavoro che viene svolto dall'expander, consiste principalmente nell'analisi dell'istruzione da trasformare e nella costruzione

di stringhe che rappresentino le istruzioni esito della trasformazione. A tal proposito, risulta essere necessario comprendere ogni singola istruzione come codifica i registri operandi: alcune, come la `POP` o la `PUSH`, inglobano la codifica direttamente nell'opcode; altre, come la `MOV`, utilizzano i campi `mod` ed `rm` del byte *ModR/M*, ma, a seconda dell'opcode cambia quale sia tra i due campi a specificare il registro sorgente e quale quello destinazione. Tenuto conto della casistica necessaria, al fine di ottenere le stringhe che rappresentano i registri operandi si mantiene un mapping tra codice e nome dei registri. Per poterlo risolvere è necessario considerare anche il valore di eventuali prefissi dell'istruzione (ad esempio, nel caso di registri `r8-r15`) e informazioni ottenute in fase di disassemblaggio riguardanti la size degli operandi (campi `op` e `opd_size` della struttura `insn_info_x86`). Si riporta la mappa dei nomi dei registri nel Listato 3.17.

```
1  char *base_r_64[] = {"rax", "rcx", "rdx", "rbx", "rsp", "rbp", "rsi", "rdi",
2      "r8", "r9", "r10", "r11", "r12", "r13", "r14", "r15"};
3  char *base_r_32[] = {"eax", "ecx", "edx", "ebx", "esp", "ebp", "esi", "edi"};
4  char *base_r_16[] = {"ax", "cx", "dx", "bx", "sp", "bp", "si", "di"};
5
6  char *get_name_from_reg(unsigned char reg,int flag){
7    if (flag == REG_SIZE_16){
8      return base_r_16[reg];
9    }
10   else if (flag == REG_SIZE_32){
11     return base_r_32[reg];
12   }
13   else{
14     return base_r_64[reg];
15   }
16 }
```

Listato 3.17: Mappa dei nomi dei registri

Come visibile dalla Tabella 3.1, le trasformazioni considerate possono essere 1 a 1, oppure 1 a 2, in riferimento al numero di istruzioni originarie e quelle risultato del cambiamento. Quando si applica una trasformazione, bisogna prestare attenzione a non disperdere le informazioni associate all'istruzione di partenza, come i riferimenti calcolati sulla base delle istruzioni di salto. Per evitare che ciò accada, parte del contenuto della struttura `insn_info_x86` dell'istruzione originale viene copiato nella struttura della prima tra le istruzioni

output della trasformazione. In questo modo, si preservano i riferimenti grazie ai quali l'assembler è in grado di ricostruire l'ELF correttamente.

3.8.4 Istruzioni *garbage*

Come dettagliatamente descritto in Sezione 2.5.1, l'expander tende a far crescere la size dell'eseguibile, inserendo in maniera pseudo-casuale delle istruzioni di *garbage* all'interno delle funzioni. Infatti, al termine di ogni processamento di un token associato ad un'istruzione della funzione, con probabilità pari al 5% delle volte, viene aggiunta un'istruzione spuria, che non impatta in alcun modo la semantica dell'eseguibile, ma contribuisce a creare rumore nella struttura. Le istruzioni *garbage* considerate sono riportate in Tabella 3.2.

Istruzione *garbage*
`NOP`
`MOV Reg, Reg`

Tabella 3.2: Istruzioni *garbage*

In Figura 3.8 si mette a confronto il codice originale della parte iniziale di una funzione di un file ELF e lo stesso codice ottenuto in seguito al processo di espansione; è possibile notare l'effetto dell'applicazione di alcune delle trasformazioni presentate in Tabella 3.1, riuscendo ad avere un'idea di come la forma del codice sia stata mutata, senza cambiarne la semantica.

```
; codice originale
0000000000001545 <foo3>:
    1545:  push   rbp
    1546:  mov    rbp,rsp
    1549:  sub    rsp,0x10
    154d:  mov    QWORD PTR [rbp-0x8],rdi
    1551:  lea    rax,[rip+0xffffffffffffd19]
    1558:  mov    QWORD PTR [rip+0x2af1],rax
    155f:  lea    rax,[rip+0xffffffffffffe88]
    1566:  mov    QWORD PTR [rip+0x2aeb],rax
    156d:  lea    rax,[rip+0xffffffffffffffb3]
    1574:  mov    QWORD PTR [rip+0x2ae5],rax
    157b:  cmp    QWORD PTR [rbp-0x8],0x0
    1580:  je     15c7 <foo3+0x82>
    1582:  mov    rcx,QWORD PTR [rbp-0x8]
    1586:  movabs rdx,0x5555555555555556
    1590:  mov    rax,rcx
    1593:  imul   rdx
    1596:  mov    rax,rcx
    1599:  sar    rax,0x3f
    159d:  sub    rdx,rax
    15a0:  mov    rax,rdx
    15a3:  add    rax,rax
    15a6:  add    rax,rdx
    15a9:  sub    rcx,rax
    15ac:  mov    rdx,rcx

    […]
```

```
; codice trasformato
00000000000015af <foo3>:
    15af:  sub    rsp,0x8
    15b3:  mov    QWORD PTR [rsp],rbp
    15b7:  mov    rbp,rsp
    15ba:  add    rsp,0x7a
    15be:  sub    rsp,0x8a
    15c5:  mov    QWORD PTR [rbp-0x8],rdi
    15c9:  lea    rax,[rip+0xffffffffffffcc9]
    15d0:  mov    QWORD PTR [rip+0x2a79],rax
    15d7:  lea    rax,[rip+0xffffffffffffe59]
    15de:  mov    QWORD PTR [rip+0x2a73],rax
    15e5:  lea    rax,[rip+0xffffffffffffff94]
    15ec:  mov    QWORD PTR [rip+0x2a6d],rax
    15f3:  cmp    QWORD PTR [rbp-0x8],0x0
    15f8:  je     163e <foo3+0x8f>
    15fa:  mov    rcx,QWORD PTR [rbp-0x8]
    15fe:  movabs rdx,0x5555555555555556
    1608:  push   rcx
    1609:  pop    rax
    160a:  imul   rdx
    160d:  push   rcx
    160e:  pop    rax
    160f:  sar    rax,0x3f
    1613:  sub    rdx,rax
    1616:  push   rdx
    1617:  pop    rax
    1618:  add    rax,rax
    161b:  add    rax,rdx
    161e:  sub    rcx,rax
    1621:  push   rcx
    1622:  pop    rdx
    1624:  nop
    […]
```

Figura 3.8: Esito del processo di metamorfosi applicato ad una funzione

4. Valutazione sperimentale

In questo capitolo, viene presentato lo studio di analisi effettuato per valutare sia l'efficacia dell'engine metamorfico realizzato e sia l'impatto che esso ha sull'eseguibile oggetto delle mutazioni in termini prestazionali, al variare dei passi di trasformazione.

4.1 Entropia e detection rate dei malware

La valutazione dell'efficacia del motore metamorfico è stata effettuata raccogliendo dati statistici su un sample di malware in formato ELF, sottoponendo ogni eseguibile alle azioni di mutazione del codice dell'engine. Per ogni malware del campione considerato, sono stati applicati più passi di trasformazione, valutando due metriche principali: l'**entropia** del file eseguibile e il **detection rate** del malware, analizzato da differenti motori di anti-virus. La prima metrica è indice della complessità interna del file e un suo aumento indicherebbe l'efficacia dell'engine. Analogamente, una riduzione del detection rate dei malware può essere indice del corretto funzionamento dello strumento realizzato, dal momento che riesce a modificare la forma dei file con successo, permettendo ad alcuni malware di sfuggire alle tecniche di detection statiche, perlopiù basate su firme, adottate da alcuni motori di anti-virus.

Sono state applicate, in ogni passo di trasformazione, tutte le regole elencate in Tabella 3.1 ogni qual volta fosse possibile farlo; la probabilità di inserimento di istruzioni *garbage* in fase di espansione è stata impostata pari al 5%.

I malware sono stati raccolti da database online. Il database principale da cui sono stati scaricati i file malevoli è *MalwareBazaar* [29], integrati poi da ulteriori malware recuperati da un repository pubblico [30]. Tuttavia, è stata necessario un processo di analisi e filtraggio dei file ottenuti, in maniera tale da scartare tutti i malware che non fossero compatibili con il formato previsto dall'engine di *MorphVM*, ottenendo quindi un campione formato unicamente da malware in formato ELF, compilati per architetture x86. La taglia totale del campione raccolto e utilizzato per l'analisi condotta ammonta a 87 eseguibili.

4.1.1 Entropia

Il concetto di entropia è molto usato nel campo della teoria dell'informazione, così come in quello della crittografia, rappresentando una misura della disorganizzazione e dell'incertezza di un sistema. L'entropia può essere applicata agli eseguibili come metrica di complessità, indice della distribuzione dei byte all'interno del file.

Il calcolo dell'entropia viene effettuato seguendo la **formula di Shannon**, introdotta nel 1948 da Claude Shannon [31]. Data una variabile aleatoria discreta X definita su un alfabeto Ω con distribuzione $p : \Omega \longrightarrow [0, 1]$, l'entropia di X è definita come:

$$H(X) := -\sum_{x \in \Omega} p(x) \log_2 p(x) = \mathbb{E}[-log_2 p(X)]$$

Nel caso di un file eseguibile, l'alfabeto Ω è l'insieme discreto di tutti i 2^8 possibili valori assumibili da un singolo byte: $\Omega = \{0, 1, 2, ..., 255\}$. Per ciascuno dei valori dell'alfabeto, se ne deriva la frequenza $p(x)$ come il numero di occorrenze di byte con valore x diviso il numero totale di byte del file.

L'entropia può assumere valori appartenenti al range $[0, 8]$; valori bassi sono indice di una bassa complessità del file, mentre valori più elevati possono indicare un livello di disordine maggiore. In pratica, file eseguibili comuni presentano valori di entropia inferiori a 5; invece, un'entropia compresa tra 5 e

6 caratterizza tipicamente file compressi, mentre valori ancora più elevati sono spesso indice di cifratura e offuscamento.

In generale, i malware hanno un'entropia più elevata di un file comune, dal momento che fanno spesso uso di tecniche di offuscamento e/o cifratura del codice stesso. Poiché l'engine metamorfico ha lo scopo di rendere complessa la struttura del binario su cui agisce, ci si aspetta che l'applicazione delle trasformazioni comporti un lieve aumento dell'entropia sui malware. Tuttavia, un aumento troppo elevato potrebbe essere utilizzato da sistemi di rilevamento come possibile indicatore di minaccia. Pertanto, viene considerato un buon risultato un aumento contenuto dell'entropia. I dati sono stati raccolti utilizzando il programma *ent* [32].

4.1.2 Detection rate

L'efficacia dell'engine metamorfico è stata valutata prendendo in considerazione anche una seconda metrica: la variazione del detection rate dei malware al variare dei passi di trasformazione. I malware sono stati sottoposti, tramite la piattaforma *VirusTotal* [2], ad un'analisi delle minacce di tipo statico, principalmente basata su firme, effettuata da numerosi motori di anti-virus, sia commerciali che open-source.

I risultati delle analisi hanno permesso di ottenere il detection rate di ciascun eseguibile, calcolato come il numero di anti-virus che lo hanno riconosciuto come minaccioso diviso il numero totali di anti-virus da cui il file è stato analizzato.

Per interagire con la piattaforma online di *VirusTotal* e raccogliere i dati necessari, sono state utilizzate le API messe a disposizione dalla piattaforma stessa, facendo uso della libreria *vt-py* [33] in un apposito programma sviluppato in linguaggio Python.

Ci si aspetta che, se l'engine è efficace, all'aumentare dei passi di trasformazione la detection rate dei malware diminuisca.

4.2 Prestazioni su benchmark

Per quanto riguarda l'impatto che l'engine ha sulle prestazioni dell'eseguibile che va a modificare, sono stati presi in considerazione due programmi che implementano algoritmi spesso utilizzati nell'ambito di benchmark prestazionali. Sono stati campionati i tempi di esecuzione di ognuno dei benchmark per ogni step di trasformazione a cui sono stati sottoposti, analizzandone l'andamento medio e la varianza all'incrementare dei passi.

Il campionamento dei tempi è stato effettuato utilizzando il programma */usr/bin/time* in ambiente virtualizzato. Le caratteristiche tecniche della macchina virtuale adottata per i test prestazionali sono riportate in Tabella 4.1.

Caratteristiche tecniche VM per i test prestazionali	
Numero di CPU	2
Memoria RAM	8 GB
Versione del kernel Linux	6.1.0
Architettura hardware	x86_64
Virtualizzatore	Oracle VM VirtualBox (v. 7.0.2 r154219)

Tabella 4.1: Caratteristiche tecniche della macchina virtuale adottata per i test prestazionali

4.2.1 *Pagerank* e *Moon-calc*

I due programmi utilizzati per i test sono **Pagerank** [34] e **Moon-calc** [35], entrambi open-source.

Il primo è un'implementazione in linguaggio C del noto algoritmo *pagerank* di *Google*, utilizzato dal motore di ricerca per effettuare il ranking delle pagine web: esso determina l'importanza di ogni singola web page contando il numero e il peso dei link che la riferiscono in altre pagine. L'idea di fondo è che siti web più importanti abbiano una probabilità maggiore di essere riferiti da altre web page.

Il benchmark in questione calcola i rank di ogni pagina web di una rete con cui viene configurato, costruendo una matrice quadrata dei link tra le pagine

e procedendo a calcolarne gli autovettori tramite la procedura iterativa che prende il nome di "metodo delle potenze". Il programma è stato configurato in maniera tale da considerare una rete formata da **10000 pagine web**, i cui link vengono generati in maniera pseudo-casuale, ma inizializzando il *random number generator* con un seme *harcoded*, così che la rete risultante su cui opera l'algoritmo sia sempre la stessa ad ogni run e i tempi di esecuzione non differiscano a causa di configurazioni differenti. Per ogni passo di trasformazione, sono state eseguite 50 run del programma, raccogliendone i tempi.

Per quanto riguarda il benchmark **Moon-calc**, è anch'esso scritto in linguaggio C e riguarda un test che stima la velocità nell'effettuare una serie di calcoli astronomici relativi alla posizione della luna, approssimata da una soluzione armonica, risultando in uno stress test delle componenti di calcolo dell'architettura sottoposta al test prestazionale. In questo caso, per ogni passo di trasformazione, sono state eseguite 10 run del benchmark.

4.3 Analisi dei risultati

In questa sezione, vengono presentati i risultati dell'analisi condotta.

Ogni malware del campione raccolto è stato sottoposto a 2 passi di trasformazione dell'engine. Si riporta in Tabella 4.2 l'**entropia** dei 10 malware che hanno registrato il maggior incremento rispetto al valore di partenza; invece, in Figura 4.1 è mostrato un boxplot che raffigura la distribuzione dell'entropia di tutti i malware nella loro forma base e in seguito a ciascuno dei due passi di trasformazione.

Come ci si aspettava, l'entropia media tende ad aumentare con i passi di trasformazione a cui sono sottoposti i malware. Ciò è indice dell'offuscamento e dell'aggiunta di complessità introdotta dall'engine metamorfico. Dal grafico di Figura 4.1, si può notare come la mediana della distribuzione salga da un valore pari a 6 al passo zero, fino ad un valore di circa 6.3 al passo di

Malware	Passo 0	Passo 1	Passo 2
1	4.970325	5.524114	5.622446
2	5.146430	5.332326	5.715483
3	5.957822	6.005625	6.387666
4	5.907044	5.960619	6.336268
5	5.906696	5.959098	6.331521
6	5.939981	6.062893	6.353623
7	5.963298	6.012901	6.373268
8	6.016485	6.083948	6.415380
9	5.952054	6.054312	6.341226
10	5.953336	6.052516	6.341471

Tabella 4.2: Entropia dei 10 malware che hanno registrato il maggior incremento al variare dei passi di trasformazione

trasformazione numero due; inoltre, anche l'entropia degli outlier che avevano evidenziato valori più bassi tende a salire.

L'incremento è ragionevole, considerando che i malware hanno, di base, un'entropia elevata, indice di una complessità non banale dei file. Nonostante ciò, l'engine di *MorphVM* contribuisce comunque ad introdurre ulteriore rumore nella struttura degli ELF, evidenziando la sua efficacia.

Un comportamento in linea con le aspettative è stato osservato anche per quanto riguarda la variazione del **detection rate** dei malware all'avanzare dei passi di trasformazione. Nella Tabella 4.3 sono riportati i 10 malware che hanno fatto registrare il detection rate più basso nell'ultimo passo di trasformazione, mentre nel boxplot in Figura 4.2 è visibile l'andamento generale del detection rate di tutti gli 87 malware del campione considerato.

Come ci si attendeva, il detection rate dei malware tende a diminuire modificando la struttura degli eseguibili tramite l'engine. Ciò è dovuto principalmente al cambiamento della firma dell'ELF, ma anche ad eventuali modifiche apportate ai pattern di istruzioni di cui alcuni engine degli anti-virus vanno alla ricerca per classificare un eseguibile come malevolo.

Dal boxplot di Figura 4.2 si vede come con un unico passo di trasformazione il detection rate raggiunge valori molto bassi (vicini al 10%) per alcuni malware,

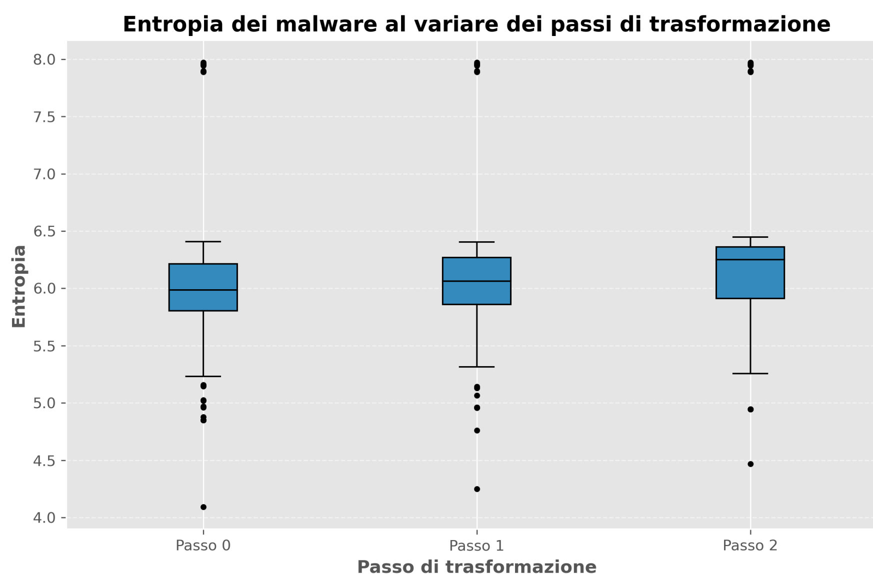

Figura 4.1: Entropia dei malware al variare dei passi di trasformazione

Malware	Passo 0	Passo 1	Passo 2
1	0.43	0.11	0.08
2	0.58	0.10	0.08
3	0.53	0.10	0.10
4	0.58	0.10	0.10
5	0.56	0.15	0.15
6	0.60	0.37	0.15
7	0.61	0.10	0.15
8	0.63	0.15	0.15
9	0.61	0.16	0.16
10	0.66	0.43	0.16

Tabella 4.3: Detection rate dei 10 malware che hanno registrato il minor detection rate nell'ultimo passo di trasformazione

ma, in generale, la mediana risulta assestarsi intorno al 50%, rispetto al 65% di partenza. Invece, al secondo step di trasformazione, la mediana si riduce fino a circa il 30% di detection rate, indicando che l'offuscamento introdotto dall'engine metamorfico riesce a far eludere le tecniche di detection statiche adottate dalla maggior parte dei motori di anti-virus. In ogni caso, va segnalata comunque la presenza di un certo numero di malware che non beneficiano molto dei cambiamenti a cui sono stati sottoposti, essendo classificati come eseguibili

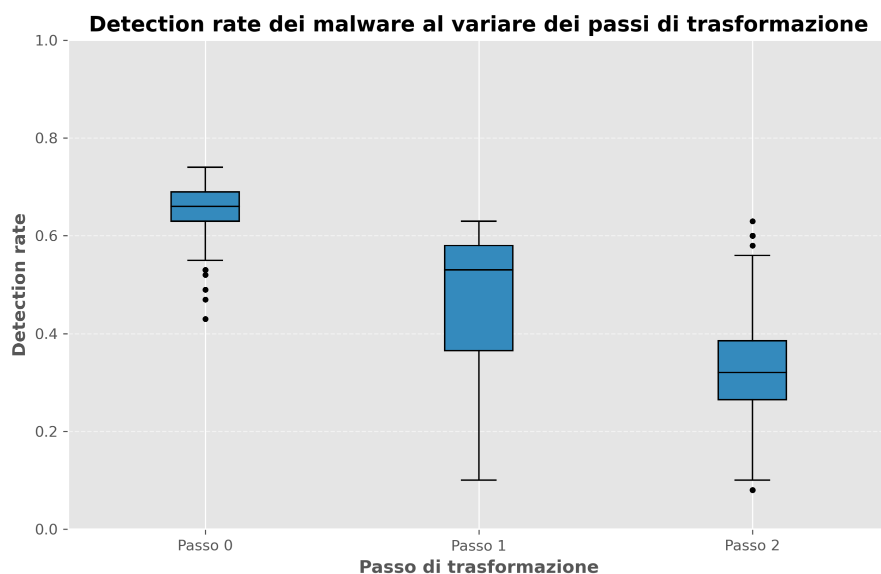

Figura 4.2: Detection rate dei malware al variare dei passi di trasformazione

malevoli da più della metà degli anti-virus.

Grazie ai risultati forniti da *VirusTotal*, è stato possibile anche effettuare un'analisi degli anti-virus, individuando quelli maggiormente soggetti a sbagliare classificazione in seguito all'applicazione delle tecniche di metamorfismo usate nell'engine di *MorphVM* e quelli che, invece, risultano essere più robusti. I risultati ottenuti sono rappresentati da grafici a barre nelle Figure 4.3 e 4.4.

In Figura 4.3 sono riportati i primi 5 anti-virus tra quelli che avevano registrato un maggior detection rate al passo zero, ma che hanno riportato il calo di prestazioni maggiore tra il passo zero e il passo uno. Il motore *ESET-NOD32* risulta essere particolarmente vulnerabile alle trasformazioni applicate dall'engine di *MorphVM* ai malware, con una riduzione del detection rate dal 98% al passo zero al 21% nel passo uno.

Invece, in Figura 4.4, sono graficati i detection rate dei primi 5 motori di anti-virus che hanno riportato il calo di prestazioni minore tra il passo zero e il passo due. Gli anti-virus più robusti alle tecniche adottate dall'engine metamorfico risultano essere *Avast* e *AVG*, probabilmente indice del fatto che effettuano detection tramite meccanismi che non sono oscurati dall'engine di *MorphVM*, se non in minima parte.

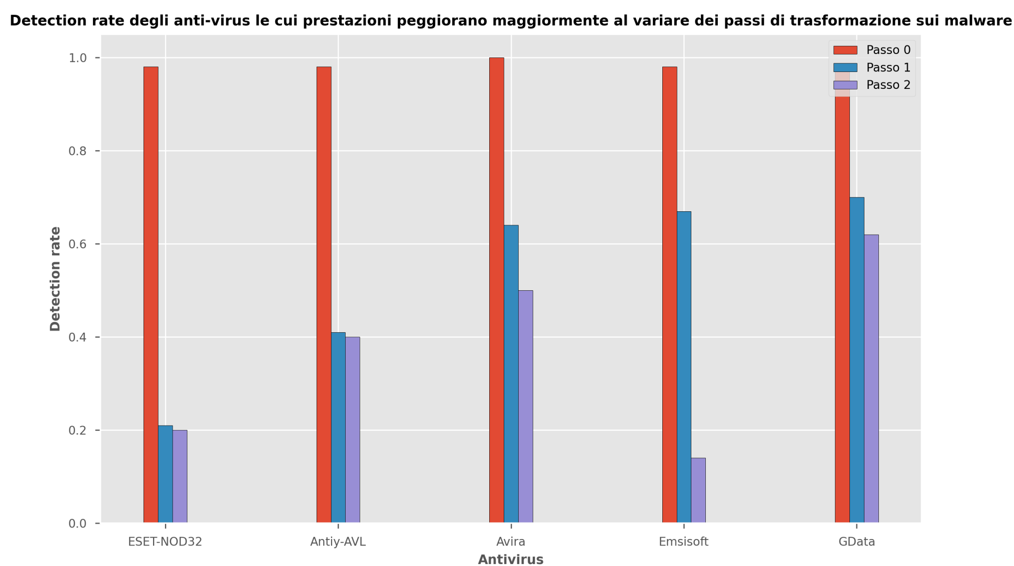

Figura 4.3: Anti-virus le cui prestazioni peggiorano maggiormente al variare dei passi di trasformazione dei malware

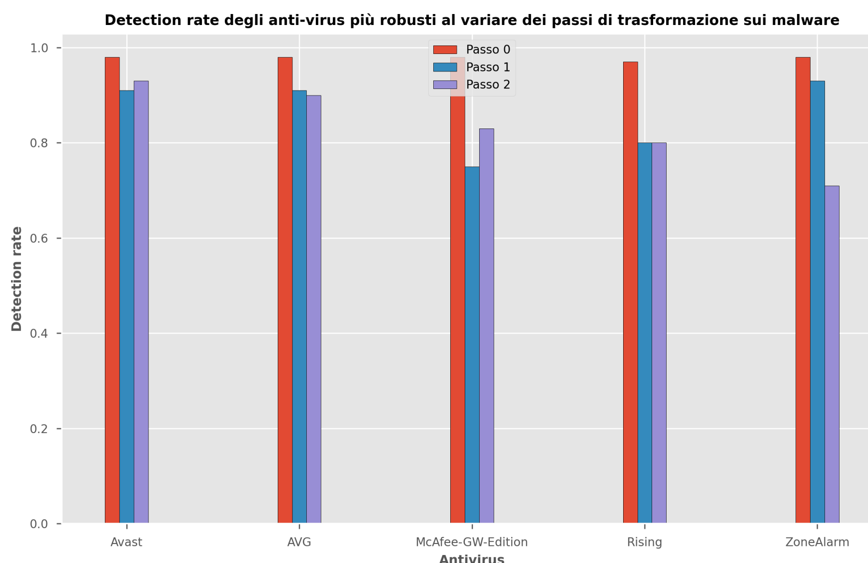

Figura 4.4: Anti-virus maggiormente robusti alle trasformazioni dei malware

Infine, i risultati dei test effettuati sui **benchmark** *Pagerank* e *Moon-calc* sono riportati rispettivamente nelle Figure 4.5 e 4.6, in cui sono presentati i tempi di esecuzione dei programmi al variare dei passi di trasformazione a cui sono stati sottoposti.

I grafici mostrano come, all'aumentare dei passi di trasformazione, i tempi

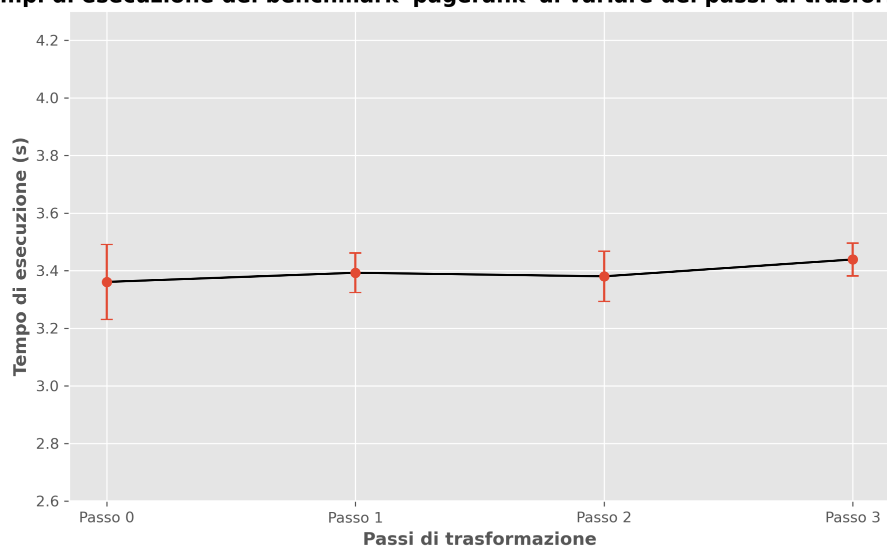

Figura 4.5: Tempi di esecuzione del benchmark *Pagerank* al variare dei passi di trasformazione

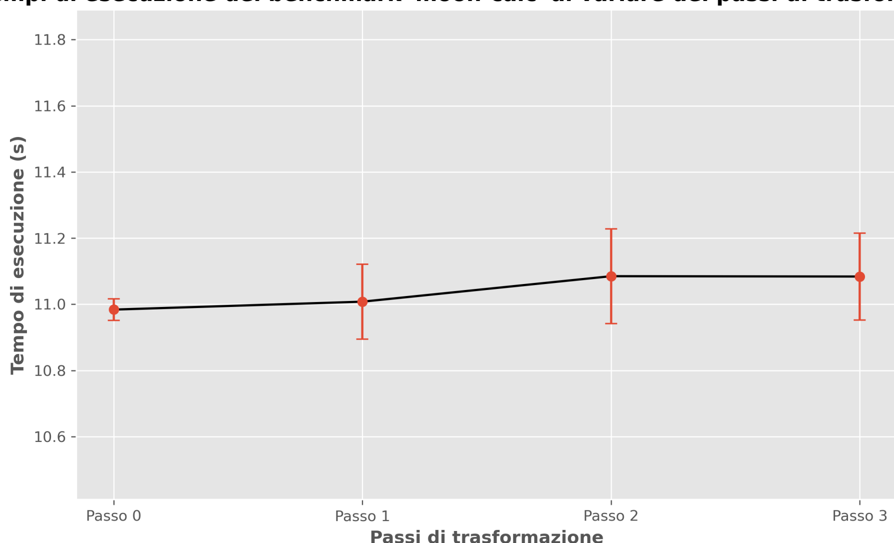

Figura 4.6: Tempi di esecuzione del benchmark *Moon-calc* al variare dei passi di trasformazione

medi di esecuzione tendano a crescere, seppur in maniera lieve, in linea con quanto ci si poteva attendere. Questo comportamento è principalmente dovuto all'applicazione delle sole tecniche di espansione e aggiunta di istruzioni *garbage* da parte dell'engine, che tendono a far crescere il codice del programma.

Per quanto riguarda il benchmark *Moon-calc*, come visibile dalla Figura 4.6, è da registrare anche un leggero incremento della varianza dei tempi di esecuzione in seguito all'applicazione delle mutazioni. Tale andamento può essere dovuto all'incremento di complessità introdotto dall'engine e alla modifica di istruzioni che possono ridurre l'impatto delle direttive di ottimizzazione con cui il programma originario era stato compilato.

5. Conclusioni e future work

L'obiettivo di questa tesi è stato quello di presentare una metodologia innovativa per la protezione della proprietà intellettuale del software. È stato esaminato l'utilizzo di tecniche di offuscamento e metamorfismo, spesso presenti in malware come armi offensive, ma utilizzate con scopi difensivi nella progettazione e realizzazione di un'engine metamorfico.

Nonostante l'engine implementato manchi dei componenti responsabili delle fasi di permutazione e compressione delle istruzioni macchina, i risultati ottenuti dall'applicazione di semplici regole di espansione sono in linea con le aspettative iniziali e dimostrano come la soluzione proposta risulti essere efficace, riuscendo ad aumentare la complessità interna dell'eseguibile, mantenendone intatta la semantica.

Sono possibili numerosissime migliorie da integrare nel progetto: in primis, l'aggiunta del permutator contribuirebbe ad aumentare ulteriormente il livello di complessità delle nuove versioni dell'eseguibile e l'implementazione dello shrinker sarebbe di vitale importanza per evitare che le dimensioni del programma crescano indefinitamente di generazione in generazione.

Inoltre, l'insieme delle regole di trasformazione ed equivalenza tra istruzioni può essere espanso, prendendo spunto dalle regole di MetaPHOR, così da incrementare il livello di non determinismo dell'engine e delle nuove versioni della VM prodotte.

Potrebbe essere interessante estendere lo studio effettuato sulle prestazioni dei motori di anti-virus, andando a valutare il loro comportamento rispetto all'applicazione di singole regole di trasformazione o combinazioni di esse. In questo modo, si potrebbero individuare le regole che risultano essere più efficaci

rispetto anche alle tecniche di detection adottate dai singoli anti-virus.

Per incrementare il non determinismo del processo di metamorfosi e l'efficacia dell'engine, si potrebbe ricorrere all'utilizzo di **approcci generativi**. In particolare, l'utilizzo di grammatiche generative [17, 36, 37, 38] può essere interessante per la generazione di strutture dati ad albero che descrivano il processo di espansione delle istruzioni, applicando regole in maniera ricorsiva. L'espansione vera e propria potrebbe avvenire processando tali alberi. Inoltre, un aspetto ancora più interessante è quello di utilizzare grammatiche che siano in grado di generare i set di regole di trasformazione, così da applicare regole diverse in ogni versione della VM. In questo modo, si ridurrebbe di molto la possibilità per un attaccante di individuare pattern di mutazione analizzando le diverse versioni della macchina virtuale, innalzandone, di conseguenza, la robustezza al reversing.

Per realizzare un meccanismo simile, si potrebbe far ricorso a tool come *Flex* e *Bison* [39], per definire una meta-grammatica grazie alla quale poter effettuare il parsing di semplici grammatiche definite con la sintassi prevista da *Bison*, in cui specificare le regole di trasformazione dell'engine. Il tutto potrebbe essere integrato nell'engine metamorfico che, a partire dalla rappresentazione delle regole che compongono la grammatica fornita in input, potrebbe essere in grado di generare le trasformazioni da applicare alle istruzioni, considerando quest'ultime come token di partenza per l'albero generativo.

Ad esempio, si supponga che nella grammatica siano definite le seguenti regole (in maiuscolo sono indicati i simboli terminali):

$$\text{mov} \rightarrow \text{push pop} \mid \text{MOV}$$

$$\text{push} \rightarrow \text{NOP PUSH} \mid \text{PUSH}$$

$$\text{pop} \rightarrow \text{NOP POP} \mid \text{POP} \mid \text{MOV ADD}$$

Se ad un'istruzione di `MOV` viene associato il token *mov*, un possibile albero generato dalla grammatica in fase di espansione dell'eseguibile, può prevedere

l'espansione di *mov* nei token *push* e *pop*, i quali, a loro volta, potrebbero essere espansi rispettivamente nei simboli terminali `NOP POP` e `MOV ADD`. Una rappresentazione grafica è fornita in Figura 5.1. L'engine potrebbe navigare la struttura dati ad albero, applicando la trasformazione generata all'istruzione `MOV` di partenza.

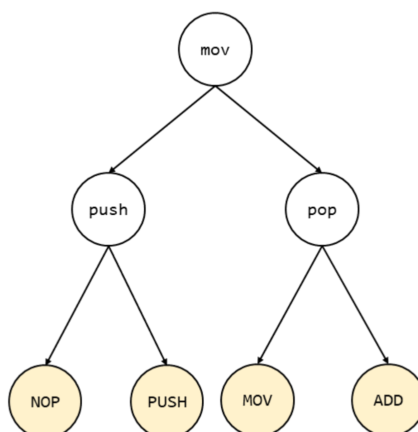

Figura 5.1: Esempio di albero generato da una grammatica

Per rendere l'engine ancora più efficace, si potrebbero utilizzare le grammatiche per generare addirittura i set di regole di trasformazione considerati, codificati secondo uno specifico formato interpretabile dall'engine. Ciò introdurrebbe ancora più variabilità tra le differenti versioni della macchina virtuale e complicherebbe enormemente i tentativi di analisi e reversing del codice.

A. Regole di trasformazione di MetaPHOR

Istruzione originale	Istruzione equivalente
XOR Reg,-1	NOT Reg
XOR Mem,-1	NOT Mem
MOV Reg,Reg	NOP
SUB Reg,Imm	ADD Reg,-Imm
SUB Mem,Imm	ADD Mem,-Imm
XOR Reg,0	MOV Reg,0
XOR Mem,0	MOV Mem,0
ADD Reg,0	NOP
ADD Mem,0	NOP
OR Reg,0	NOP
OR Mem,0	NOP
AND Reg,-1	NOP
AND Mem,-1	NOP
AND Reg,0	MOV Reg,0
AND Mem,0	MOV Mem,0
XOR Reg,Reg	MOV Reg,0

SUB Reg,Reg	MOV Reg,0
OR Reg,Reg	CMP Reg,0
AND Reg,Reg	CMP Reg,0
TEST Reg,Reg	CMP Reg,0
LEA Reg,[Imm]	MOV Reg,Imm
LEA Reg,[Reg+Imm]	ADD Reg,Imm
LEA Reg,[Reg2]	MOV Reg,Reg2
LEA Reg,[Reg+Reg2]	ADD Reg,Reg2
LEA Reg,[Reg2+Reg2+xxx]	LEA Reg,[2*Reg2+xxx]
MOV Reg,Reg	NOP
MOV Mem,Mem	NOP

Tabella A.1: Regole di trasformazione di MetaPHOR con mapping 1 ad 1

Istruzioni originali	**Istruzione equivalente**
PUSH Imm POP Reg	MOV Reg,Imm
PUSH Imm POP Mem	MOV Mem,Imm
PUSH Reg POP Reg2	MOV Reg2,Reg
PUSH Reg POP Mem	MOV Mem,Reg
PUSH Mem POP Reg	MOV Reg,Mem
PUSH Mem POP Mem2	MOV Mem2,Mem
MOV Mem,Reg PUSH Mem	PUSH Reg
POP Mem MOV Reg,Mem	POP Reg
POP Mem2 MOV Mem,Mem2	POP Mem
MOV Mem,Reg MOV Reg2,Mem	MOV Reg2,Reg
MOV Mem,Imm PUSH Mem	PUSH Imm
MOV Mem,Imm OP Reg,Mem	OP Reg,Imm
MOV Reg,Imm ADD Reg,Reg2	LEA Reg,[Reg2+Imm]
MOV Reg,Reg2 ADD Reg,Imm	LEA Reg,[Reg2+Imm]

MOV Reg,Reg2 ADD Reg,Reg3	LEA Reg,[Reg2+Reg3]
ADD Reg,Imm ADD Reg,Reg2	LEA Reg,[Reg+Reg2+Imm]
ADD Reg,Reg2 ADD Reg,Imm	LEA Reg,[Reg+Reg2+Imm]
OP Reg,Imm OP Reg,Imm2	OP Reg,(Imm OP Imm2)
OP Mem,Imm OP Mem,Imm2	OP Mem,(Imm OP Imm2)
LEA Reg,[Reg2+Imm] ADD Reg,Reg3	LEA Reg,[Reg2+Reg3+Imm]
LEA Reg,[(RegX+)Reg2+Imm] ADD Reg,Reg2	LEA Reg,[(RegX)2*Reg2+Imm]
POP Mem PUSH Mem	NOP
MOV Mem2,Mem MOV Mem3,Mem2	MOV Mem3,Mem
MOV Mem2,Mem OP Reg,Mem2	OP Reg,Mem
MOV Mem2,Mem MOV Mem2,xxx	MOV Mem2,xxx
MOV Mem,Reg CALL Mem	CALL Reg
MOV Mem,Reg JMP Mem	JMP Reg
MOV Mem2,Mem CALL Mem2	CALL Mem
MOV Mem2,Mem JMP Mem2	JMP Mem

MOV Mem,Reg MOV Mem2,Mem	MOV Mem2,Reg
OP Reg,xxx MOV Reg,yyy	MOV Reg,yyy
Jcc @xxx !Jcc @xxx	JMP @xxx
NOT Reg NEG Reg	ADD Reg,1
NOT Reg ADD Reg,1	NEG Reg
NOT Mem NEG Mem	ADD Mem,1
NOT Mem ADD Mem,1	NEG Mem
NEG Reg NOT Reg	ADD Reg,-1
NEG Reg ADD Reg,-1	NOT Reg
NEG Mem NOT Mem	ADD Mem,-1
NEG Mem ADD Mem,-1	NOT Mem
NEG Mem != Jcc (CMP without Jcc)	NOP
TEST X,Y != Jcc	NOP
POP Mem JMP Mem	RET
PUSH Reg RET	JMP Reg

MOV Reg,Mem CALL Reg	CALL Mem
XOR Reg,Reg MOV Reg8,[Mem]	MOVZX Reg,byte ptr[Mem]
MOV Reg,[Mem] AND Reg, 0FFh	MOVZX Reg,byte ptr[Mem]

Tabella A.2: Regole di trasformazione di MetaPHOR con mapping 2 ad 1

Istruzioni originali	**Istruzione equivalente**

`MOV Mem,Reg`
`OP Mem,Reg2`
`MOV Reg,Mem`
	`OP Reg,Reg2`

`MOV Mem,Reg`
`OP Mem,Imm`
`MOV Reg,Mem`
	`OP Reg,Imm`

`MOV Mem,Imm`
`OP Mem,Reg`
`MOV Reg,Mem`
	`OP Reg,Imm`

`MOV Mem2,Mem`
`OP Mem2,Reg`
`MOV Mem,Mem2`
	`OP Mem,Reg`

`MOV Mem2,Mem`
`OP Mem2,Imm`
`MOV Mem,Mem2`
	`OP Mem,Imm`

`CMP Reg,Reg`
`JO/JB/JNZ/JA/JS/JNP/JL/JG @xxx`
`!= Jcc`
	`NOP`

`CMP Reg,Reg`
`JNO/JAE/JZ/JBE/JNS/JP/JGE/JLE`
`@xxx`
`!= Jcc`
	`JMP @xxx`

`PUSH EAX`
`PUSH ECX`
`PUSH EDX`
	`APICALL_BEGIN`

`POP EDX`
`POP ECX`
`POP EAX`
	`APICALL_END`

Tabella A.3: Regole di trasformazione di MetaPHOR con mapping 3 ad 1

Istruzioni originali	Istruzioni equivalenti
MOV Mem,Imm CMP/TEST Reg,Mem Jcc @xxx	CMP/TEST Reg,Imm Jcc @xxx
MOV Mem,Reg SUB/CMP Mem,Reg2 Jcc @xxx	CMP Reg,Reg2 Jcc @xxx
MOV Mem,Reg AND/TEST Mem,Reg2 Jcc @xxx	TEST Reg,Reg2 Jcc @xxx
MOV Mem,Reg SUB/CMP Mem,Imm Jcc @xxx	CMP Reg,Imm Jcc @xxx
MOV Mem,Reg AND/TEST Mem,Imm Jcc @xxx	TEST Reg,Imm Jcc @xxx
MOV Mem2,Reg CMP/TEST Reg,Mem2 Jcc @xxx	CMP/TEST Reg,Mem Jcc @xxx
MOV Mem2,Mem AND/TEST Mem2,Reg Jcc @xxx	TEST Mem,Reg Jcc @xxx
MOV Mem2,Mem SUB/CMP Mem2,Reg Jcc @xxx	CMP Mem,Reg Jcc @xxx
MOV Mem2,Mem AND/TEST Mem2,Imm Jcc @xxx	TEST Mem,Imm Jcc @xxx

`MOV Mem2,Mem`
`SUB/CMP Mem2,Imm`
`Jcc @xxx`
	`CMP Mem,Imm`
	`Jcc @xxx`

Tabella A.4: Regole di trasformazione di MetaPHOR con mapping 3 a 2

Bibliografia

[1] Free Software Foundation. *GNU Binutils*. URL: https://www.gnu.org/software/binutils/.

[2] *VirusTotal website*. https://www.virustotal.com/gui/home/upload.

[3] Bradevanrosen. *46 DC EA D3 17 FE 45 D8 09 23 EB 97 E4 95 64 10 D4 CD B2 C2 – by "Ben S"*. 2011. URL: https://yalelawtech.org/2011/03/01/46-dc-ea-d3-17-fe-45-d8-09-23-eb-97-e4-95-64-10-d4-cd-b2-c2/.

[4] Wikipedia. *Sony Computer Entertainment America, Inc. v. Hotz*. 2011. URL: https://en.wikipedia.org/wiki/Sony_Computer_Entertainment_America,_Inc._v._Hotz.

[5] N.D.C.A. *Sony Computer Entertainment America LLC v. Hotz (Docket Report), vol. No. 3:11-cv-00167*. 2011. URL: https://ia600204.us.archive.org/14/items/gov.uscourts.cand.235965/gov.uscourts.cand.235965.docket.html.

[6] Runhao Wang et al. «DeepTrace: A Secure Fingerprinting Framework for Intellectual Property Protection of Deep Neural Networks». In: *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. 2021, pp. 188–195. DOI: 10.1109/TrustCom53373.2021.00042.

[7] Siping Zeng e Xiaozhen Guo. «Research on Key Technology of Software Intellectual Property Protection». In: *2021 International Conference on Intelligent Transportation, Big Data & Smart City (ICITBS)*. 2021, pp. 329–332. DOI: 10.1109/ICITBS53129.2021.00088.

[8] Stephen Drape. «Intellectual property protection using obfuscation». In: (2010).

[9] Robert NM Watson e David Dagon. «First USENIX Workshop on Offensive Technologies (WOOT'07)». In: (2007).

[10] Sameer S Shende e Allen D Malony. «The TAU parallel performance system». In: *The International Journal of High Performance Computing Applications* 20.2 (2006), pp. 287–311.

[11] James Newsome et al. «Vulnerability-Specific Execution Filtering for Exploit Prevention on Commodity Software.» In: *NDSS*. 2006.

[12] *Pin*. `https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html`.

[13] *Dyninst*. `https://github.com/dyninst/dyninst`.

[14] *Valgrind*. `https://valgrind.org/`.

[15] *DynamoRIO*. `https://dynamorio.org/`.

[16] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z*. Intel Corporation. 2022.

[17] Éric Filiol. «Metamorphism, Formal Grammars and Undecidable Code Mutation». In: *International Journal of Computer and Information Engineering* 1.2 (2007), pp. 281–286.

[18] Peter Szor e Peter Ferrie. «Hunting for metamorphic». In: *Virus bulletin conference*. Citeseer. 2001.

[19] P. Szor P. Ferrie. *Zmnist opportunities*. Virus analysis. Oxfordshire (UK), 2001.

[20] The Mental Driller. *Metamorphism in practice or "How I made MetaPHOR and what I've learnt"*. Rapp. tecn. VX Heaven, 2002. URL: `https://web.archive.org/web/20130808230413/http://vxheaven.org/lib/vmd01.html`.

[21] The Mental Driller. *MetaPHOR*. Codice sorgente del virus MetaPHOR. Disponibile all'indirizzo https://github.com/mal-project/win32.MetaPHOR/blob/master/METAPHOR.ASM. 2002.

[22] Jean-Marie Borello. «Étude du métamorphisme viral: modélisation, conception et détection». Tesi di dott. École doctorale Matisse, Université de Rennes 1, 2012.

[23] Joseph Koshy. *libelf by Example*. 2010. URL: %5Curl%7Bhttps://atakua.org/old-wp/wp-content/uploads/2015/03/libelf-by-example-20100112.pdf%7D.

[24] Linux Foundation. *Symbol Table Documentation*. URL: https://refspecs.linuxbase.org/elf/gabi4+/ch4.symtab.html.

[25] Alessandro Pellegrini. «Hijacker: Efficient static software instrumentation with applications in high performance computing: Poster paper». In: *2013 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2013, pp. 650–655.

[26] D. Cingolani A. Pellegrini S. Economo. *Hijacker*. HPDCS. URL: https://github.com/HPDCS/hijacker.

[27] Oracle Corporation. *Man pages section 3, Extended Library Functions, elf_update*. URL: https://docs.oracle.com/cd/E19683-01/816-0217/6m6nhtaa8/index.html.

[28] *Keystone*. URL: https://www.keystone-engine.org.

[29] *Malware Bazaar*. URL: https://bazaar.abuse.ch/.

[30] *Repository malware-samples*. URL: https://github.com/InQuest/malware-samples.

[31] Claude E Shannon. «A mathematical theory of communication». In: *The Bell system technical journal* 27.3 (1948), pp. 379–423.

[32] *ent*. URL: https://www.fourmilab.ch/random/.

[33] *vt-py*. URL: https://github.com/VirusTotal/vt-py.

[34] *Pagerank implementation in C.* URL: https://github.com/andreacarrara/pagerank.

[35] *Moon-calc.* URL: https://github.com/paulrho/mthtest2.

[36] Jason Gerald Young. «Code Generation: An Introduction to Typed EBNF». In: (2015).

[37] Zhiwu Xu, Lixiao Zheng e Haiming Chen. «A toolkit for generating sentences from context-free grammars». In: *2010 8th IEEE International Conference on Software Engineering and Formal Methods*. IEEE. 2010, pp. 118–122.

[38] Augusto Celentano et al. «Compiler testing using a sentence generator». In: *Software: Practice and Experience* 10.11 (1980), pp. 897–918.

[39] John Levine. *Flex & Bison: Text Processing Tools*. " O'Reilly Media, Inc.", 2009.

Ringraziamenti

Ci tengo a dedicare uno spazio nell'elaborato alle persone che mi hanno sostenuto durante questo percorso di studi. In primis, vorrei ringraziare di cuore il mio relatore Alessandro Pellegrini, una persona verso cui nutro un'immensa stima e che si è dimostrata disponibile, gentile e simpatica.

Un ringraziamento speciale va alla mia famiglia, che mi ha sempre sostenuto in questo percorso e mi ha trasmesso i valori che mi hanno reso la persona che sono oggi.

Grazie a tutti i colleghi e le colleghe con i quali ho avuto il piacere di collaborare e condividere idee ed opinioni. Se la competitività e il confronto hanno contribuito al mio percorso di crescita, il sostegno reciproco e le risate hanno reso quest'esperienza indimenticabile.

Un grazie particolare va a Matteo, compagno di mille progetti; non avrei potuto desiderare un amico e collega migliore.

Grazie infinite anche a Valerio, l'amico che c'è sempre stato nel momento del bisogno e che mi ha sempre sostenuto: MVP!

Grazie a Niccolò e Pasquale, due calabresi di quelli originali! Grazie per la vostra dose giornaliera di risate ed allegria.

Grazie di cuore anche a Chiara, amica e confidente; grazie per la tua spontaneità e semplicità, non cambiare mai.

Infine, grazie a tutti coloro che mi hanno sostenuto anche solo con un pensiero, vi voglio bene.