
POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

# Tecniche di Binary Code Similarity e Binary Diffing per la classificazione di malware

**Relatore**
prof. Giovanni Squillero
dott.ing. Andrea Marcelli

**Candidato**
Daniele Bevilacqua

Anno accademico 2018-2019

# Indice

# 1 Sommario

La tesi affronta il problema della valutazione della similarità tra due file binari (**Binary Code Similarity o Binary Diffing**), che consiste nel determinare, a partire dal codice compilato, se le funzioni presenti all'interno dei due file sono simili tra loro. Il problema trova diverse applicazioni pratiche nell'ambito della sicurezza informatica, in particolare nel campo della malware analysis e nell'individuazione di vulnerabilità presenti all'interno delle applicazioni.

Durante la prima fase del lavoro condotto, si è analizzato lo stato dell'arte dei modelli utilizzati per affrontare la problematica della Binary Code Similarity Detection e della Binary Diffing. Nello specifico, lo stato dell'arte è rappresentato da modelli basati sugli "embeddings", una tecnica di machine learning che permette di associare a ciascuna porzione di codice un vettore all'interno di uno spazio n-dimensionale. Successivamente applicando algoritmi basate sulle metriche degli spazi vettoriali è possibile catturare la somiglianza sintattica e semantica dei binari.

Le tecniche esistenti differiscono nel pre-processamento utilizzato per estrarre gli embeddings, come l'estrazione del Control Flow Graph (CFG), o l'utilizzo di attributi statistici e strutturali caratteristici dei programmi.

Altri modelli basano il loro funzionamento sugli algoritmi di Locality Sensitive Hashing (LSH) applicati ai frammenti di codice che costituiscono il binario. In tali modelli la somiglianza tra due funzioni è data dalla similarità dei loro hash. In questo caso vengo applicati degli algoritmi di hash che garantiscono maggiore probabilità di collisione per oggetti vicini tra di loro.

Sulla base dello studio effettuato, gli algoritmi analizzati sono stati applicati nel contesto della malware analysis, con lo scopo di individuare e classificare binari malevoli.

A tale scopo è stato creato un dataset contenete 1000 sample suddivisi equamente tra malware e goodware. I malware presi in considerazione appartengono a 16 famiglie diverse, raggruppate in tre macro categorie (Banking Trojan, Cryptojacking e Ransomware).

I test condotti sono stati effettuati con l'obiettivo di capire quali sono gli algoritmi più precisi in termini di percentuale di corretta classificazione e allo stesso tempo quelli più efficienti e scalabili. I risultati mostrano che gli approcci presi in considerazione, sono in grado di classificare un gran numero di sample, attribuendogli la famiglia malware corretta. In particolare, il livello di precisione varia dal 56% al 83% in base alla tipologia di algoritmo utilizzato.

# 2 Introduzione

Con il rapido sviluppo dei sistemi informatici e della digitalizzazione dei documenti, i ***malware*** rappresentano una delle principali minacce informatiche al giorno d'oggi. Qualsiasi software in grado di eseguire azioni dannose (tra cui information stealing, spionaggio, blocco dei sistemi, ecc) può essere etichettato con il termine generico di malware. Kaspersky Labs definisce il malware come *«un programma per computer progettato per infettare i sistemi informatici con lo scopo di arrecare danno su di esso o agli utenti attraverso molteplici modalità.»*

Essi infatti rappresentano una componente estremamente importante in diversi attacchi informatici finalizzati al furto di informazioni, all'estorsione o al danneggiamento di infrastrutture telematiche.

Un caso particolarmente eclatante è rappresentato dal ransomware ***WannaCry*** che nel maggio del 2017 ha fatto la sua prima apparizione, attaccando numerosi sistemi in tutto il mondo. Colpendo 300.000 computer in oltre 150 paesi, con un danno stimato pari a 8 miliardi di dollari.

Oltre ai guadagni generati dalle estorsioni verso le aziende, i malware vengono spesso utilizzati nello spionaggio informatico nazionale. In questi casi sono utilizzati per attacchi mirati, orientati ad ottenere informazioni economiche, politiche o militari.

Infine, i malware possono essere utilizzati come cyber armi in una guerra cibernetica allo scopo di mettere in ginocchio le infrastrutture telematiche della nazione nemica. Questa eventualità si è concretizzata nel 2017, quando un gruppo di hacker noto con il nome di ***ShadowBrokers***, è entrata in possesso degli oltre ottomila file che compongono l'archivio ***Vault7*** della CIA. L'archivio conteneva diversi zero-day tra cui il famoso EternalBlue utilizzato da WannaCry per diffondersi in rete e strumenti in grado di penetrare, infestare e controllare svariati sistemi operativi per mobile (Sia Android che iPhone) e pc (Mac, Windows e Linux).

In questo contesto, difendere gli host e le reti contro queste minacce, continua ad essere una priorità assoluta. Tuttavia, l'evoluzione continua dei malware costringe i fornitori di antivirus (AV) a cercare soluzioni sempre più sofisticate per difendere i propri clienti.

Seguendo questa direzione, i ricercatori hanno spostato la loro attenzione dai tradizionali metodi di rilevamento basati su signature, a sistemi dinamici che raccolgono tracce sul comportamento del malware.

Tra le nuove tecnologie di rilevazione sulle quali i fornitori di antivirus stanno puntando vi sono quelle basate su tecniche di ***machine learning***.

L'utilizzo del machine learning non è una novità nell'ambito della cyber security. Infatti, viene già implementato in diversi motori di correlazione chiamati ***UBA (User Behavior Analytics)*** che permettono di rilevare minacce interne, attacchi mirati e frodi finanziarie. Le soluzioni UBA esaminano i modelli di comportamento umano e quindi applicano algoritmi e analisi statistiche per rilevare anomalie significative da tali modelli che potrebbero indicare potenziali minacce.

Nell'ambito dei malware, il machine learning può essere utilizzato per rilevare nuove minacce che non sono ancora riconosciuti dai sistemi tradizionali. Un'altra possibilità è quella di supportare l'analista durante la fase di studio del malware, al fine di capire rapidamente se tale file può essere ricondotto ad una famiglia di malware già nota.

Gli studi fatti per individuare motori di machine learning utilizzati per l'individuazione e la classificazione di malware si basano sugli algoritmi di **Binary Code Similarity Detection (BCSD)**. Essi hanno lo scopo di determinare il grado di similarità tra due file binari. Tale problema può essere ricondotto nel determinare, a partire dal codice compilato, se le funzioni che costituiscono i due file sono simili tra loro e in che percentuale.

Nello specifico questi algoritmi basano il loro successo nella creazioni di dataset contenenti delle caratteristiche peculiari delle funzioni che costituiscono i binari. Quindi, durante la fase di pre-elaborazione i binari vengono disassemblati e rielaborati per estrapolare le funzioni contenute al loro interno. Successivamente per ogni funzione vengono effettuate delle manipolazioni intermedie, ad esempio calcolo degli hash o analisi sulla struttura semantica e sintattica della funzione.

Solo dopo aver raccolto queste informazioni è possibile ottenere dei vettori n-dimensionali (la dimensione dipende dal numero di caratteristiche che si vogliono prendere in considerazione) su cui applicare algoritmi di machine learning. Tali vettori n-dimensionali prendono il nome di embedding e sono alla base delle moderne tecniche di BCSD.

L'utilizzo di tecniche di BCSD pone una serie di sfide. Il problema principale nell'individuare le somiglianze tra due file binari, è che lo stesso codice sorgente può essere compilato usando diversi compilatori, diversi livelli di ottimizzazione o utilizzare architetture diverse. In questo modo, è possibile che a partire dallo stesso codice sorgente verranno generati codici binari sintatticamente differenti.

Il lavoro svolto, va ad esplorare lo stato dell'arte di queste soluzioni, confrontandoli sulla base della percentuale di corretta classificazione, efficienza e scalabilità degli algoritmi.

# 3 Background

## 3.1 Binary Code Analysis

I file binari eseguibili contengono al loro interno innumerevoli informazioni sul loro comportamento durante l'esecuzione. L'analisi binaria consiste nell'analizzare i programmi a livello di codice macchina, al fine di studiarne il loro funzionamento. Idealmente, l'analisi binaria produce informazioni sul codice (istruzioni, basic block, funzioni e moduli), sull'organizzazione (controllo e flusso di dati) e sulla strutture dati (variabili globali e stack) del programma.

I due principali approcci nell'analisi dei file binari sono: ***l'analisi statica*** e ***l'analisi dinamica***. Nel primo caso, l'analisi avviene senza eseguire il programma da analizzare e prevede come primo passo, la lettura e l'interpretazione del formato binario (ELF, PE, Mach-O, ecc). Ciò fornisce informazioni sulla posizione delle varie sezioni del binario. Una volta individuata la sezione di testo, è possibile procedere con la fase di *disassembling* del codice che trasforma il linguaggio macchina del file binario in codice assembly. Questa fase non è semplice perché alcune architetture hanno istruzione di lunghezza variabile che rendono complicato il processo. Inoltre, il codice assembly non prevede una suddivisione netta tra parte codice e parti dati, il che significa che le istruzioni e i dati possono essere mescolati insieme, complicando ulteriormente il processo.

Al contrario, l'analisi dinamica richiede l'esecuzione del programma per ottenere una traccia di esecuzione. Da questo tipo di analisi è possibile recuperare molte informazioni (ad esempio il valore di ciascun registro in ogni fase dell'esecuzione del programma), ma è limitato al flusso seguito durante l'esecuzione del programma.

Le informazioni raccolte dalla Binary Code Analysis sono alla base di tutte le tecniche di ***Binary Code Similarity Detection*** e ***di Binary Diffing***. La prima tipologia di tecniche, cerca di individuare le componenti simili di due binari, mentre la seconda tipologia individua la similarità dei binari basandosi sulle differenze.

## 3.2 Binary Code Similarity Detection

Dati due file binari, il problema di valutare quanto questi file siano simili prende il nome di ***Binary Code Similarity Detection (BCSD)***.

Le tecniche alla base del BCSD sono presenti in diversi campi dell'ingegneria del software e della sicurezza informatica. In particolare, è applicata per individuare codice plagiato, classificare malware, ed individuare vulnerabilità o bachi del software.

Le soluzioni BCSD allo stato dell'arte si basano su quattro approcci:

- ***confronto dei byte***;
- ***confronto delle istruzioni assembly***;
- ***confronto dei Control Flow Graph***;
- ***confronto di attributi strutturali***.

Il primo approccio prevedere il confronto dei byte presenti all'interno del binario [1], senza applicare alcuna trasformazione intermedia. In Fig.3.1, si riporta un esempio dell'utilizzo di tale approccio.


Figura 3.1. Approccio basato sul confronto dei byte contenuti nei file.

Nello specifico è stato utilizzato *Synalyze*[1], un editor di testo esadecimale che permette di evidenziare, attraverso l'uso di colori, i byte uguali contenuti nei due binari. Tale approccio ha la limitazione di non tenere conto della logica complessiva del programma.

[1]Synalyze: https://www.synalysis.net/additional-features.html

Il secondo approccio, invece, prevede di effettuare il confronto sulla base del dissambly ottenuto durante la fase di disassembling. In cui il codice macchina presente all'interno del binario viene convertito in codice Assembly. In questo approccio spesso vengono applicate una o più fasi di normalizzazione del codice prima di effettuare il confronto [2].

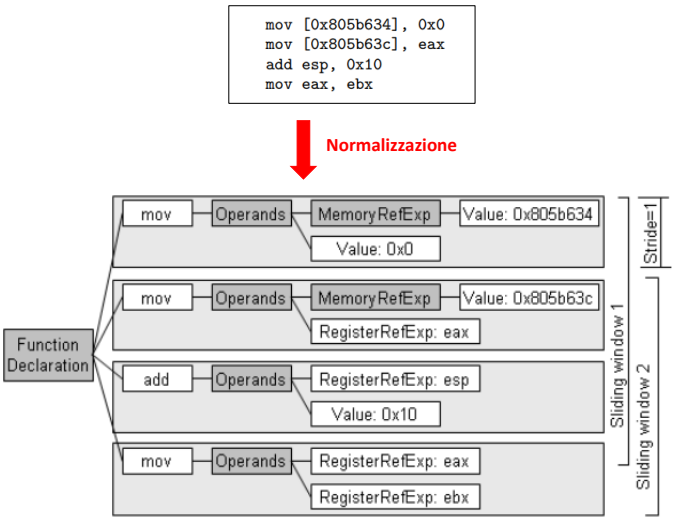


Figura 3.2. Approccio basato sul confronto delle istruzioni assembly.

In Fig.3.2, si riporta il processo di normalizzazione, attraverso rappresentazioni intermedie, di un frammento di codice assembly. In tale rappresentazione, ogni istruzione viene partizionata in *operazioni* (ad esempio mov, add, ecc) e *operandi*. Gli operandi vengono suddivisi ulteriormente in operandi di *registro*, *memoria* e *costanti*. Per ogni operando viene applicata una funzione di generalizzazione, che trasforma ogni registro in un registro generalizzato (ad esempio eax vine trasformato in RegGen). In modo analogo, si effettua una trasformazione per gli indirizzi in memoria e per i valori costanti.

Una volta che il codice assembly è stato normalizzato, vengono applicati algoritmi di *Exact Clone Detection* e *Inexact Clone Detection*. I primi sfruttano, la corrispondenza esatta tra gli hash calcolati sulle istruzioni assembly normalizzate. Viceversa, gli algoritmi di *Inexact Clone Detection* sfruttano gli algoritmi di LSH per stabile la somiglianza tra due istruzioni normalizzate.

La combinazione dei due algoritmi permette di individuare blocchi di codice simili. Tuttavia, anche questo approccio, come il primo, non tiene conto della logica complessiva del programma.

Il terzo approccio si basa sull'utilizzo di una caratteristica sintattica del codice binario chiamata ***Control Flow Graph (CFG)*** [3]. La problematica del BCSD, in questo modo, si riduce all'individuazione dei CFG simili contenuti all'interno di un binario. In questo ambito, molti algoritmi basano le proprie soluzioni sulla teoria dell'isomorfismo tra grafi.

In Fig. 3.3, si riportano le strutture di due CFG estratti da due binari, che evidenziano una somiglianza nella struttura a blocchi: numero di blocchi e connessioni. Tuttavia, essi presentano una complessità polinomiale, pertanto richiedono tempi di esecuzione elevati.

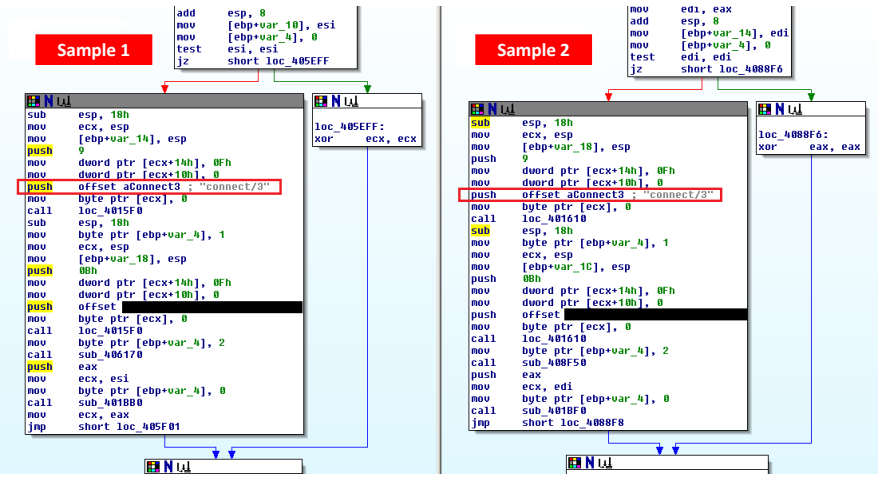


Figura 3.3. Approccio basato sui CFG .

Per tale ragione, il quarto approccio si basa sull'estrazione di attributi strutturali. Attributi numerici che possono essere estratti dai CFG o dal disassembly generato. In questo caso si parla di ***Attributed Control Flow Graph (ACFG)***.

La Fig. 3.5 mostra come a partire dal CFG di una funzione è possibile ottenere il ACFG corrispondente.

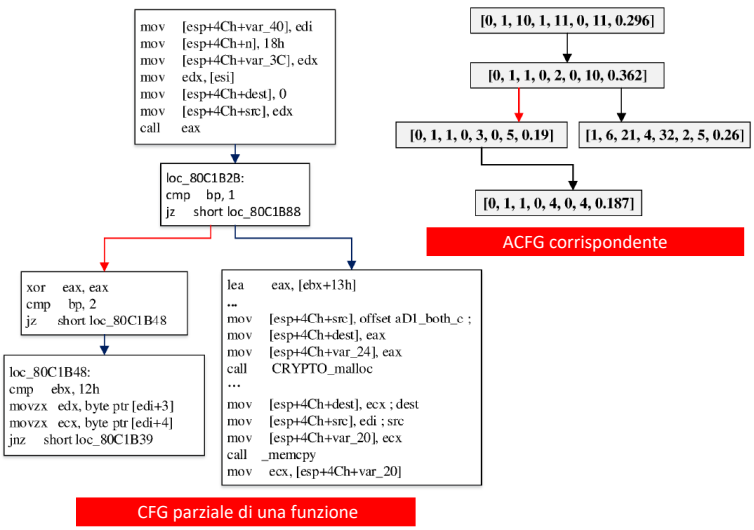


Figura 3.4. Approccio basato sugli ACFG .

La scelta degli attributi estratti (detti anche features) dagli ACFG derivano dalla conoscenza di esperti, che talvolta potrebbero influenzare significativamente sui risultati del BCSD.

L'applicazione di tecniche di BCSD pone una serie di sfide. In primo luogo, diverse ottimizzazioni del compilatore [4] producono binari diversi. Ad esempio il compilatore potrebbe decidere di applicare la tecnica di ottimizzazione chiamata *costant folding*, in cui le costanti vengono valutate e calcolate durante la compilazione. Oppure se il codice contiene dei cicli, il compilatore potrebbe decidere di ripetere sequenzialmente le istruzioni presenti nel ciclo senza effettuare salti, utilizzando la tecnica di ottimizzazione *loop unrolling*.

Anche compilare lo stesso codice con compilatori diversi genera binari differenti; così come il codice compilato su piattaforme diverse produce binari differenti. Tutti i binari così ottenuti, sono tra di loro semanticamente identici ma sintatticamente presentano delle differenze.

Un'altra problematica è legata all'efficienza di tali algoritmi. Infatti motori di BCSD veloci sono fondamentali per gli analisti, al fine di individuare con rapidità ed efficacia le parti di codice clonato.

La scalabiltà è un altro fattore da tenere in considerazione, in quanto le funzioni assembly presenti all'interno di un repository devono scalare fino a milioni. In questo caso, è necessario considerare anche il degrado delle prestazioni di ricerca a fronte di un aumento del numero di funzioni da analizzare.

Altra caratteristica fondamentale di tali algoritmi è quella di accettare degli aggiornamenti incrementali sul repository delle funzioni, senza reindicizzare le funzioni già presenti.

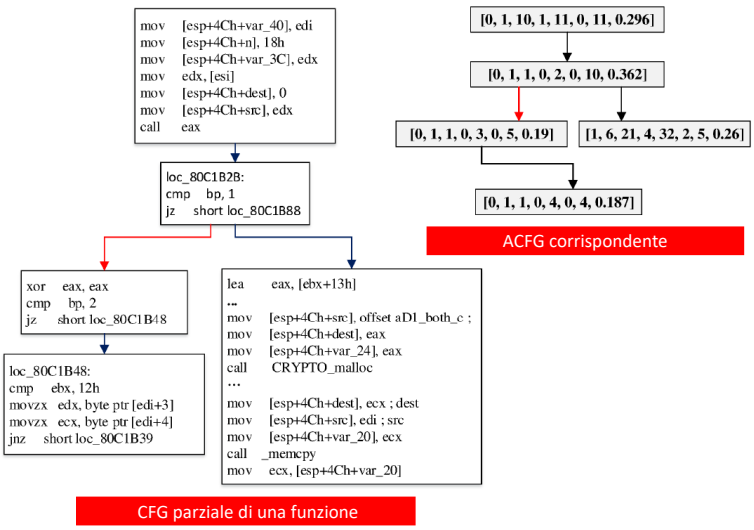


Figura 3.5. Approccio basato sugli ACFG .

## 3.3 Binary Diffing

Un'altra tecnica utilizzata nell'analisi dei binari prende il nome di ***Binary Diffing***. In contrapposizione alla BCSD, che lavora sulla similarità, la Binary Diffing si concentra sulle differenze tra file binari.

Tali tecniche vengono applicate principalmente nell'individuazione di vulnerabilità o nello studio di *0-day*. In questo caso è più corretto parlare di ***Patch Diffing***: in cui si confrontano due binari, il primo contenente una vulnerabile e il secondo contenente la rispettiva correzione di sicurezza (o patch). In questo modo è possibile determinare i dettagli tecnici alla base di bollettini di sicurezza o per stabilire le cause principali, i vettori di attacco e le potenziali varianti delle vulnerabilità in questione.

Analogamente, il binary diffing può essere utilizzato per scoprire discrepanze tra due o più versioni di un prodotto, se condividono lo stesso codice core e coesistono sul mercato, ma sono servite indipendentemente dal fornitore. Un esempio di tale software è il sistema operativo Windows, che attualmente ha tre versioni in supporto attivo: Windows 7, 8 e 10. Mentre Windows 7 ha ancora una quota quasi del 50% sul mercato desktop, Microsoft è nota per aver introdotto una serie di miglioramenti strutturali alla sicurezza e talvolta anche correzioni ordinarie solo alla più recente piattaforma Windows. Ciò crea un falso senso di sicurezza per gli utenti dei sistemi più vecchi e li rende vulnerabili a difetti del software che possono essere rilevati semplicemente individuando piccole modifiche nel codice [5].

Le tecniche di Binary Diffing si basano su ***Symbolic Names Matching*** o ***Fingerprint Hash Map***. La prima tecnica basa la ricerca di differenze utilizzando la tabella dei simboli presente all'interno di ogni intestazione dei file binari. La tabella dei simboli viene creata in fase di compilazione e contiene tutti i nomi simbolici presenti all'interno di un programma (ad esempio nomi di funzione, variabili, ecc) e gli indirizzi di memoria puntati (assoluti o rilocabili).

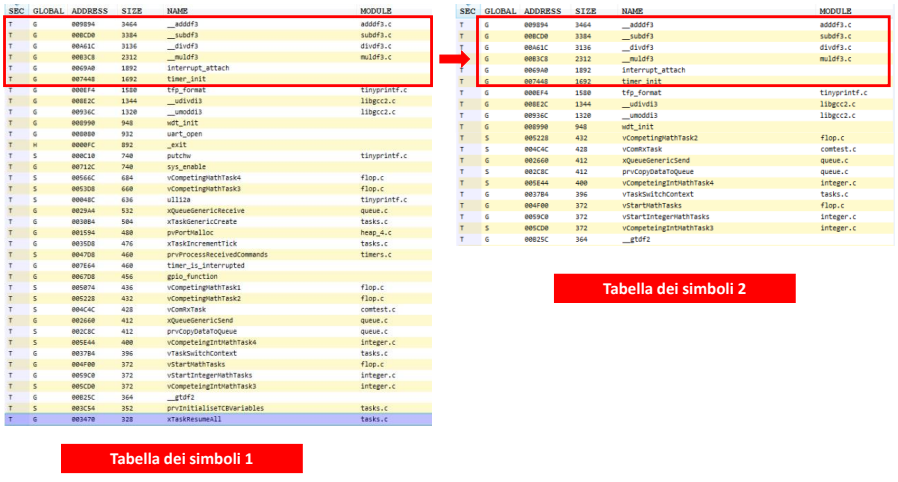


Figura 3.6. Approccio basato sulle tabelle dei simboli.

In Fig. 3.6 viene proposto l'approccio utilizzato attraverso la tabella dei simboli. Sostanzialmente consiste nell'identificare l'esatto match dei nomi simbolici.

Il metodo ***Fingerprint Hash Map*** consiste nell'associare ad una sequenza di istruzioni (definita come basic block) un hash definito *fingerprint*. La dimensione del basic block può essere variabile in base al frammento di codice da analizzare. Una volta calcolati tutti i fingerprint dei basic block, vengono riportati all'interno di una tabella di Hash. In cui la chiave è il fingerprint e il valore è il basic block corrispondente. Al termine dell'operazione si avranno due tabelle di Hash da confrontare per ottenere i basic block differenti.

## 3.4 Disassembler

Un Disassembler è un programma che permette di effettuare una conversione da linguaggio macchina a linguaggio assembly (o una rappresentazione intermedia). Ciò lo rende differente dai decompilatori che vengono utilizzati per effettuare la traduzione da un linguaggio ad alto livello a linguaggio di basso livello. L'output ottenuto dal disassembler prende il nome di ***disassembly*** e può essere ottenuto da due tipologie di tecniche di disassembling: ***dinamico*** e ***statico***.

I disassembler che utilizzano tecniche dinamiche eseguono il codice binario in emulatori o simulatori come QEMU[3], registrando i passi di esecuzione di un determinato input. Poiché le tecniche dinamiche registrano solo un percorso di esecuzione determinato dagli input in ingresso, il disassembly ottenuto sarà una rappresentazione parziale del funzionamento del programma.

I disassembler che utilizzano tecniche statiche tentano di ottenere il disassembly senza eseguire il codice binario. In generale, esistono tre approcci statici per disassemblare un binario.

Il primo approccio, chiamato ***scansione lineare***, inizia dal punto di ingresso del codice binario e disassembla istruzioni fino a quando non raggiunge la fine (o istruzioni illegali). Tuttavia, se i dati sono mischiati alla parte codice, questo approccio potrebbe produrre sequenze di codice errate.

Il secondo approccio, chiamato ***attraversamento ricorsivo***, disassembla le istruzioni basandosi sul CFG del binario. Questo permette di superare i limiti del primo approccio a patto che il disassembler sia in grado di ricostruire in maniera corretta il CFG. La costruzione di un CFG corretto spesso risulta difficoltoso quando nel codice sono presenti chiamate indirette che dipendono dal contenuto dei registri o posizioni di memoria.

L'ultimo approccio combina la scansione lineare all'attraversamento ricorsivo, combinando i vantaggi di entrambi.

Esistono diversi disassemblatori (open source e commerciali) che possono essere utilizzati. Il disassemblatore più popolare è probabilmente il prodotto commerciale IDA Pro [12], ampiamente utilizzato dai programmatori e dagli analisti di sicurezza per il reverse enginering. IDA Pro è un disassembler basato su tecniche di attraversamento ricorsivo. Oltre al codice assembly, IDA Pro può generare il FCG di tutto il programma, ricostruendo tutte le funzioni presenti al suo interno. Per fare ciò, si appoggia su un algoritmo chiamato ***Fast Library Indication and Recognition Technology (FLIRT)***.

Una volta effettuato il disassembling del binario, IDA Pro visualizza la seguente interfaccia Fig. 3.7, in cui si evidenziano le tre sezioni principali del programma.

[3]QEMU (abbreviazione di Quick Emulator) è un emulatore gratuito e open source che esegue la virtualizzazione dell'hardware. Permette di emulare il processore della macchina fornendo una serie di diversi modelli hardware e di dispositivi. Ciò consente di eseguire un gran numero di sistemi operativi guest.

La prima sezione visualizza il codice assembly ottenuto in rappresentazione CFG. Tale rappresentazione può essere cambiata per ottenere il listato assembly completo.

La seconda sezione elenca la lista delle funzioni ottenuto a cui è stato assegnato un nome simbolico durante la fase di disassembling. E' possibile rinominare tale funzioni andando a modificarli in automatico in tutto il programma.

La terza sezione mette a disposizione una linea di comando per interfacciarsi con l'interprete Python presente all'interno del software.

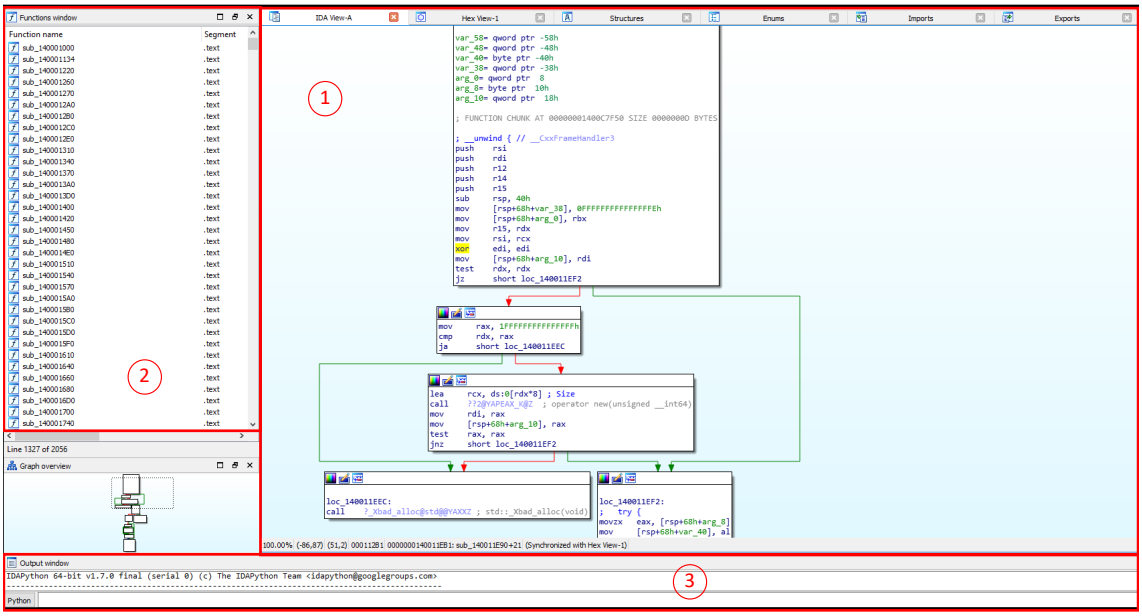


Figura 3.7. Interfaccia Principale di IDA Pro.

Un altro disassembler molto utilizzato è Radare (attualmente alla versione 2). Esso è un framework open source costituito da un gran numero di tool a riga di comando che possono essere utilizzati insieme o in maniera indipendente. Di seguito si riporta la lista dei tool principali presenti al suo interno:

- ***radare2 (r2)***: lo strumento principale dell'intero framework. radare2 consente di aprire un numero di sorgenti di input/output come se fossero semplici file (ad esempio dischi, connessioni di rete, driver del kernel, processi in fase di debug, ecc). Implementa un'interfaccia a riga di comando avanzata per spostarsi all'interno dei file, analizzare dati, disassemblare, applicare patch binarie, confrontare dati, cercare, sostituire e visualizzare. Può essere integrato con molti linguaggi di programmazione, inclusi Python, Ruby, JavaScript, Lua e Perl.

- ***rabin2***: un programma per estrarre informazioni dai binari eseguibili, come ELF, PE, Java CLASS o Mach-O. rabin2 viene utilizzato dal framework per ottenere informazioni dall'eseguibile (ad esempio simboli esportati, importazioni, riferimenti incrociati (xrif), dipendenze di librerie e sezioni).

- ***rasm2***: un assemblatore e disassemblatore da riga di comando per più architetture (inclusi Intel x86 e x86-64, MIPS, ARM, PowerPC, ecc).

- ***rahash2***: uno strumento di hash basato su blocchi. Dalle stringhe di testo di piccole dimensioni ai dischi di grandi dimensioni, rahash2 supporta diversi algoritmi, tra cui MD4, MD5, CRC16, CRC32, SHA1, SHA256 e altri. rahash2 può essere utilizzato per verificare l'integrità o tenere traccia delle modifiche sui file di grandi dimensioni (dump di memoria o dischi).

- ***radiff2***: un applicazione di utilità per trovare le differenze a livello di byte all'interno dei file binari.

A supporto del framework Radare, duranti gli anni, sono state sviluppate molte interfacce grafiche. In Fig.3.8, viene mostrata la GUI più utilizzata per Radare2 che prende il nome di *Cutter*. L'utilizzo di tale GUI, rende Radare esteticamente molto simile ad IDA Pro.

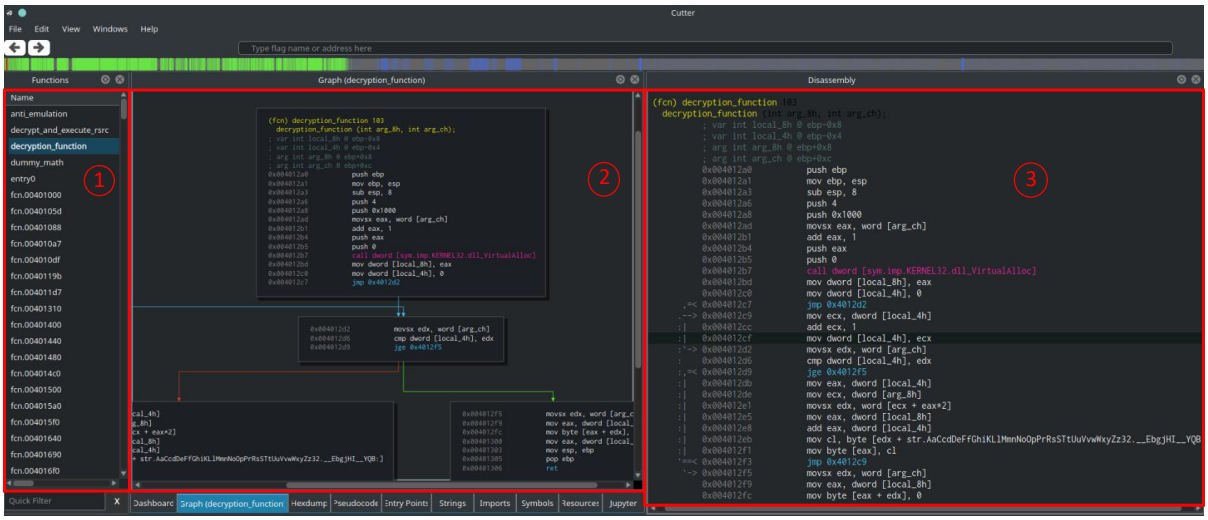


Figura 3.8. Interfaccia grafica Cutter per Radare.

Un altro disassembler di recente creazione è Xori [13], un framework per il reverse engeeniring che permette di automatizzare l'analisi statica dei binari. Esso utilizza oltre alle classiche tecniche di analisi statica del codice anche tecniche di analisi dinamica. Tale approccio permette di emulare i file binari in un ambiente sicuro e controllato.

Il framework sviluppato in Rust, permette di emulare stack, stati dei registri e tabelle dei simboli, al fine di identificare funzionalità sospette all'interno dell'eseguibile analizzato. L'architettura del framework Xori è costituita da quattro componenti:

- ***PE Loader***: gestisce il caricamento del PE file in memoria ,importa il TEB, inizializza gli offset delle DLL e le tabelle degli import.
- ***Memory Manager***: questa struttura contiene tutta le funzionalità necessarie per gestire e manipolare il file caricato in memoria. Inoltre imposta i flag necessarie per evitare accessi a zone di sola lettura della memoria.
- ***Analysis***: il cuore principale dell'applicazione, che effettua il disassembling, l'estrazione delle funzioni e dei riferimenti esterni.
- ***State***: questa struttura permette di emulare lo stato della CPU, dei registri e dello stack durante la fase di esecuzione del binario in ambiente controllato.

Come risultato delle analisi, Xori è in grado di generare un file json contenente tutte le informazioni relative alla struttura del file PE, del FCG complessivo e del CFG delle funzioni estratte, ottenute durante l'analisi del file. Inoltre, oltre all'interazione da riga di comando, è possibile installare una GUI Web-Based, sviluppata in NodeJs, che consente la navigazione all'interno del codice assembly.

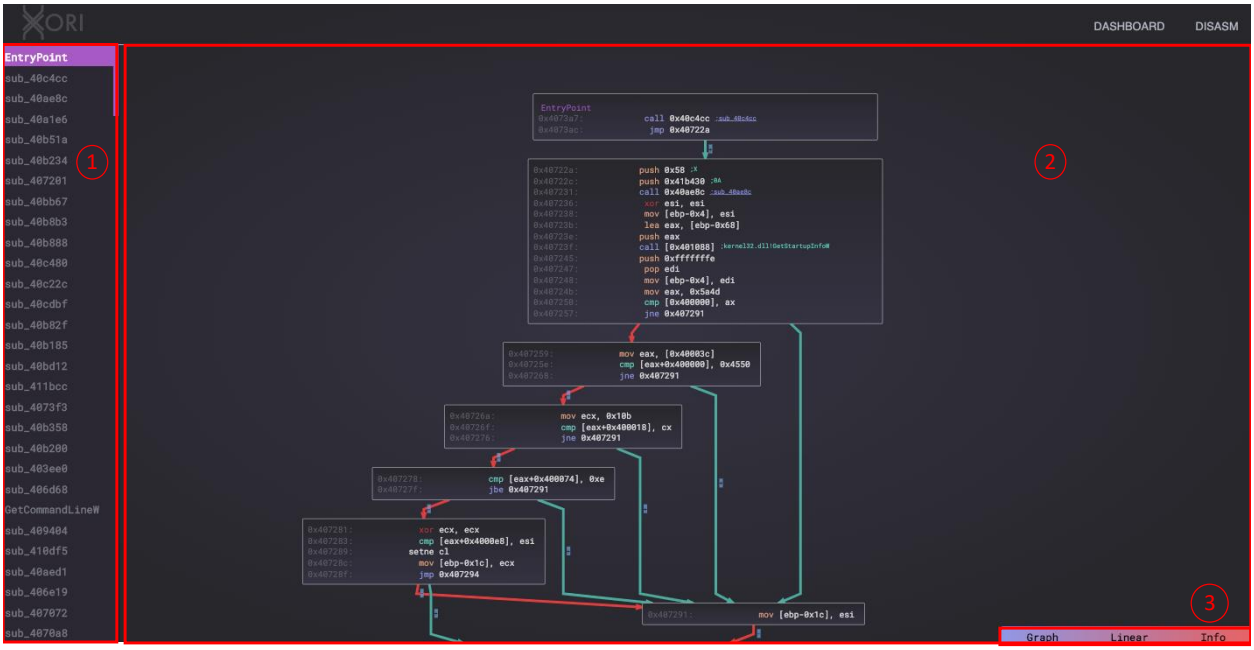


Figura 3.9. Interfaccia grafica di Xori.

La GUI, come mostrato in Fig. 3.9, è suddivisa in tre sezioni principali. La prima sezioni contiene la lista di tutte le funzioni estratte a cui è stato assegnato un nome simbolico da Xori. E' possibile selezionare una determinare una determinata funzione per visualizzare nella sezione 2 il contenuto attraverso un rappresentazione CFG. Nella sezione 3 è possibile cambiare il tipo di visualizzazione, scegliendo tra:

- ***Graph***: visualizzazione con rappresentazione in FCG della funzione;

- ***Linear***: visualizzazione delle istruzioni assembly in maniera sequenziale;
- ***Info***: visualizzazione delle caratteristiche principali del PE file (ad esempio stringhe, funzioni importate, ecc).

## 3.5 Function Call Graph e Control Flow Graph

Il ***grafo delle chiamati a funzioni*** (detto anche ***Function Call Graph*** o ***FCG***) è una rappresentazione, tramite diagramma di flusso, che permette di descrivere le relazioni di *chiamante* e *chiamato* tra le funzioni di un programma. In particolare, il FCG è basato su un grafo orientato etichettato i cui nodi rappresentano le funzioni del programma e gli archi le relazioni tra le funzioni.

Il FCG è il risultato dell'analisi di un programma al fine di capirne il funzionamento o come base per ulteriori analisi. Tale grafo può essere ricavato attraverso due tecniche: ***dinamica*** e ***statica***. Un FCG dinamico viene ricavato dall'esecuzione del programma, ma permette di avere un grafo limitato al flusso seguito durante l'esecuzione. Per ottenere un grafo completo attraverso le tecniche dinamiche è necessario eseguire più volte il programma (tante volte quanto il numero dei possibili flussi). Un FCG statico è un grafo che rappresenta ogni possibile esecuzione del programma. Inoltre, con tale tecnica vengono rappresentate anche funzioni che durante l'esecuzione non verrebbero mai eseguite a causa di errori di programmazione.

Inoltre, i FCG possono essere suddivisi in ***context sensitive*** o ***context insensitive***. I call graph di tipo context sensitive aggiungono un nodo aggiuntivo per ogni configurazione di parametri che viene utilizzata per chiamare tale funzione. I context insensitivi call graph utilizzano un unico nodo per rappresentare la funzione chiamata, senza distinguere i parametri passati alla funzione.

Tipicamente, i FCG estratti dai programmi presentano una struttura ad albero. In cui esiste un nodo radice che rappresenta l'entry point del programma. L'insieme dei nodi che rappresenta le funzioni utilizzate dal programma possono essere suddivise in tipologie diverse: funzioni locali (definite dall'utente) e funzioni non locali o esterne (definite all'interno di librerie terze). Tale differenza è essenziale perché il compilatore può modificare i nomi delle funzioni locali all'interno del binario, ma i nomi delle funzioni esterne rimangono gli stessi.

Poiché esiste una corrispondenza uno a uno tra le funzioni del programma e i nodi del FCG, tutti i nodi possono essere etichettati in modo univoco in base al nome della funzione che rappresenta.

**Definizione 1** *(Function Call Graph). Il grafo delle chiamate a funzioni di un programma è un grafo orientato etichettato definito dalla quintupla G = (V(G); E(G); Z; L; T). Dove V(G) è l'insieme dei vertici, ognuno dei quali rappresenta una singola funzione del programma. E(G) è un insieme di archi orientati dove (u, v) ∈ E(G) indica che esiste una funzione u che al suo interno effettua una chiamata alla funzione v. Z è un insieme di possibili etichette che indica il nome delle funzioni del programma. L : V(G) ↦ Z è una funzione di etichettatura che assegna a ciascun*

*nodo un'etichetta univoca. T: V(G) ↦ L, N è una funzione che assegna ad ogni nodo la caratteristica di essere una funzione locale (L) o non locale (N). Inoltre, è possibile indicare l'insieme delle funzioni locali del FCG con* $V_L(G) = \{ v \mid v \in V(G) : T(v) = L \}$ *e in maniera analoga è possibile indicare l'insieme delle funzioni non locali* $V_N(G) = \{ v \mid v \in V(G) : T(v) = N \}$.

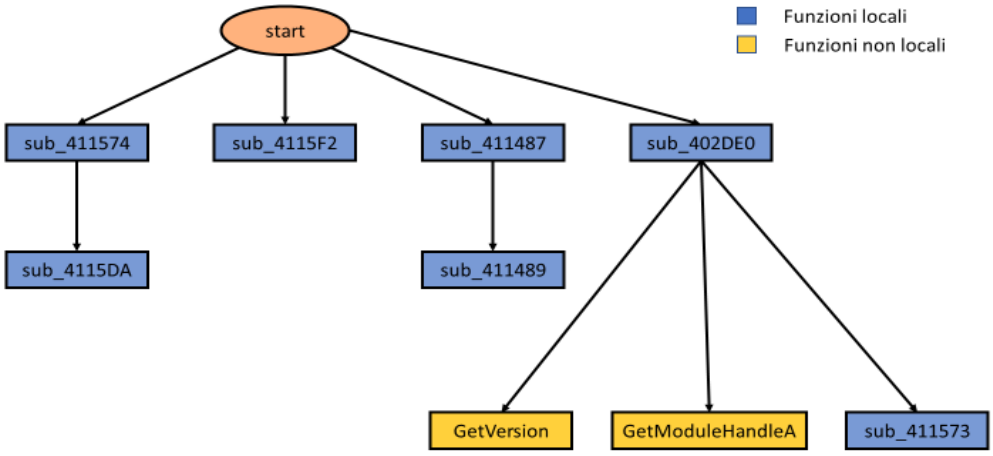


Figura 3.10. Esempio di Function Call Graph, con distinzione tra funzioni locali e non locali.

In alcuni casi può essere utile effettuare una distinzione ulteriore all'interno delle funzioni non locali; evidenziando la differenza tra funzioni di libreria a caricamento dinamico (DLL) e funzioni collegate staticamente (Statically-linked). Il codice delle funzioni a caricamento dinamico viene aggiunto al programma durante la fase di caricamento in memoria e quindi a runtime. Mentre le funzioni statiche sono aggiunte durante la fase di compilazione del programma.

I FCG ottenuti attraverso l'analisi statica possono contenere anche altre informazioni rilevanti estratte dal binario. In generale, è possibile estrarre il codice operativo che costituisce ogni funzione locale del binario. Queste sequenze di codice possono essere utilizzate per creare diagrammi di flusso delle funzioni locali (***Control Flow Graph*** o ***CFG***).

Le sequenze di istruzioni assembly consecutive che non effettuano diramazioni nel normale flusso di esecuzione prendono il nome di *basic block*. Il diagramma di flusso di una funzione è costituito da una serie di basic block interconnessi tra di loro che evidenziano tutti i possibili percorsi di una funzione locale.

**Definizione 2** *(Control Flow Graph). Il control flow graph di un programma è un grafo orientato etichettato definito dalla quintupla G = (V(G); E(G)). Dove V(G) è l'insieme dei vertici, ognuno dei quali rappresenta un basic block del programma. E(G) è un insieme di archi orientati dove (u, v) ∈ E(G) indica che esiste un flusso di esecuzione dal basic block u al basic block v.*

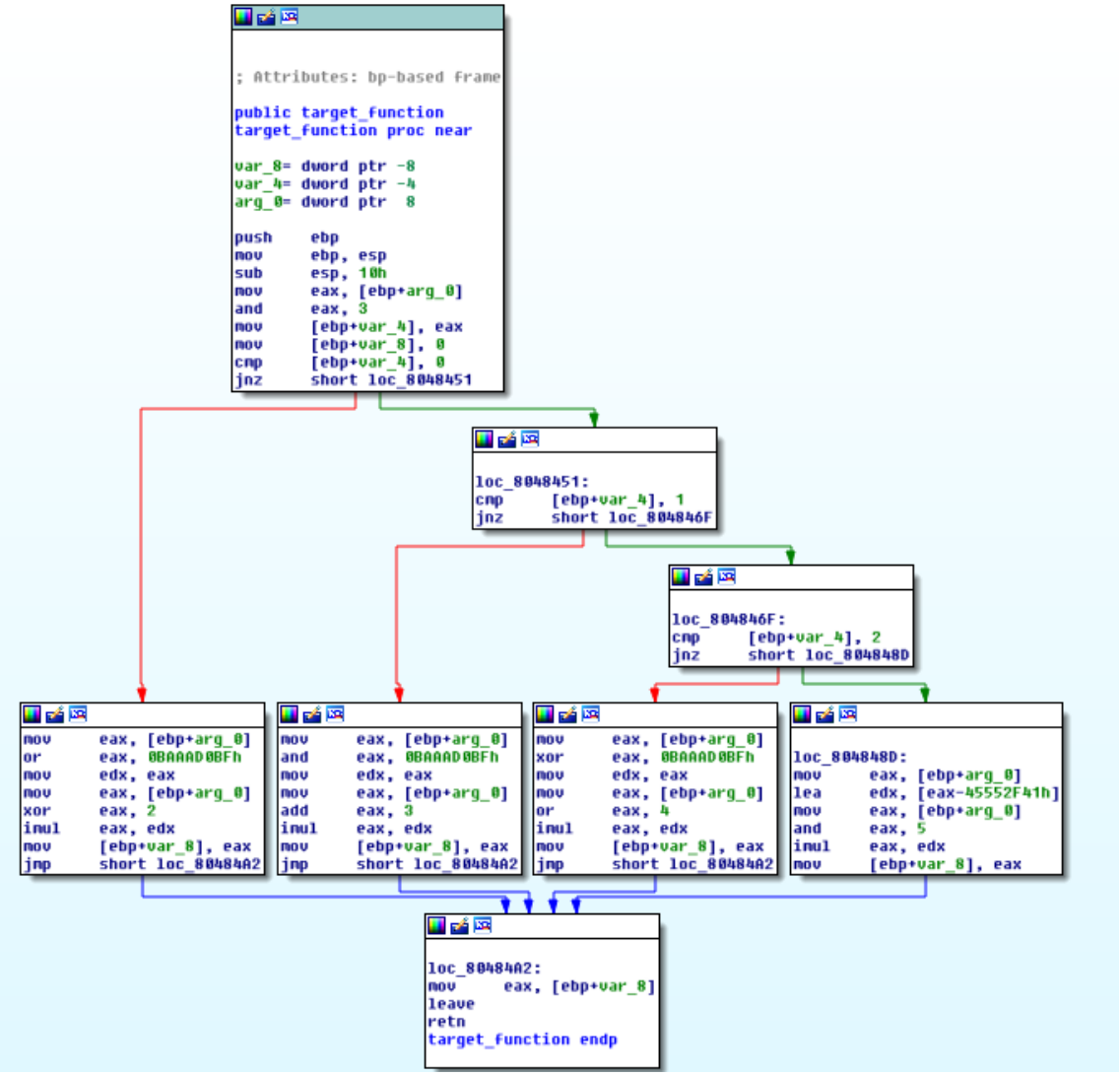


Figura 3.11. Esempio di Control Flow Graph, estratto attraverso il decompilatore IDA Pro.

I FCG e i CFG sono generati ed estratti attraverso tool chiamati disassemblatori. In primo luogo, bisogna capire se al binario sono state applicate tecniche di ***binary obfuscation***[1] o ***binary packing***[2].

Una volta individuata la tecnica di offuscamento utilizzata, è necessaria rimuoverla per poter analizzare il binario con il disassemblatore. Il disassemblatore viene utilizzato per identificare le funzioni ed assegna un nome simbolico univoco. Ciò è necessario perché i nomi delle funzioni scritte dal programmatore non sono conservati durante la compilazione del software. Quindi vengono assegnati dal decompilatore dei nomi casuali univoci.

Per le funzioni esterne, invece, vengono mantenuti i nomi contenuti all'interno delle librerie. Nel caso di funzioni esterne importate dinamicamente, si può ottenere il suo nome dall' *Import Address Table (IAT)* presente nell'header PE dell'eseguibile. Mentre per le funzioni esterne collegate staticamente esistono dei software in grado di riconoscerle ed assegnare i nomi canonici corretti delle librerie standard.

Una volta che tutte le funzioni, cioè i nodi del function call graph, sono stati identificati, vengono inseriti all'interno di una struttura dati FIFO. Successivamente per ogni funzione presente, partendo dall'*entry point*, all'interno della struttura dati vengono analizzati tutte le relazioni chiamante-chiamato per identificare gli archi. Quando tutte le funzioni sono state processate l'algoritmo termina e il function call graph è creato.

Spesso creare un FCG completo da un binario non è banale. Infatti, l'utilizzo di tecniche di programmazione come la reflaction e i puntatori a funzione rendono l'estrazione delle funzioni più complessa. E sopratutto, non è possibile capire se tale funzione verrà richiamata (e da chi) durante la fase di esecuzione.

[1] La binary obfuscation permette di trasformare un programma sorgente in un programma destinazione, in modo date che entrambi i programmi abbiano lo stesso comportamento, ma il programma destinazione è molto più complicato da analizzare attraverso tecniche di reverse engineering. Tale tecniche vengono utilizzate dalle aziende per proteggere frammenti di codice sensibile del loro software o dai malware per rendersi irriconoscibili dagli antivirus basati su signature.

[2] Il binary packing è una tecnica di obfuscation che nasconde il codice reale del programma attraverso uno o più strati di compressione. Durante l'esecuzione la funzione di unpacking decomprime il codice originale in memoria e lo esegue.

## 3.6 Similarità

Uno dei problemi fondamentale del data mining è quello di esaminare i dati al fine di individuare oggetti "simili" all'interno dello stesso spazio metrico. Tale problema evidenzia una serie di sfide. In primo luogo, la presenza di numerose coppie di elementi rende dispendioso calcolare il grado di similarità di ogni coppia, anche se il calcolo della somiglianza di una coppia può essere molto semplice. Tale problema può essere risolto con due tipologie di approcci:

- algoritmi di ***Nearest Neighbor***
- algoritmi di ***Approximate Nearest Neighbor***.

Ciò rende necessario l'utilizzo di tecniche di locality-sensitive hashing (LSH), al fine di ridurre lo spazio vettoriale e focalizzare la ricerca alle sole coppie che hanno maggiori probabilità di essere simili.

## 3.7 Nearest Neighbor e Approximate Nearest Neighbor

L'algoritmo Nearest Neighbor (NN) è un algoritmo di machine learning ampiamente utilizzato dagli anni settanta per analizzare i dati in ambienti di apprendimento supervisionati.

**Definizione 3** *(Nearest Neighboor (NN)). Dato un insieme di punti* $x_1,\ldots,x_n \in R^d$ *e un punto da ricercare q, il problema del Nearest Neighbor cerca di trovare il punto* $x_i$ *che minimizza la distanza* $|| x_i - q ||$

Spesso, tale algoritmo viene indicato anche k-NN [14], in cui la "k" si riferisce al numero di vicini più prossimi utilizzati per classificare o predire i risultati in un set di dati. La classificazione o la previsione di ogni nuova osservazione viene calcolata in relazione a una distanza specificata in base a delle medie ponderate. L'algoritmo k-NN è una scelta obbligata quando c'è poca conoscenza della distribuzione dei dati che verranno analizzati.

L'algoritmo k-NN risulta fondamentale quando si affronta il problema di attribuire una specifica etichetta ad un set di dati che hanno delle caratteristiche simili tra di loro. Questa relazione di somiglianza è espressa come una metrica di distanza tra i punti del dataset. k-NN è un algoritmo non parametrico in quanto non fa assunzioni esplicite sulla forma funzionale della relazione. È anche un esempio di apprendimento basato sull'istanza, nel senso che si applica direttamente sui dati reali, piuttosto che su un modello di dati specifico.

La complessità dell'algoritmo è legata intrinsecamente alle dimensioni prese in considerazione. Per tale ragione, nei casi reali in cui gli spazi vettoriali presi in

considerazione hanno un gran numero di dimensioni è necessario utilizzare degli algoritmi di NN più efficienti rispetto alla semplice ricerca lineare.

Ciò ha generato interesse per tutti gli algoritmi che affrontano il problema applicando algoritmi di ricerca approssimati, chiamato Approximate Nearest Neighbor (ANN) [15][16].

**Definizione 4** *(Approximate Nearest Neighbor (ANN)). Dato un insieme di punti* $x_1,\ldots,x_n \in R^d$*, la metrica* $||\cdot||$ *e un punto da ricercare q, il problema del Approximate Nearest Neighbor cerca di trovare il punto* $x_i$ *tale che* $||\ x_i - q\ || \leq c \min_j ||\ x_j - q\ ||$

Il grado di affidabilità dell'approssimazione è misurata in termini di precisione, definendola come la percentuale dei punti di ricerca per i quali è stato trovato il vicino più prossimo.

Il problema del calcolo della prossimità tra due punti $x_i$ e $x_j$, può essere ricondotta a calcolare un indice di *similarità* o *dissimilarità*. L'indice di similarità (o prossimità) tra i due generici punti $x_i$ e $x_j$ è definito come funzione dei rispettivi vettori riga $y_i$ e $y_j$ (che costituiscono le componenti del punto preso in considerazione). Come indici di similarità tra punti vengono adottate le distanze.

**Definizione 5** *Si definisce distanza una funzione d:* $\Re^p \times \Re^p \rightarrow \Re$ *che gode delle seguenti proprietà:*

1. *non negatività:* $d(x,y) \geq 0$ *per ogni* $x,\ y \in \Re^p$*;*
2. *identità:* $d(x,y) = 0$ *se e solo se* $x \equiv y$*;*
3. *simmetrica:* $d(x,y) = d(y,x)$ *per ogni* $x,\ y \in \Re^p$*;*
4. *disuguaglianza triangolare:* $d(x,y) \leq d(x,z) + d(z,y)$ *per ogni* $x,y,z \in \Re^p$*.*

La coppia $(\Re^p, d)$ viene chiamato *spazio metrico*.

metriche: Gli intervalli tra le osservazioni sono misurati come distanza Euclidea, Manhattan, Chebyshev, Hamming o anche Coseno

## 3.8 Locality Sensitive Hashing

Le tecniche di locality-sensitive hashing (LSH), vengono utilizzate principalmente per ridurre le dimensioni dello spazio vettoriali su cui opera un insieme di dati. Ciò consente di rendere notevolmente più veloci gli algoritmi di ricerca dei vicini o il rilevamento dei dati duplicati.

L'idea chiave, che sta alla base delle tecniche di LSH, è quella di utilizzare diverse funzioni di hash in modo da garantire che, per ogni funzione, la probabilità di collisione sia molto più alta per gli oggetti vicini tra di loro rispetto a quelli distanti. Da questo punto di vista, LSH differisce dagli algoritmi di hashing tradizionali che

tentano di evitare il più possibile le collisioni per punti simili. Nel hashing tradizionale (detto anche crittografico) una piccola perturbazione dell'input può alterare in modo significativo l'hash, mentre in LSH, le piccole alterazione dell'input vengono ignorate permettendo di individuare punti simili tra loro.

**Definizione 6** *(Locality Sensitive Hashing Family). Data una distanza r in uno spazio vettoriale specifico, un grado di approssimazione $\xi$ e due probabilità $p_1 > p_2$, e una famiglia di funzioni di hash $H \rightarrow \{h{:}R^d \rightarrow U\}$ è (r,$\xi$r,p1,p2)-sensitive tale che:*

- *se $o \in B(q,r)$, allora $P_{r_H} |h(q)=h(o)| \geq p_1$*
- *se $o \notin B(q,r)$, allora $P_{r_H} |h(q)=h(o)| \leq p_1$*

Le famiglie di LSH possono essere applicati su molti spazi metrici come similarità cosinuisoidale [17], distanza di Hamming [18] e distanza di Jacard [19].

# 4 Analisi sullo stato dell'arte

## 4.1 Problema

Il problema, alla base dell'elaborato è quello di classificare in maniera corretta i file binari malevoli. Tale attività, viene svolta giornalmente dagli analisti di sicurezza che si occupano di malware analysis. Essi effettuano il reverse engeneering del malware al fine di estrarre informazioni utili sul comportamento del malware.

Dalle analisi dei malware vengono estratti degli ***indicatori di compromissione (IOC)*** che caratterizzano in maniera univoca quel singolo campione o la specifica infezione. Gli indici di compromissione permettono all'analista di verificare (attraverso l'analisi della rete e degli host) se l'infezione ha convolto il perimetro di analisi.

Quindi risulta di fondamentale importanza che tali IOC vengano estratti in maniera rapida e completa. Per tale ragione spesso l'analista di malware è supportato da strumenti chiamati *sandbox* che permettono di "detonare" in tutta sicurezza il malware, registrando le attività eseguite all'interno del sistema operativo (Ad esempio file creati, chiavi di registro modificate, connessioni di rete effettuate, ecc).

Tuttavia, in alcuni casi tali soluzioni di sandboxing non sono sufficienti per fronteggiare i malware più sofisticati. Infatti, molte tipologie di malware al loro interno presentano delle funzionalità definite *antiVM* o *antiSandBoxing* che permettono al malware di capire se si trova in un ambiente virtualizzato o meno. In questo modo il file malevolo potrebbe alterare il proprio comportamento in base all'ambiente in cui viene eseguito.

In questi casi, la corretta classificazione e l'estrazione degli IOC necessita di un intervento umano. Ovvero, risulta necessario studiare in profondità il codice del binario in oggetto. L'analisi del codice assembly, risulta essere molto più dispendiosa in termini di tempo e molto più soggetta ad errori.

In tale contesto, gli esperti di sicurezza informatica stanno utilizzando delle tecniche di machine learning, per creare degli strumenti in grado di supportare l'analista nell'individuare frammenti di codice malevoli.

Il problema dell'individuazione del codice assembly simili tra due binari, come è stato descritto nel capitolo precedente pone una serie di sfide. Tali sfide vengono ulteriormente complicate quando tale problematica viene applicata all'ambito

della malware analysis. Infatti, per loro natura, i malware sono scritti con lo scopo di essere difficilmente individuati dagli analisti. In particolare, l'uso di tecniche di ***obfuscation*** e ***packing*** tendono a rendere estremamente complicata la loro comprensione.

Il problema dell'individuazione del codice simile al fine di individuare e classificare binari malevoli può essere formalizzato nel seguente modo: Dato un repository di malware precedentemente analizzati e un nuovo file da analizzare, l'obiettivo è identificare tutti i frammenti di codice nel repository che sono sintatticamente o semanticamente simili ai frammenti di codice presenti all'interno del file da analizzare.

Le sfide correlate alla seguente problematica possono essere raggruppate nei seguenti due punti:

- ***Efficienza e scalabilità***: la dimensione di un file assembly può variare da un paio di kilobyte a oltre decine di megabyte. Tale disomogeneità impatta sull'efficienza e sulla scalabilità delle soluzioni. L'efficienza di un metodo di rilevamento dei cloni si riferisce al periodo di tempo richiesto per identificare tutti i cloni. La scalabilità si riferisce alla sua capacità di gestire una grande collezione
- ***Ricerca Deterministica***: per supportare l'analisi del malware efficacemente, il metodo di rilevamento del clone deve essere deterministico. Ovvero, i risultati del rilevamento del clone identificati devono essere ripetibili in base alla stessa query di ricerca e al repository malware. Altrimenti, sarà molto difficile per un analista trarre conclusioni sui risultati del clone.

Inoltre, l'individuazione dei cloni, passa dalla definizione del clone stesso. E' possibile definire due concetti di cloni. Il primo concetto, definisce due frammenti di codice come una ***coppia esatta di cloni*** se hanno la stessa sequenza di istruzioni di assemblaggio. Tale ricerca è di semplice applicazione, utilizzando ad esempio una funzione di hash sui frammenti di codici da confrontare. Tale ricerca viene ricondotta all'individuazione dell'uguaglianza tra gli hash associati.

Il secondo concetto, definisce due frammenti di codice come una ***coppia inesatta di cloni*** se hanno delle caratteristiche simili (sintattiche o semantiche). Tale ricerca è maggiormente impattante rispetto alla precedente e prevede la definizione di una funzione di similarità che restituisca come risultato un punteggio di similarità.

Nelle sezioni successive verranno esposti gli algoritmi e i modelli analizzati per affrontare i problemi appena descritti.

## 4.2 Kam1n0

Kam1n0 [17] è un motore di Binary Code Similarity Detection, in grado di individuare in modo efficiente cloni di funzioni all'interno di file binari. Esso si basa sul framework Apache Spark[5] e sul database noSQL Cassandra[6]. Il framework propone al suo interno, un nuovo schema di LSH, integrandolo con degli algoritmi per la ricerca di isomorfismi tra grafi.

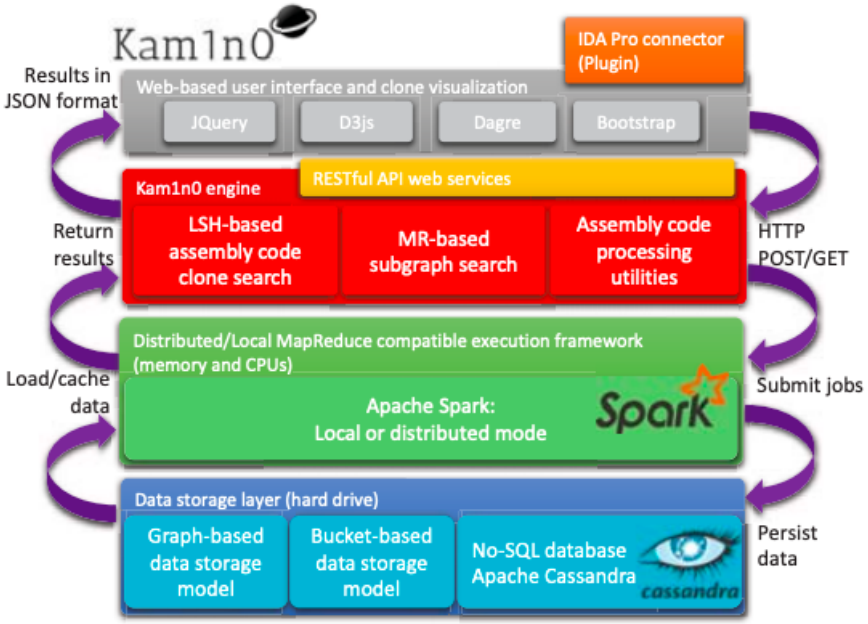


Figura 4.1. Approccio proposto dal framework Kam1n0

L'approccio utilizzato in Kam1n0 come mostra la Fig 4.1 è basato su tre livelli architetturali. Il livello di data storage, riguarda il modo con cui sono memorizzati ed indicizzati i dati. Il secondo livello si occupa dell'esecuzione delle operazioni necessarie ai vari algoritmi di BCSD.

Il terzo livello include il vero e proprio motore di Kam1n0. Esso suddivide una query di ricerca in più job e ne coordina il processo dialogando con il livello sottostante. Tale livello architetturale fornisce anche delle API RESTful per l'integrazione con l'interfaccia web.

[5]Apache Spark è un framework open source per il calcolo distribuito . A differenza del paradigma MapReduce, basato sul disco a due livelli di Hadoop, le primitive "in-memory" multilivello di Spark forniscono prestazioni fino a 100 volte migliori. Ciò permette ai programmi utente di caricare dati in memoria e interrogarlo ripetutamente. Ciò lo rende adatto agli algoritmi di machine learning.

[6]Cassandra è un DBMS non relazionale basato su chiavi, distribuito con licenza open source e ottimizzato per la gestione di grandi quantità di dati.

Sopra all'architettura appena descritta, è stato sviluppata un'interfaccia utente web-based e un plugin per IDAPro. Quest'ultimo di fondamentale importanza per il recupero delle informazioni del binario.

La Fig 4.2, mostra il processo di BCSD utilizzato da Kam1n0. In particolare, le fasi principali sono tre: *preprocessing*, *ricerca dei basic block simili* e *ricerca dei sottografi simili*.

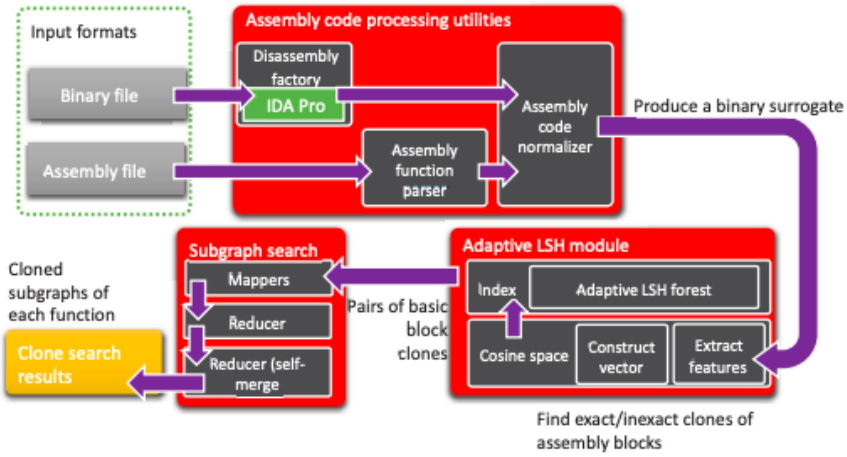


Figura 4.2. Processo di Binary Code Similarity Detection con Kam1n0

La fase di preprocessing consiste nell'analisi del file di input. Kam1n0 accetta due formati per i file in ingresso: binario o codice assembly. Nel primo caso viene estratto l'assembly attraverso operazioni di disassembling effettuate con IDAPro. Successivamente, il codice assembly viene normalizzato, ottenendo una versione generalizzata del codice. Sempre in questa fase vengono estratti anche i CFG delle funzioni.

Nella seconda fase viene applicato un algoritmo di *Adaptive Locality Sensitive Hashing (ALSH)* sulle informazioni ottenute dalla fase di preprocessing, al fine di individuare i basic block simili.

Nell'utlima fase attraverso il modulo di MapReduce, abbina la lista dei basic block simili trovati alle caratteristiche strutturali dei CFG. In questo modo è possibile ottenere i sottografi simili.

Il modulo preposto alla fase di preprocessing in Kam1n0 è costituito da tre sottomoduli principali: il *disassembly factory*, il *assembly function parser* e il *assembly code normalizer*.

Il primo sottomodulo è costituito da un plugin scritto in IDAPython e consente la comunicazione con il disassembler IDAPro. Esso entra in funzione ogni volta che viene fornito come input un file binario. L'output restituito è costituito dal codice assembly, dai CFG delle funzioni e FCG complessivo del programma analizzato.

Il secondo sottomodulo è un parser assembly che permette di suddividere un listato assembly nelle funzioni che lo costituiscono. In questo modo è possibile ottenere lo stesso output generato dal disassembly factory.

Il terzo sottomodulo, si occupa della normalizzazione del codice assembly. Tale normalizzazione in Kam1n0 viene effettuato tramite una trasformazione in uno spazio vettoriale cosinusoidale [18]. Tale spazio vettoriale cosinusoidale permette di mantenere la semantica del codice assembly. Ciò lo rende robusto alle diverse impostazioni dei compilatori.

Ad esempio, al fine di ottimizzare un programma che contiene dei cicli, il compilatore può decidere di trasformare il ciclo in una serie consecutiva di operazioni complessivamente equivalenti. In questo caso, la somiglianza del coseno tra il ciclo così trasformato e quello originale è ancora elevata. Ciò è dovuto perché la distanza del coseno considera solo l'angolo incluso tra i due vettori.

Le features selezionate da Kam1n0 includono istruzioni, operandi e label contenute all'interno del codice assembly. I frammenti di codice assembly, possono essere rappresentati in forme diverse. Per mitigare tale problema, vengono normalizzati gli operandi del codice assembly durante la preelaborazione. Kam1n0 mette a disposizione tre livelli di normalizzazione: *root*, *type* e *specific*.

Il problema di ricercare dei basi block simili, viene affrontata da Kam1n0 attraverso l'uso di un algoritmo *k-NN (K-Nearest Neighbors)* approssimato.

## 4.3 Genius

Genius [19] è un framework che utilizza tecniche di machine learning per individuare bachi o vulnerabilità di sicurezza note all'interno dei software. L'approccio utilizzato da Ginius per individuare funzioni binarie simili può essere suddiviso in quattro passaggi fondamentali, come mostrato nella figura 4.3: *estrazione delle feature*, *generazione del codebook*, *codifica delle feature* e *ricerca online*.

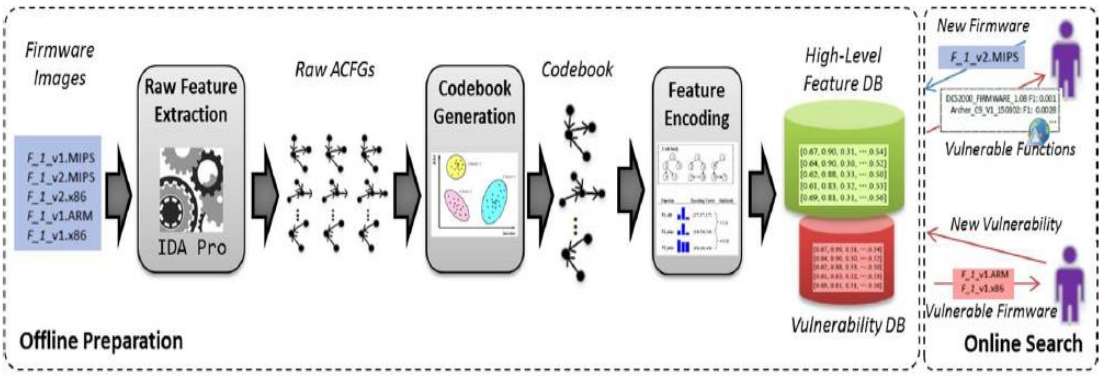


Figura 4.3. Approccio proposto dal framework Genius

Il primo passo consiste nell'estrarre gli ACFG di tutte le funzioni che costituiscono il binario. Successivamente, nella fase di generazione del codebook, vengono utilizzate tecniche di machine learning non supervisionato al fine di effettuare una classificazione preliminare dei grafi estratti. Durante la fase di codifica delle feature i grafi vengono trasformati in vettori di attributi. Infine, data una funzione binaria, la ricerca online mira a trovare efficientemente le funzioni più simili applicando l'algoritmo di *Local Sensitive Hashing (LSH)*.

Come detto in precedenza, Genius non basa il proprio algoritmo esclusivamente sui CFG estratti dal binario, ma li arricchisce con degli attributi relativi ai basic block che li costituiscono, rendendoli ACFG. La scelta degli attributi ricopre quindi un ruolo fondamentale al fine di ottenere ottimi risultati. Nell'approccio utilizzato in Genius sono stati selezionati sei attributi di tipo ***statistico*** e due attributi di tipo ***strutturale***.

Dopo aver estratto le feature necessarie, bisogna passare alla creazione del codebook. Ovvero un dataset che mira ad effettuare una prima categorizzazione delle feature estratte. Formalmente, un codebook C è un set finito e discreto di elementi chiamati codeword: $C = \{c_1, c_2, \ldots, c_k\}$, dove $c_i$ è la i-esima codeword, o "centroide" ed i è un indice intero associato a tale centroide. Il codebook è generato a

| Type | Feature Name | Weight ($\alpha$) |
|---|---|---|
| Statistical Features | String Constants | 10.82 |
| | Numeric Constants | 14.47 |
| | No. of Transfer Instructions | 6.54 |
| | No. of Calls | 66.22 |
| | No. of Instructions | 41.37 |
| | No. of Arithmetic Instructions | 41.37 |
| Structural Features | No. of offspring | 198.67 |
| | Betweenness | 30.66 |

Tabella 4.1. Feature dei basic block usate Genius.

partire da un training set di feature, attraverso algoritmi di machine learning non supervisionato. La generazione del codebook è costituita da due fasi: *calcolo del livello di similarità* e *clustering*.

Genius affronta il problema del livello di similarità delle feature estratte, come un problema di corrispondenza tra ACFG. Il problema di ricercare una corrispondenza esatta tra grafi è un problema *NP-completo*[7] che richiede risorse computazionali elevate. Per tale motivo, Genius risolve il problema *NP-completo* attraverso l'uso di matching approssimati al fine di ottimizzare il processo di ricerca. In particolare, il framework utilizza l'algoritmo bipartite graph matching per quantificare la somiglianza tra gli ACFG.

[7] Nella teoria della complessità computazionale i problemi *NP-completi* sono i più difficili problemi della classe NP ("problemi non deterministici in tempo polinomiale").

## 4.4 Gemini

Gemini [20] è prototipo di motore per la BCDS, che fa uso di *reti neurali* ed *embeddings* al fine di migliorare le prestazioni di ricerca. Tale progetto nasce principalmente per individuare vulnerabilità o bachi all'interno dei firmware, ma i principi possono essere estesi a qualsiasi problema di BCDS. Gemini ripercorre in parte i passi fatti dagli sviluppatori di Genius, migliorando il framework dal punto di vista delle prestazioni e dell'accuratezza. La Fig.4.4 mostra l'approccio proposto da Gemini.

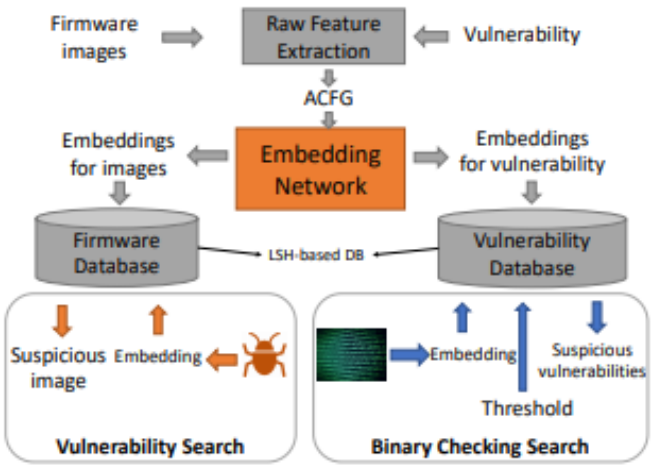


Figura 4.4. Approccio proposto dal framework Gemini

I ricercatori di Gemini, partano dall'assunto di poter trasformare qualsiasi funzione binari in un ACFG. In questo modo il problema può essere risolto attraverso l'addestramento di una rete siamese per riconoscere e differenziare la somiglianza tra due input ACFG.

In particolare viene utilizzata un'architettura a *rete siamese* affiancata ad una struttura di embedding *Structure2Vec*. Un architettura siamese prende due funzioni come input e produce un punteggio di similarità come output. Ciò consente al modello di essere allenato end-to-end con la sola supervisione di una coppia di ACFG.

Il training di una rete siamese richiede un gran numero di coppie di funzioni simili, nonché coppie di funzioni diverse. Tuttavia, nella maggior parte dei casi, i dati di verità di base sono limitati. Per risolvere questo problema, Gemini utilizza una politica di default che considera le funzioni equivalenti (cioè le funzioni binarie compilate dallo stesso codice sorgente) sono simili, e le funzioni non equivalenti non lo sono, in modo da facilitare la generazione di un grande dataset di training.

La rete siamese utilizzata viene utilizzata per elaborare embedding di tipo *Structure2Vec*. In cui gli attributi che costituiscono gli embeddings per certi versi sono simili a quelli utilizzati dal framework Genius Tab.4.2.

| Type | Feature Name ($\alpha$) |
|---|---|
| Block-level attributes | String Constants<br>Numeric Constants<br>No. of Transfer Instructions<br>No. of Calls<br>No. of Instructions<br>No. of Arithmetic Instructions |
| Inter-block attributes | No. of offspring<br>Betweenness |

Tabella 4.2. Feature estratte dai ACFG usate in Gemini.

La caratteristica principale della rete siamese è quella di essere costituita da due reti neurali perfettamente identiche, i cui parametri non variano in fase di apprendimento. La struttura complessiva della rete siamese è illustrata in Fig. 4.5

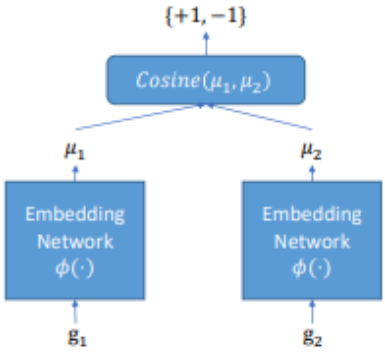


Figura 4.5. Architettura Siamese di Gemini.

Complessivamente il sistema è costituito da due componenti principali. La prima permette di estrarre gli ACFG, ed è stata implementata utilizzando il plugin messo a disposizione da Genius, che permette l'interfacciamento con il decompilatore IDA Pro. La seconda parte, che costituisce la rete siamese, è stata realizzata in TensorFlow[8] Python.

[8]TensorFlow: https://www.tensorflow.org/tutorials

## 4.5 Diaphora

Diaphora [21] è un plugin per IDA Pro, scritto per IDAPython[9], che supporta l'analista nel processo di BinDiffing. Esso ripercorre metodi utilizzati in altri prodotti o progetti open source come *Zynamics BinDiff*, *DarunGrim* o *TurboDiff*. Tuttavia, è in grado di eseguire più azioni rispetto ai precedenti plug-in realizzati per IDA.

Diaphora si basa su diverse tipologie di euristiche per trovare la corrispondenza tra due funzioni. Tali euristiche possono essere suddivise in quattro categorie principali: ***best matches***, ***partial matches***, ***unreliable matches*** ed ***experimental***.

Le euristiche sono applicate su tutte le funzioni che costituiscono i binari in esame. Per poter applicare le euristiche il plugin deve estrarre un gran numero di informazioni relative ai due binari (ad esempio numero di basic block, numero di funzioni, numero di istruzioni, ecc). A tale scopo Diaphora raccoglie il tutto all'interno di database SQLite.

Diaphora è costituito da diversi script. Di seguito si riportano le componenti principali del programma:

- ***diaphora.py***: è lo script IDAPython principale. Contiene tutto il codice delle euristiche, l' interfaccia di esportazione del database e la visualizzazione dei grafi.
- ***jkutils/kfuzzy.py***: è una libreria estratta dal progetto *DeepToad* che permette di effettuare il fuzzy hashing di file binari. Tale libreria è utilizzata da alcune euristiche di Diaphora.
- ***jkutils/factor.py***: questo script è una versione modificata di un toolkit per la clusterizzazione di malware basato sulla teoria dei grafi. Questa libreria permette la fattorizzazione dei numeri in maniera molto rapida e può essere utilizzata per confrontare array di numeri primi. Diaphora utilizza questo script per confrontare i fuzzy AST hash e i call graph fuzzy hash.
- ***Pygments/***: Questa directory contiene una distribuzione della libreria Python *Pygments*[10], che permette di evidenziare la sintassi di tutti i tipi di software, quali sistemi di forum, wiki o altre applicazioni che necessitano l'arrichimento del codice sorgente.

Per avviare il plugin Diaphora, è sufficiente eseguire lo script ***diaphora.py*** dal menu di IDAPro. Una volta caricato, verrà aperta una finestra di dialogo come quella in Fig. 4.6.

Questa finestra di dialogo viene utilizzata sia per l'esportazione del database IDA dal file analizzato, sia per effettuare il confronto con il secondo file.

[9] IDAPython è un plugin per IDAPro che permette di scrivere script nel linguaggio di programmazione Python. Esso offre pieno supporto sia all'API IDA sia a qualsiasi modulo Python installato.

[10] Pygments http://pygments.org/

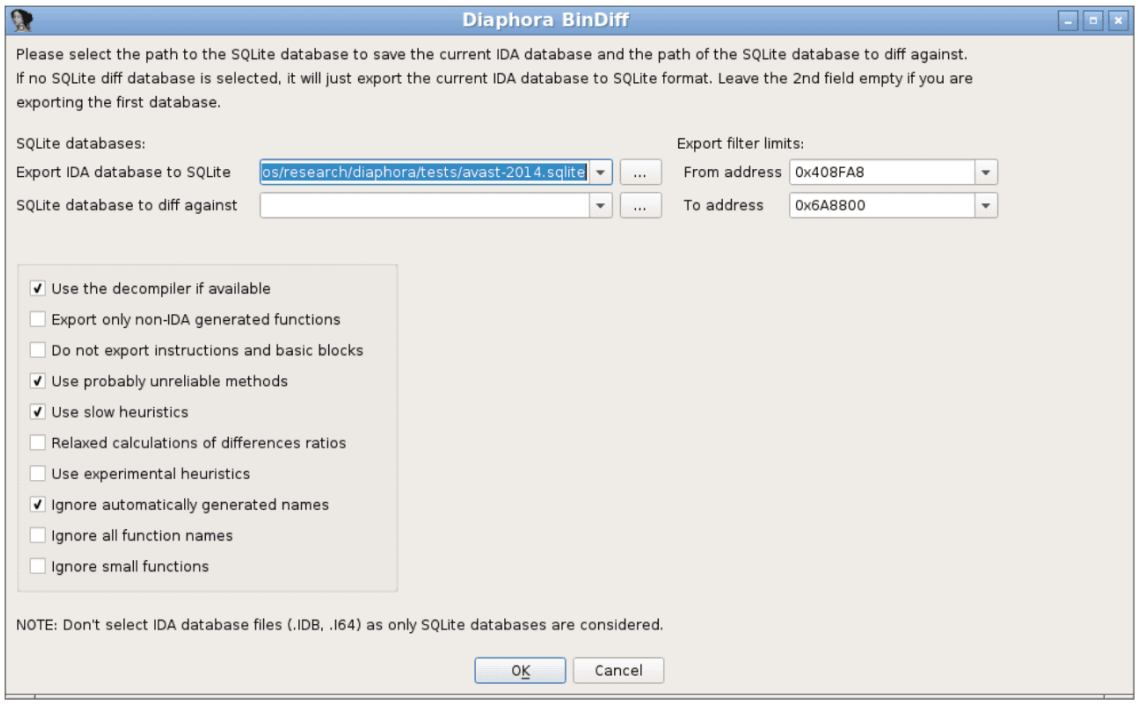


Figura 4.6. Interfaccia Diaphora per esportare il database.

Dalla finestra di dialogo è possibile specificare diverse opzioni, che permettono di rendere il confronto più occurato o di migliorarne le prestazioni:

- ***Uso del decompilatore se disponibile.*** Se il decompilatore Hex-Rays è installato con IDAPro, Diaphora lo utilizzerà per ottenere maggiori informazioni che veranno utilizzato nel processo di bindiffing.
- ***Esportare solo funzioni non generate da IDAPro.*** Pemette di esportare solo le funzioni con nomi non generati automaticamente da IDAPro.
- ***Non esportare istruzioni e basic block.*** Esporta solo le funzioni complete. Questa funzionalità è di aiuto quando si lavora con database di grandi dimezzioni, velocizzandone le operazioni. Tuttavia le capacità di bindiffing saranno limitate.
- ***Uso di euristiche probabilmente inaffidabili.*** Diaphora utilizza molte euristiche per confrontare due funzioni. Tuttavia, alcune euristiche non sono realmente affidabili o il rapporto di somiglianza è molto basso. Con questa opzione è possibile attivare o disattivare l'uso di queste euristiche.
- ***Uso delle euristiche lente.*** Alcune euristiche possono essere piuttosto onerose in termini di tempo. Per database medio grandi è consigliato lasciare disattivata questa opzione. In generale è consigliata l'attivazione solo se con le euristiche precedenti non si ottengono buoni risultati.

- ***Uso delle euristiche sperimentali.*** Permette di abilitare le euristiche sperimentali.
- ***Ignora i nomi generati automaticamente.*** Se abilità permette a Diaphora di escludere il confronto basato sui nomi generati automaticamente da Diaphora. In questo modo non si basa solo sul nome ma analizza il contenuto della funzione.
- ***Ignora le funzioni piccole.*** Permette di ignorare tutte le funzioni con meno di 6 istruzioni assembly.

Dopo aver selezionato correttamente i valori appropriati, lo script inizierà ad esportare tutti i dati in un database SQLite. Al termine di tale operazione sarà possibile analizzare il secondo file confrontandolo con il database del file appena analizzato Fig. 4.7.

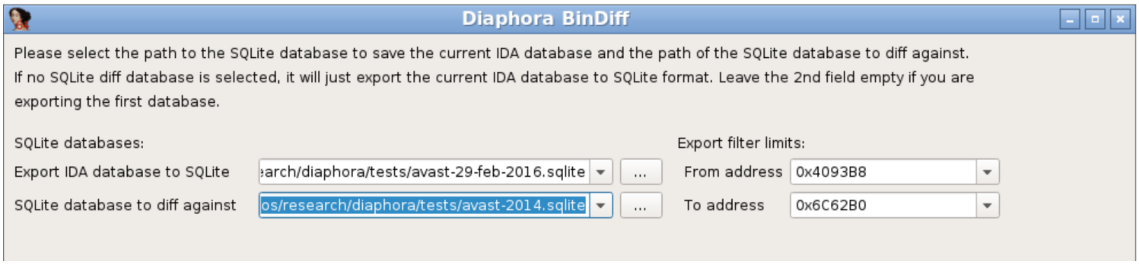


Figura 4.7. Interfaccia Diaphora per effettuare il confronto.

Al completamente dell'operazione di confronto, veranno visualizzate delle liste di funzioni Fig. 4.8: *unmatched in primary*, *unmatched in secondary*, *partial matches* e *best matches*.


Figura 4.8. Elenco con le liste di funzioni estratte.

La prima lista contiene l'elenco di tutte le funzioni presenti all'interno del primo database, che non hanno alcuna corrispondenza all'interno del secondo database. Viceversa, la seconda lista contiene l'elenco di tutte le funzioni presenti all'interno del secondo database che non hanno alcuna corrispondenza con quelle del primo.

Le funzioni in comune tra i due database sono nelle tabelle *best matches* o *partial matches*. I risultati associati ai *best matches* Fig. 4.9 sono funzioni trovate attraverso euristiche che garantiscono l'uguaglianza perfetta.

| Line | Address | Name | Address 2 | Name 2 | Description |
|---|---|---|---|---|---|
| 00000265 | 0040b9c0 | int_set.isra.3 | 0040b9c0 | int_set.isra.3 | 100% equal |
| 00000266 | 0040ba40 | ini_callback | 0040ba40 | ini_callback | 100% equal |
| 00000267 | 0040bc00 | config_free | 0040bc00 | config_free | 100% equal |
| 00000268 | 0040bc50 | config_read | 0040bc50 | config_read | 100% equal |
| 00000269 | 0040bea0 | recv_buffer | 0040bea0 | recv_buffer | 100% equal |
| 00000270 | 0041b680 | StreamingUpdateClient::GetLastUpdateTim... | 0041b520 | StreamingUpdateClient::GetLast... | Equal pseudo-code |
| 00000271 | 00458450 | CryptoPP::Integer::operator-(void)const | 0045bb10 | CryptoPP::Integer::operator-(voi... | Equal pseudo-code |
| 00000272 | 0043fd00 | dep_strAnsiToOem | 0043fe20 | dep_strAnsiToOem | Equal pseudo-code |
| 00000273 | 0043fd20 | dep_strOemToAnsi | 0043fe40 | dep_strOemToAnsi | Equal pseudo-code |

Figura 4.9. Elenco delle funzioni che hanno la corrispondenza migliore.

La Fig. 4.10 , mostra invece l'elenco delle funzioni che hanno una corrispondenza parziale tra i due database. Nella collonna *Description* è presente l'euristica utilizzata e il grado di similarità calcolato attraverso tale euristica.

Valori prossimi ad uno indicano un grado di similarità alto, mentre valori prossimi allo zero indicano un grado di similarità basso.

| Line | Address | Name | Address 2 | Name 2 | Description |
|---|---|---|---|---|---|
| 00000000 | 0040e880 | load_vps | 0040eab0 | load_vps | Bytes hash and names (ratio 0.825000) |
| 00000001 | 0040ec80 | setup_ini_callback | 0040ed70 | setup_ini_callback | Bytes hash and names (ratio 0.923077) |
| 00000002 | 0040ecc0 | engine_init | 0040edb0 | engine_init | Bytes hash and names (ratio 0.753846) |
| 00000003 | 0040f0d0 | engine_scan | 0040f270 | engine_scan | Bytes hash and names (ratio 0.980769) |
| 00000004 | 0040f3e0 | engine_exclude_path | 0040f5b0 | engine_exclude_path | Bytes hash and names (ratio 0.854545) |
| 00000005 | 0040f520 | engine_set_packers | 0040f6f0 | engine_set_packers | Bytes hash and names (ratio 0.913043) |
| 00000006 | 0040f5c0 | engine_set_flags | 0040f790 | engine_set_flags | Bytes hash and names (ratio 0.913043) |
| 00000007 | 0040f660 | engine_set_sensitivity | 0040f830 | engine_set_sensitivity | Bytes hash and names (ratio 0.913043) |
| 00000008 | 0040fc60 | engine_verify_vps | 0040fe30 | engine_verify_vps | Bytes hash and names (ratio 0.818653) |
| 00000009 | 00413c20 | avldrGetEngineInformation | 00413db0 | avldrGetEngineInformation | Bytes hash and names (ratio 0.961749) |
| 00000010 | 00413e70 | aswLoaderDllMain | 00414000 | aswLoaderDllMain | Bytes hash and names (ratio 0.750000) |
| 00000011 | 0041b610 | InitializeStrUpdate | 0041b4b0 | InitializeStrUpdate | Bytes hash and names (ratio 0.727273) |
| 00000012 | 0042b150 | aswcmnbsDllMain | 0042afc0 | aswcmnbsDllMain | Bytes hash and names (ratio 0.652174) |
| 00000013 | 0042b2c0 | cmnbInit | 0042b130 | cmnbInit | Bytes hash and names (ratio 0.746479) |
| 00000014 | 0042b6f0 | aswcmnosDllMain | 0042b560 | aswcmnosDllMain | Bytes hash and names (ratio 0.788991) |
| 00000015 | 0042b880 | cmnosInit | 0042b6f0 | cmnosInit | Bytes hash and names (ratio 0.724490) |
| 00000016 | 0044a950 | DSA_FileVerifyWithSigCompare | 0044ddc0 | DSA_FileVerifyWithSigCom... | Bytes hash and names (ratio 0.978495) |
| 00000017 | 0044b130 | DSA_BlockVerify | 0044e5a0 | DSA_BlockVerify | Bytes hash and names (ratio 0.975610) |
| 00000018 | 0040a200 | main | 0040a200 | main | Same address, nodes, edges and mnemonics (ratio 0.995290) |
| 00000019 | 0040bf00 | handle_scan_item | 0040bf00 | handle_scan_item | Perfect match, same name (ratio 0.434211) |

Figura 4.10. Elenco delle funzioni che hanno una corrispondenza parziale.

E' possibile che a questi elenchi venga aggiunto la lista delle funzioni considerate come *unreliable matches*. Ovvero quei confronti che Diaphora non reputa affidabili.

Inoltre, se viene abilitata l'opzione che cosente l'uso delle euristiche sperimentali veranno visualizzate in una tabella aggiuntiva.

Selezionando una funzione di questo elenco è possibile visualizzare il CFG o le istruzione assembly. In Fig. 4.12 si mostra come Diaphora evidenzia in modo diverso i vari basic block del CFG.

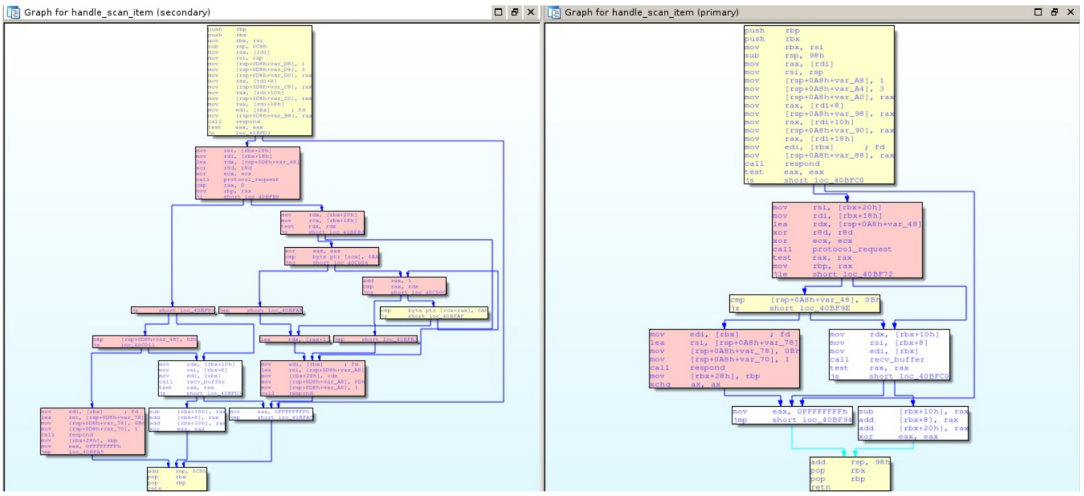


Figura 4.11. Control Flow Graph in Diaphora.

I blochi colorati di giallo sono quelli che presentano poche modifiche, quelli rosa sono quelli che presentano molto modifiche e quelli bianchi quelli che non presentano alcuna modifica.

In maniera analoga a quanto fatto per i CFG, è possibile mostrare le differenze all'interno del codice assembly. La Fig. 4.12 mostra come Diaphora evidenzia tale differenza.

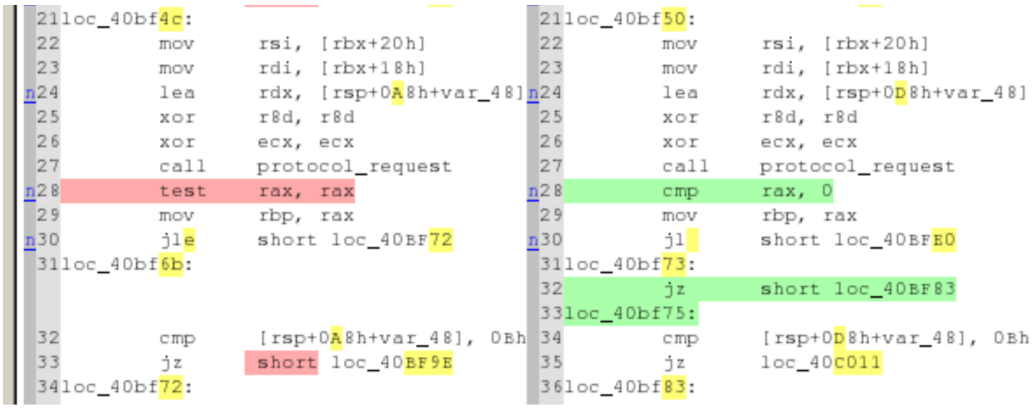


Figura 4.12. Differenze nel codice assembly.

In rosso vengono evidenziate le righe rimosse, in verde le righe aggiunte e in giallo le istruzioni modificate.

## 4.6 SAFE

SAFE (Self-Attentive Function Embeddings) [22] è una nuova architettura che combina l'utilizzao dei *function embeddings* alle *reti neurali self-attentive (SA-NN)* per individuare il grado di similarità tra due binari.

L'approccio utilizzato da SAFE prevede un modello di riconoscimento strutturato su due fasi. La prima fase è realizzata attraverso il componente *assembly instructions embedding*, che trasforma una sequenza di istruzioni assembly in una sequenza di vettori. Nella seconda fase, una *rete neurale self-attentive* trasforma una sequenza di vettori in un singolo vettore di embedding. In Fig. 4.13 una rappresentazione schematica dell'architettura generale utilizzata nella rete di apprendimento di SAFE.

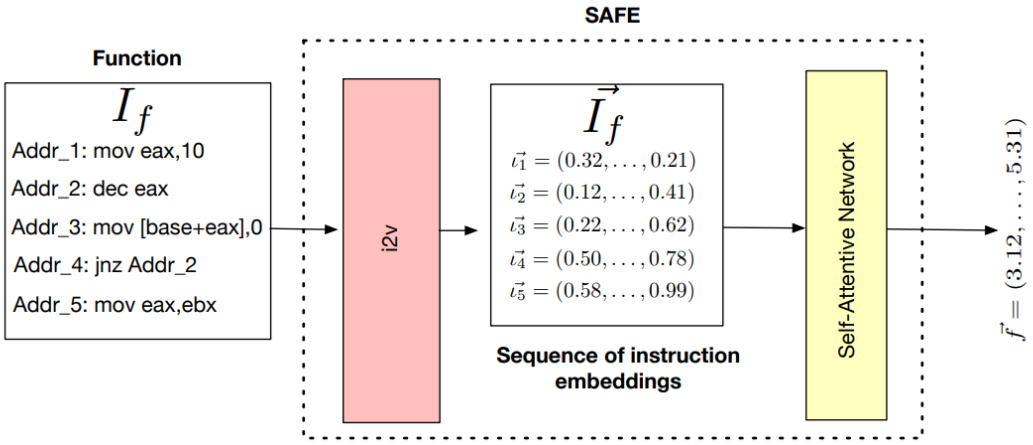


Figura 4.13. Architettura utilizzata in SAFE.

Il primo passo nella soluzione proposta da SAFE, consiste nell'associare un vettore di embeddings a ciascuna istruzione $i$ contenuta in $I_f$. Tale associazione viene effettuata utilizzato l'approccio *instruction2vec (i2v)*[11], utilizzando il metodo *skip-gram*. L'idea di skip-gram è di utilizzare l'istruzione corrente per prevedere le istruzioni precedenti o successivi.

Prima di applicare questa trasformazione l'istruzione assembly viene manipolata attraverso una rappresentazione intermedia. In particolare per ogni istruzione vengono esaminati gli operandi, ed ogni indirizzo di memoria viene sostituito con il simbolo speciale *MEM* e tutte le costanti che superano un certo valore vengono sostituite con il simbolo speciale *IMM*. Questa trasformazione intermedia da un lato non altera la semantica dell'istruzione dall'altro permette di ottenere degli embeddings che sono più facili da confrontare.

[11] La tecnica *istruction2vec (i2v)* è un sottinsieme della tecnica di apprendimento *word2vect (w2v)*. L'*i2v* è applicabile con maggiore efficacia alle istruzioni di un linguaggio di programmazione, a differenza del *w2v* classico che è apllicabile al linguaggio naturale.

Nella seconda parte, viene applicata una rete neurale self-attentive. Nel caso particolare SAFE utilizza una rete neurale bidirezionale riccorrente di recente creazione [23], la cui struttura è rappresentata in FIG. 4.14.

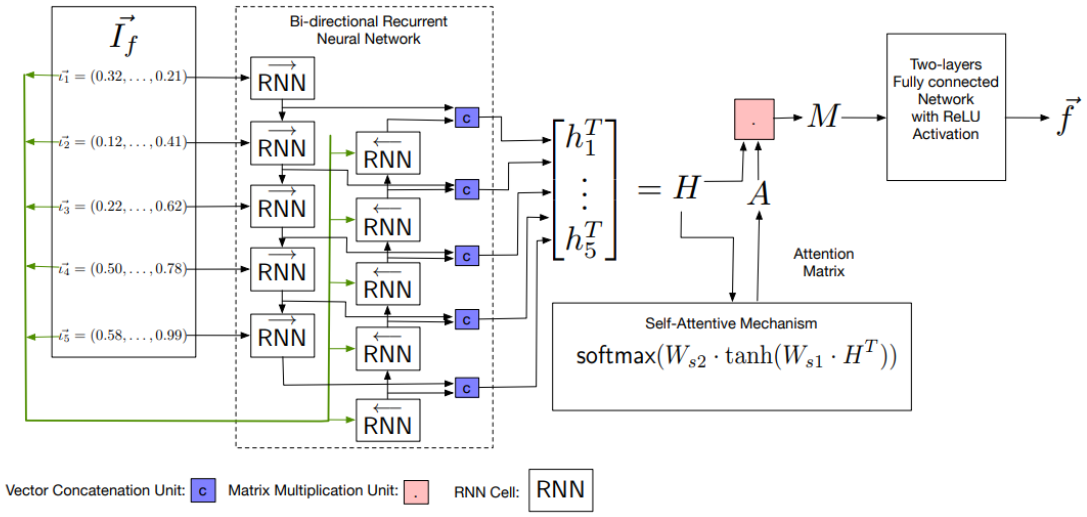


Figura 4.14. Rete Neurale Self-Attentive Network: architettura dettagliata

Il calcolo dell'embedding $\overrightarrow{f}$ di una funzione $f$ viene effettuato utilizzando la sequenza di vettori di istruzioni $\overrightarrow{I_f}$:($\overrightarrow{i_1}$,...,$\overrightarrow{i_m}$). Tali vettori vanno ad alimentare una rete neurale bidirezionale, ottenendo per ciascun vettore $\overrightarrow{i_i} \in \overrightarrow{I_f}$ in un vettore somma di dimensioni $u$:

$$h = \overrightarrow{RNN}(\overrightarrow{h_{i-1}}, i_i) \oplus \overleftarrow{RNN}(\overleftarrow{h_{i+1}}, i_i)$$

dove $\oplus$ è una concatenazione delle reti $\overrightarrow{RNN}$ (diretta) e $\overleftarrow{RNN}$ (retroazione). Mentre $\overrightarrow{h_{i-1}}$ e $\overleftarrow{h_{i+1}}$ sono gli stati diretti e di retroazione delle RNN. Gli stati di ogni cella RNN ha dimensione pari a $\frac{u}{2}$.

Dai vettori ottenuti è possibile ottenere una matrice $H$ di dimensione $m$ x $u$. Tale matrice ha come righe i vettori somma ottenuti. Da tale matrice è possibile ottenere una matrice $A$ di attenzione di dimensione $r$ x $m$ calcolata utilizzando una rete neurale a due strati:

$$A = softmax(W_{s2} \cdot tanh(W_{s1} \cdot H^{T}))$$

Dove $W_{s1}$ è una matrice dei pesi di dimensione $d_a$ x $u$, con $d_a$ il grado di attenzione. La matrice $W_{s2}$ è una matrice dei pesi di dimensione $r$ x $d_a$, con r il numero di hop di attenzione del modello.

In base a quando detto, dato un r = 1, la matrice $A$ collassa in un unico vettore di attenzione, dove ogni valore è il peso di un vettore somma specifico. Quando r > 1, $A$ diventa una matrice e ogni riga è un hop di attenzione indipendente. Questo

significa che ogni hop assegna un peso d'attenzione su un aspetto specifico della funzione binaria. La matrice degli embedding è la seguente:

$$B = (b_1, b_2, ..., b_u) = AH$$

Tale matrice ha dimensione $r$ x $u$. Al fine di trasformare la matrice di embedding in un vettore $\overrightarrow{f}$ di dimensione $n$, è necessario applicare una funzione di attivazione ReLU:

$$\overrightarrow{f} = W_{out2} \cdot ReLU(W_{out1} \cdot (b_1 \oplus b_2 ... \oplus b_u))$$

dove $W_{out1}$ è una matrice dei pesi di dimensione $e$ x $(r + u)$ e $W_{out2}$ è una matrice dei pesi di dimensione $n$ x $e$.

Utilizzando, un approccio chiamato rete siamese Fig. 4.15 è possibile calcolare la similarità tra due vettori $\overrightarrow{f_1}, \overrightarrow{f_2}$. L'idea principale, che sta alla base di questa tecnica è quella di unire due reti di embedding identiche (dove per identiche si intendono reti che condividono gli stessi parametri). L'output finale dell'architettura siamese è il punteggio di somiglianza tra le due fnzioni in ingresso.

Nel dettaglio date due funzioni $<f_1,f_2>$ i vettori $<\overrightarrow{f_1}, \overrightarrow{f_2}>$ sono ottenuti utilizzando la stessa funzione di embedding della rete. Questi vettori vengono confrontati usando la somiglianza del coseno come metrica di distanza, con la seguente formula:

$$similarity(\overrightarrow{f_1}, \overrightarrow{f_2}) = \frac{\sum_{i=1}^{n}(\overrightarrow{f_1[i]} \cdot \overrightarrow{f_2[i]})}{\sqrt{\sum_{i=1}^{n} \overrightarrow{f_1[i]}} \cdot \sqrt{\sum_{i=1}^{n} \overrightarrow{f_2[i]}}}$$

dove $\overrightarrow{f[i]}$ indica la i-esima componente del vettore $\overrightarrow{f}$.

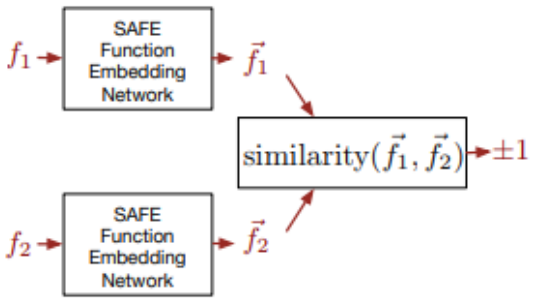


Figura 4.15. Rete Siamese

Per addestrare la rete, è richiesto in input un insieme di coppie di funzioni $K<\overrightarrow{f_1}, \overrightarrow{f_2}>$, i cui output hanno un valore di verità compreso nel range $y_i \in \{+1,-1\}$, dove $y_i$=+1 indica che le due funzioni di input sono simili e $y_i$=-1 che non lo sono.

Quindi utilizzando l'outuput ottenuto dalla rete siamese è possibile definire la seguente funzione obiettivo:

$$J = \sum_{i=1}^{K} (similarity(\overrightarrow{f_1}, \overrightarrow{f_2}) - y_i)^2 + |(A \cdot A^T - I)|_{\mathrm{F}}$$

La funzione obiettivo J è ridotta ai minimi termini utilizzando il gradiente stocastico. Il termine $|(A \cdot A^T - I)|_{\mathrm{F}}$ viene utilizzato per introdurre una penalizzazione ad ogni iterazione della matrice di attenzione A.

## 4.7 Machoc

Il Machoc hash è un algoritmo di fuzzy hashing basato sul CFG di una funzione. Tale algoritmo è stato sviluppato all'interno del progetto Polichombr [24].

Gli attuali algoritmi di hashing (come MD5, SHA1 o SHA256) non sono per loro natura resistenti ai cambiamenti. Infatti, uno dei punti di forza degli algoritmo di hash è l'*effetto valanga*[12], in cui per una piccola variazione dell'input, produce una notevole variazione all'interno del hash. Per tale ragione i classici algoritmi di hashing non sono adatti a risolvere il problema di individuazione di funzioni simili.

Gli algoritmi SSDEEP e SDHash tollerano modifiche parziali dei dati, ma non sono adatti per i file binari. Tuttavia, nel caso specifico dell'analisi del malware, solo alcune aree del sample sono utili e dovrebbero essere prese in considerazione dai metodo di confronto. Per tale motivo, nell'ambito del progetto Polichombr, è stato necessario la definizione di un nuovo tipo di algoritmo di hashing, resistente ai cambiamenti minimi all'interno del codici.

L'algoritmo Machoc, calcola per ogni funzione un fuzzy hash basandosi sulle relazioni tra i basic block del CFG. Quindi un hash Machoc identifica una singola funzione all'interno del binario. Tuttavia, la concatenazione di una sequenza di tali hash è rappresentativa del campione analizzato e lo identifica. Nello specifico, l'algoritmo prevede i seguenti passi per ottenere il Machoc hash:

1. Per ogni funzione, i basic block che compongono il CFG devono essere indicati con numeri progressivi, ordinati in base all'indirizzo di partenza del blocco.
2. Ogni basic block deve essere tradotto in una stringa con il seguente formato: *N: [c,] [DST, ...]* seguito da un punto e virgola. Dove:
   - *N*: è il numero del basic block preso in cosiderazione;
   - *c*: indica la presenza di istruzioni *call* all'interno del blocco;
   - *DST, ...*: indica il numero dei blocchi successivi. Ogni blocco deve essere separato da una virgola.
3. L'output ottenuto viene elaborato attraverso una funzione di fuzzy hashing (nell'ultima implementazione di Machoc è stata utilizzata *Murmurhash3* che genera una stringa a 32 bit);
4. Infine, le stringhe ottenute vanno concatenate per ottenere il Machoc hash dell'intero binario.

In Fig.4.16, l'applicazione dell'algoritmo Machoc al CFG di una funzione costituita da 10 basic block, due dei quali contengono l'istruzione *call*.

Nel contesto del progetto Polichombr, il Machoc Hash viene utilizzato per individuare funzioni simili estratte da campioni differenti. La corrispodenza viene trovata

[12]Effetto valanga: https://en.wikipedia.org/wiki/Avalanche_effect

in base al grado di similarità utilizzando la distanza di Jaccard o attraverso euristiche (Ad esempio, date due funzioni in due file binari, se le funzioni circostanti in entrambi i binari sono in corrispondenza diretta, le due funzioni sono probabilmente simili nonostante producano un hash diverso).

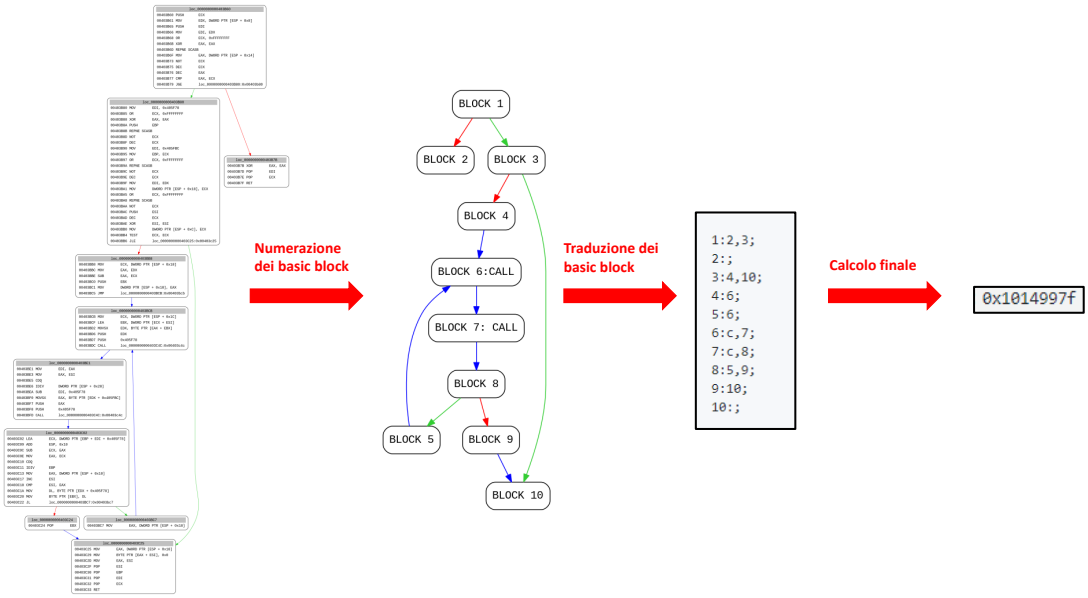


Figura 4.16. Applicazione dell'algoritmo Machoc ad un CFG di una funzione contenente 10 basic block, due dei quali contengono l'istruzione *call*.

Come detto in precedenza, i Machoc hash possono essere combinati per ottenere una signature dell'intero binario. Esperimenti empirici mostrano che una distanza di Jaccard dell'80% indica una buona corrispondenza tra due binari.

Le caratteristiche descritte, rendono l'algoritmo molto utile nell'ambito della malware analysis. Per tale ragione sono state create delle implementazione che si interfacciano con IDA Pro[13] e Radare2[14].

[13]Idadiff: https://github.com/0x00ach/idadiff

[14]Machoke: https://github.com/conix-security/machoke/

# 5 Soluzione

## 5.1 Dataset

Al fini di testare i vari algoritmi di Binary Code Similarity Detectione e Binary Diffing è stato creato un dataset costituito da 1000 file binari con architettura win32. Il dataset è equamente ripartito tra *goodware* (principalmente file di sistema di Windows) e *malware*. I malware presi in considerazione sono suddivisi in 16 famiglie principali, contenenti numerose varianti recuperati nell'arco temporale di 3 mesi.

| Famiglia | Tipologia | n° varianti | n° sample |
|---|---|---|---|
| Zeus | Banking Trojan | 14 | 39 |
| Neverquest | Banking Trojan | 6 | 14 |
| Gozi (Ursnif) | Banking Trojan | 29 | 46 |
| Dridex | Banking Trojan | 18 | 35 |
| Ramnit | Banking Trojan | 8 | 23 |
| GozNym | Banking Trojan | 6 | 10 |
| Timba | Banking Trojan | 4 | 11 |
| Gootkit | Banking Trojan | 11 | 31 |
| Trickbot | Banking Trojan | 10 | 21 |
| Quadars | Banking Trojan | 5 | 10 |
| Emotet | Banking Trojan | 16 | 40 |
| Adylkuzz | Cryptojacking | 4 | 20 |
| Gbhveixig | Cryptojacking | 8 | 11 |
| Locky | Ransomware | 7 | 14 |
| Petya | Ransomware | 5 | 24 |
| WannaCry | Ransomware | 5 | 26 |
| Generic Malware | altro | | 125 |
| TOT | | | 500 |

Tabella 5.1. Famiglie malware suddivise per tipologia.

Come mostra la Tab. 5.1, il dataset è costituito principalmente da malware appartenenti alla tipologia *Banking Trojan*. Con tale termine viene identificata una categoria di malware in grado di rubare credenziali bancarie dai pc delle vittime allo scopo di effettuare transazioni fraudolente. Essi sfruttano tecniche di attacco chiamate ***"Man in the Browser"*** in cui i trojan prendono il controllo del browser, intercettando tutte le chiamate effettuate con i siti di istituti bancari. Si è scelto

di inserire nel dataset un gran numero di famiglie di questa tipologia perché, come evidenzia Proofpoint nei report del 2018 [25][26], attualmente risultano essere i malware maggiormente diffusi all'interno delle email di phishing. Tra questi *Emotet* è stata la famiglia più diffusa, rappresentando il 57% dei Banking Trojan.

La seconda tipologia con maggior numero di sample all'interno del dataset sono i *Ransomware*. Con tale termine si indica una categoria di malware che ha lo scopo di rendere inutilizzabile gran parte del sistema infetto (ad esempio cifrando file e cartelle), al fine di richiedere un riscatto economico alla vittima per ripristinare il normale funzionamento del sistema. Il precursore di tale tipologia, è di certo *WannaCry*, che ha causato danni per miliardi di dollari in tutto il mondo. Successivamente, nuove famiglie di ransomware hanno trovato grande diffusione, rimanendo al secondo posto nei report di Proofpoint.

La terza tipologia che viene indicata nella tabella si riferisce ai *Cryptojacking*. Con tale termine si indicano una tipologia di malware che sfruttano le capacità computazionali di un pc (CPU, GPU, RAM, ecc) al fine di generare criptovalute (ad esempio Bitcoin, Monero o Ethereum), senza che la vittima ne sia a conoscenza.

Di seguito si riporta il grafico 5.1 delle famiglie di malware utilizzate. Il grafico è stato tagliato per mostrare le 16 famiglie di malware analizzate. Tutte le famiglie di malware, presenti all'interno del dataset che hanno meno di 10 campioni sono state scartate durate il processo di classificazione.

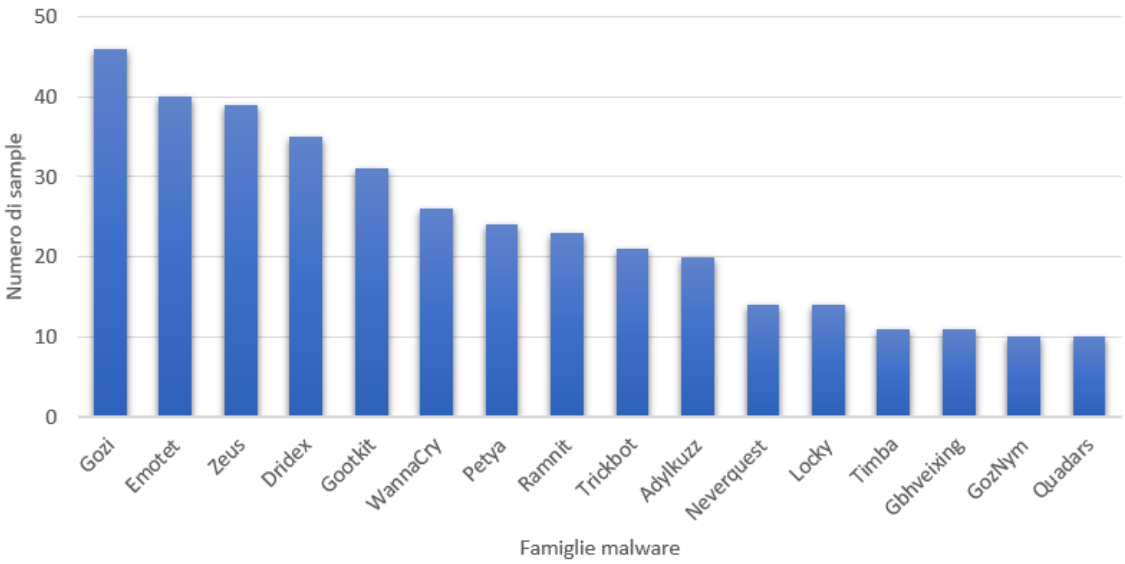


Figura 5.1. Distribuzione famiglie malware all'interno del dataset.

## 5.2 Metodologia

I malware utilizzati sono stati scaricati per la maggior parte dal repository online di ***MalShare***[5] e in piccola parte dal repository di ***theZoo***[6]. Successivamente la classificazione è stata effettuata attraverso il tool AVCLASS [27]. AVCLASS è uno strumento di etichettatura automatico, che fornisce signature antivirus per un numero elevato di campioni. Esso fornisce come risultato, le famiglie malware più probabili d'associare per ciascun campione.

L'etichettatura di un eseguibile malevolo, come variante di una famiglia nota, è importante per molteplici aspetti in sicurezza. In primo luogo, risulta fondamentale per l'identificazione di nuove minacce, per la realizzazione di signature antivirus, per l'attribuzione e massive triage durante le fasi di incident response.

Tale processo può essere effettuato manualmente dagli analisti, o automaticamente. Gli approcci automatici di classificazione si basano sull'apprendimento automatico supervisionato (assumendo che il campione appartenga a una famiglia nota all'interno del dataset di training), oppure attraverso approcci di clustering (non supervisionato) in cui vengono evidenziati gruppi di sample appartenenti allo stesso cluster. In quest'ultimo caso l'etichetta identifica il cluster e viene assegnata manualmente.

Il problema principale dei motori di etichettatura è quello di reperire delle etichette corrette ed affidabili. Le etichette generate dagli antivirus (AV) spesso sono incoerenti tr di loro. In particolare, i motori AV spesso non concordano sul fatto che lo stesso campione sia malevolo o meno, e il nome di famiglia che l'etichetta codifica può differire. Inoltre, gli AV non utilizzano solitamente una convenzione standard per assegnare le etichette (convenzioni come CARO [28] e CME [29] non sono ampiamente utilizzati). Ciò è dovuto al fatto che l'obiettivo principale di un AV è il rilevamento e non sempre è di interesse la classificazione.

AVCLASS propone un approccio di etichettatura automatica che affronta le più importanti cause di discrepanza tra AV. Principalmente, permette di unificare e normalizzare i diversi schemi di nomenclatura, i token generici e gli alias.

In Fig.5.2 è rappresentata l'architettura generale di AVCLASS. In tale architettura, la fase di etichettatura è il nucleo di AVCLASS che implementa il processo di normalizzazione delle etichette. Tale processo necessità in input di un campione da etichettare, un elenco di token generici, un elenco di alias e facoltativamente di un elenco di motori antivrus da utilizzare.

Per ogni campione da etichettare, il tool genera una classifica dei nomi di famiglia più probabili. L'elenco di token generici ed alias sono il risultato della fase di

[5] MalShare è un progetto in collaborazione con diverse comunità di sicurezza, che ha lo scopo di creare un repository pubblico ed accessibile a chiunque, per condividere e studiare malware. Fondato da Silas Cutler, vanta diverse collaborazioni con grossi vendor di security come CrowdStrike, Tehtris e Farsight. Sito: https://malshare.com/

[6] theZoo è un progetto open source che raggruppa molti malware in un unico repository. Ciò lo rende consultabile da chiunque in modo sicuro. Realizzato inizialmente da Yuval Tisf Nativ adesso e mantenuto e aggiornato da Shahak Shalev. Sito: https://github.com/ytisf/theZoo

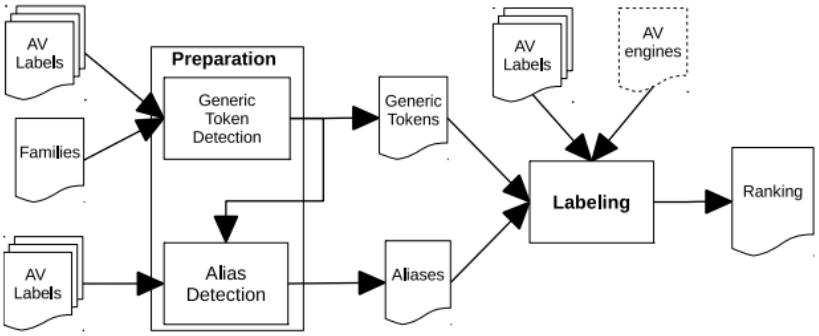


Figura 5.2. Architettura di AVCLASS.

preparazione. Da impostazione di default, AVCLASS utilizza tutti i motori antivirus al fine di scegliere le etichette più probabili da assegnare.

Tuttavia, fornendo un elenco di motori AV, l'analista può limitare l'elaborazione alle sole etichette dei motori selezionati. Ad ogni modo, è consigliabile utilizzare il maggior numero di antivirus possibile, in quanto alcuni sample possono essere stati analizzati solo da alcuni vendor di AV. In particolare, AVCLASS estrae le etichette AV da ***VirusTotal (VT)***[7], che utilizza diversi motori AV per scansionare i file sottomessi.

Complessivamente, i campioni vengono scansionati da 99 diversi motori AV, elencati in dettaglio in Tab 5.2. In tabella sono elencati diversi motori antivirus per lo stesso vendor (ad esempio McAffee e McAffee-GW-Edition o TrendMicro e TrendMicro-HouseCall), ciò è dovuto al fatto che spesso solo alcuni engine dello stesso venditore rilevano alcune minacce.

Al fine di effettuare la normalizzazione dell'etichetta, nella prima fase AVCLASS rimuove i duplicati. Successivamente, attraverso analisi empiriche, ha evidenziato che la maggior parte del rumore introdotto nelle etichette AV è indotta dal suffisso (cioè la parte dell'etichetta AV dopo il nome della famiglia). In generale, risulta complicato rimuovere tali suffissi per tutti i motori AV, poiché i fornitori utilizzano strutture di etichettatura differenti.

In particolare, AVCLASS applica delle regole specifiche in base alla tipologia di AV. Le regole principali che vengono applicate per troncare i suffissi sono:

- troncare l'etichetta dopo l'ultimo punto;
- troncare l'etichetta dopo l'ultimo carattere "-";
- troncare l'etichetta dopo l'ultimo carattere "!".

[7]VirusTotal è un sito web che permette di analizzare gratuitamente file, url o IP, utilizzando diversi motori AV o strumenti di ip/url reputetion. Sito: https://www.virustotal.com

Il passo successivo, è quello di dividere ogni etichetta in token. Utilizzando una regola di tokenizzazione che divide l'etichetta in una sequenza di caratteri alfanumerici consecutivi.

| Engine VT | | |
|---|---|---|
| Ikarus | DrWeb | VirusBuster |
| TheHacker | Sophos | eTrust-Vet |
| F-Prot | F-Secure | TrendMicro-HouseCall |
| Fortinet | AhnLab-V3 | SUPERAntiSpyware |
| BitDefender | Norman | Emsisoft |
| CAT-QuickHeal | Rising | VIPRE |
| AVG | AntiVir | PCTools |
| Microsoft | GData | Authentium |
| ClamAV | ViRobot | ByteHero |
| Avast | K7AntiVirus | Sunbelt |
| McAfee | Comodo | TotalDefense |
| TrendMicro | nProtect | NOD32 |
| VBA32 | McAfee-GW-Edition | ESET-NOD32 |
| Symantec | Antiy-AVL | Commtouch |
| Kaspersky | eSafe | Agnitum |
| Panda | Jiangmin | Kingsoft |
| Malwarebytes | AegisLab | MicroWorld-eScan |
| K7GW | McAfee+Artemis | NANO-Antivirus |
| Prevx | PandaBeta | NOD32Beta |
| NOD32v2 | Zillya | Arcabit |
| Ewido | FileAdvisor | SecureWeb-Gateway |
| eTrust-InoculateIT | Tencent | VIRobot |
| UNA | Zoner | Command |
| Baidu | Cyren | PandaB3 |
| Baidu-International | Avira | eScan |
| F-Prot4 | Webwasher-Gateway | DrWebSE |
| Bkav | AVware | ESET |
| Antivir7 | a-squared | Yandex |
| CMC | Avast5 | TotalDefense2 |
| T3 | McAfeeBeta | SymCloud |
| Prevx1 | FortinetBeta | Qihoo-360 |
| Ad-Aware | PandaBeta2 | AhnLab |
| SAVMail | ALYac | TrendMicroXgen |

Tabella 5.2. Motori antivirus utilizzati da VT.

Dopo aver suddiviso l'etichetta in token, è necessario rimuovere tutti quei token che non sono riconducibili a famiglie malware specifiche.

A tale scopo, AVCLASS utilizza una lista di token generici suddivisi in diverse categorie come mostra la Tab 5.3.

Una volta effettuata la classificazione attraverso AVCLASS, si è provveduto a realizzare uno script per disassemblare i file presenti all'interno del dataset.

| Category | Tokens | Example Tokens |
|---|---|---|
| Architecture | 14 | android, linux, unix |
| Behavior: download | 29 | download, downware, dropped |
| Behavior: homepage-modification | 2 | homepage, startpage |
| Behavior: injection | 5 | inject, injected, injector |
| Behavior: kill | 5 | antifw, avkill, blocker |
| Behavior: signed | 2 | fakems, signed |
| Behavior: other | 3 | autorun, proxy, servstart |
| Corrupted | 2 | corrupt, damaged |
| Exploit | 2 | expl, exploit |
| File-types | 15 | html, text, script |
| Generic-families | 13 | agent, artemis, eldorado |
| Heuristic-detection | 12 | generic, genmalicius, heuristic |
| Macro | 11 | badmacro, macro, x2km |
| Malicious-software | 5 | malagent, malicious, malware |
| Malware-classes | 53 | spyware, trojan, virus |
| Misc | 9 | access, hack, password |
| Packed | 17 | malpack, obfuscated, packed |
| Packer | 6 | cryptor, encoder, obfuscator |
| Patch | 3 | patched, patchfile, pepatch |
| Program | 5 | application, program, software |
| PUP | 29 | adware, pup, unwanted |
| Suspicious | 13 | suspected, suspicious, variant |
| Test | 2 | test, testvirus |
| Tools | 8 | fraudtool, tool, virtool |
| Unclassified | 3 | unclassifiedmalware, undef, unknown |

Tabella 5.3. Motori antivirus utilizzati da VT.

## 5.3 Test e Risultati

Durante la fase di test sono stati presi in esame cinque algoritmi differenti: *Kam1n0*, *Genius*, *Gemini*, *Diaphora*, *SAFE* e *Machoc*, al fine di analizzare prestazioni e precisione nell'individuazione e dei malware.

Il dataset è stato suddiviso in due parti: *training set* e *test set*, mantenendo per entrambi la copertura di tutte le famiglie malware prese in considerazione. In particolare il 67% dei campioni costituisce il training set, mentre il restante 33% viene utilizzato per il test set. La seguente suddivisione si è resa necessaria al fine di allenare gli algoritmi a riconoscere determinate famiglie di malware.

I risultati ottenuti sono stati rappresentati attraverso **"matrici di confusione"**. Tali matrici, definite anche come tabelle di errata classificazione, restituiscono una rappresentazione dell'accuratezza statistica ottenuta. Le colonne della matrice rappresentano i valori predetti, mentre le righe rappresentano i valori reali. L'elemento sulla riga *i-esima* e sulla colonna *j-esima* è il numero di casi in cui il classificatore ha classificato la classe $i$ come classe $j$. Di particolare interesse sono i valori posizionati

sulla diagonale della matrice che rappresentano i veri positivi per ogni famiglia di malware analizzati. Inoltre per ogni algoritmo, viene rappresentata anche la matrice di classificazione normalizzata sulla base dei campioni analizzati.

In Fig. 5.3 e In Fig. 5.4 si riportano i risultati di classificazione ottenuti con l'algoritmo Kam1n0. Complessivamente l'algoritmo ha classificato in maniera corretta il 52% dei campioni sottoposti ed ha sbagliato nel restante 48% dei casi.

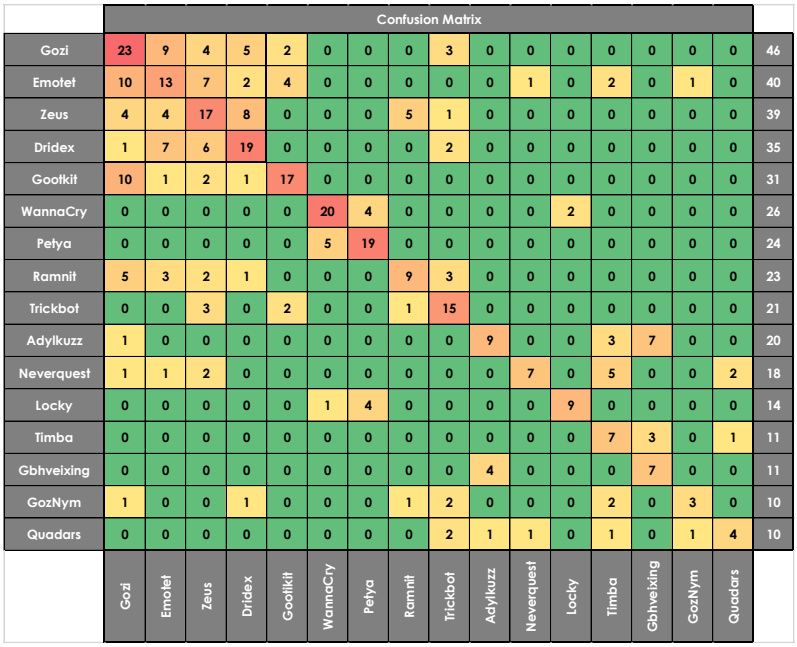


Figura 5.3. Matrice di confusione di Kam1n0

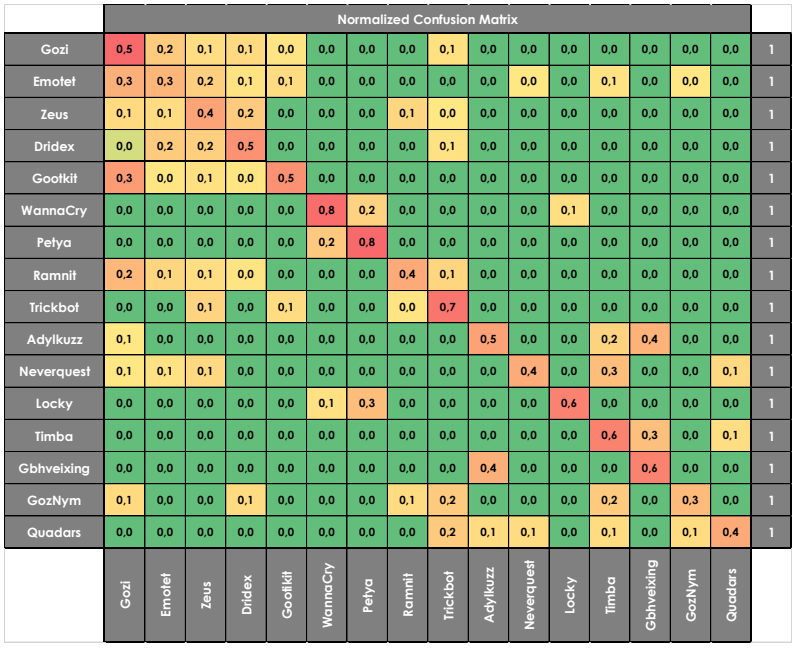


Figura 5.4. Matrice di confusione normalizzata di Kam1n0

In Fig. 5.5 e In Fig. 5.6 si riportano i risultati di classificazione ottenuti con l'algoritmo Genius. Complessivamente l'algoritmo ha classificato in maniera corretta il 82% dei campioni sottoposti ed ha sbagliato nel restante 18% dei casi.

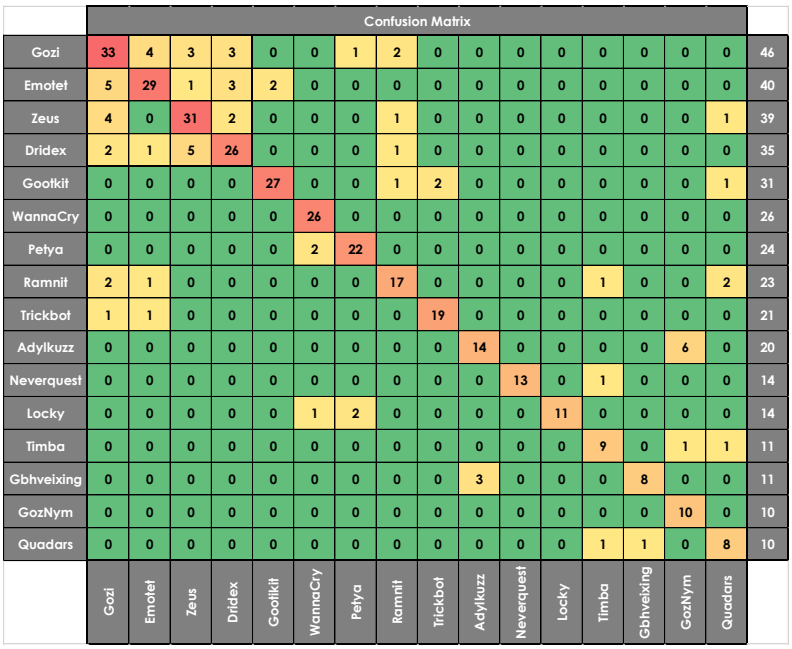


Figura 5.5. Matrice di confusione di Genius

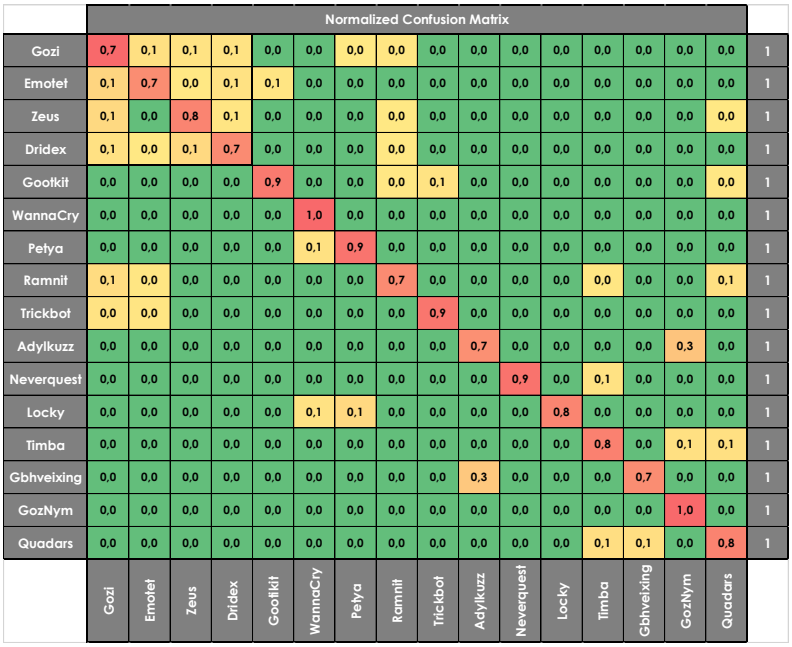


Figura 5.6. Matrice di confusione normalizzata di Genius

In Fig. 5.5 e In Fig. 5.6 si riportano i risultati di classificazione ottenuti con l'algoritmo Gemini. Complessivamente l'algoritmo ha classificato in maniera corretta il 83% dei campioni sottoposti ed ha sbagliato nel restante 17% dei casi.

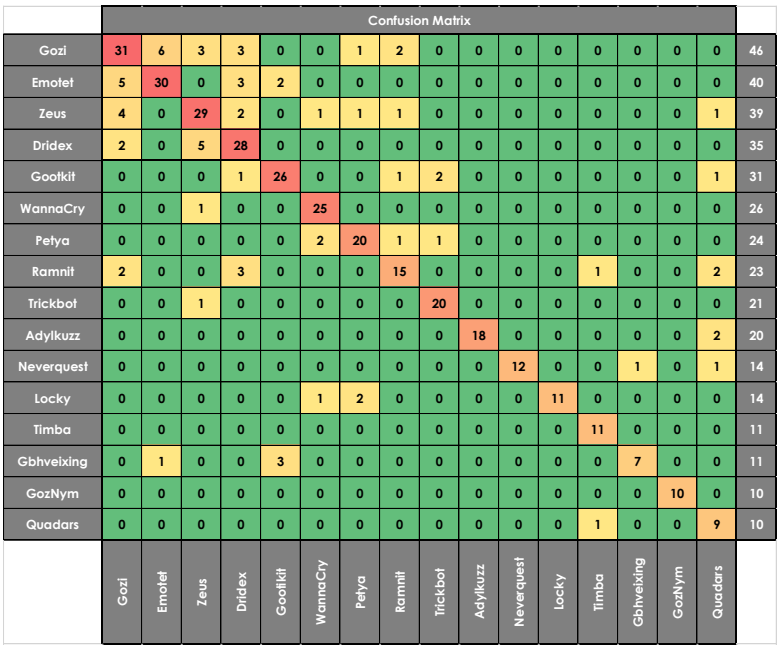


Figura 5.7. Matrice di confusione di Gemini

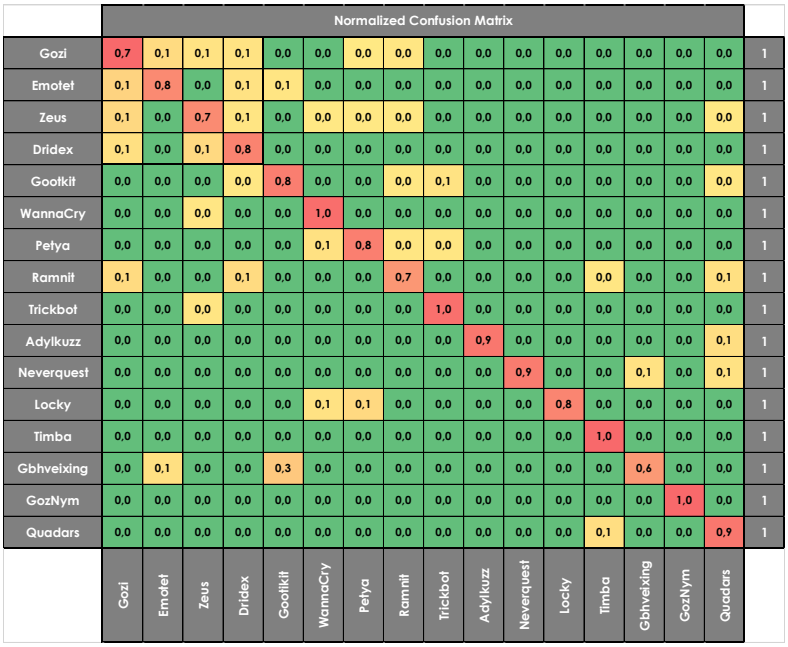


Figura 5.8. Matrice di confusione normalizzata di Gemini

In Fig. 5.9 e In Fig. 5.10 si riportano i risultati di classificazione ottenuti con l'algoritmo Diaphora. Complessivamente l'algoritmo ha classificato in maniera corretta il 83% dei campioni sottoposti ed ha sbagliato nel restante 17% dei casi.

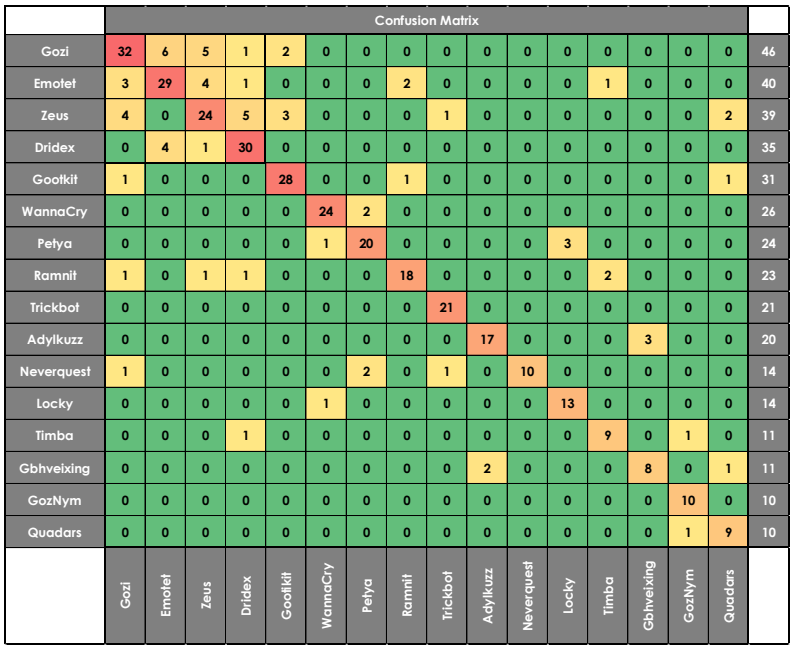


Figura 5.9. Matrice di confusione di Diaphora

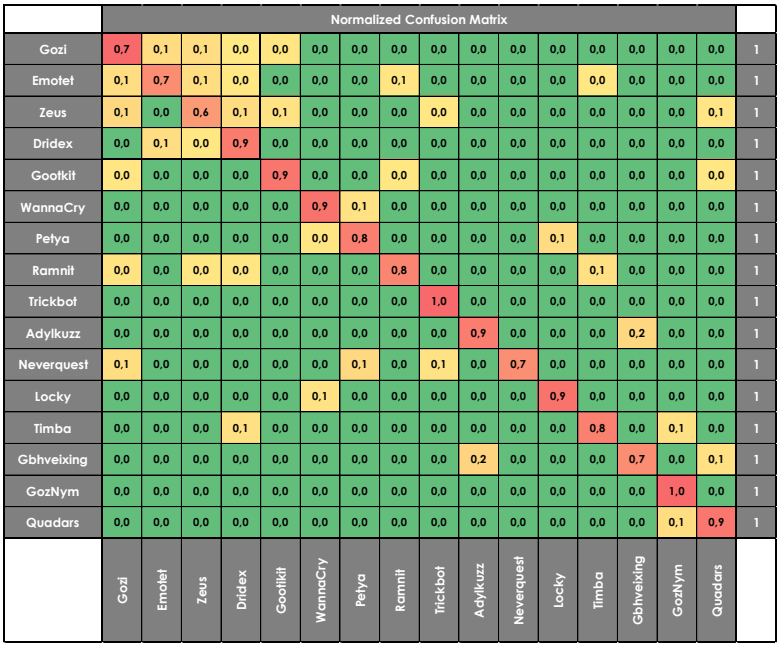


Figura 5.10. Matrice di confusione normalizzata di Diaphora

In Fig. 5.11 e In Fig. 5.12 si riportano i risultati di classificazione ottenuti con l'algoritmo SAFE. Complessivamente l'algoritmo ha classificato in maniera corretta il 79% dei campioni sottoposti ed ha sbagliato nel restante 21% dei casi.

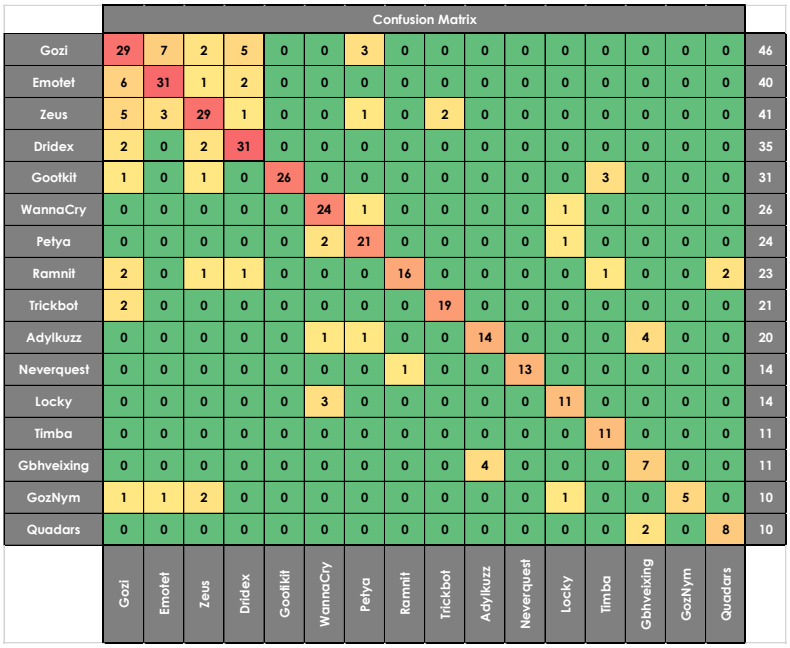


Figura 5.11. Matrice di confusione di SAFE

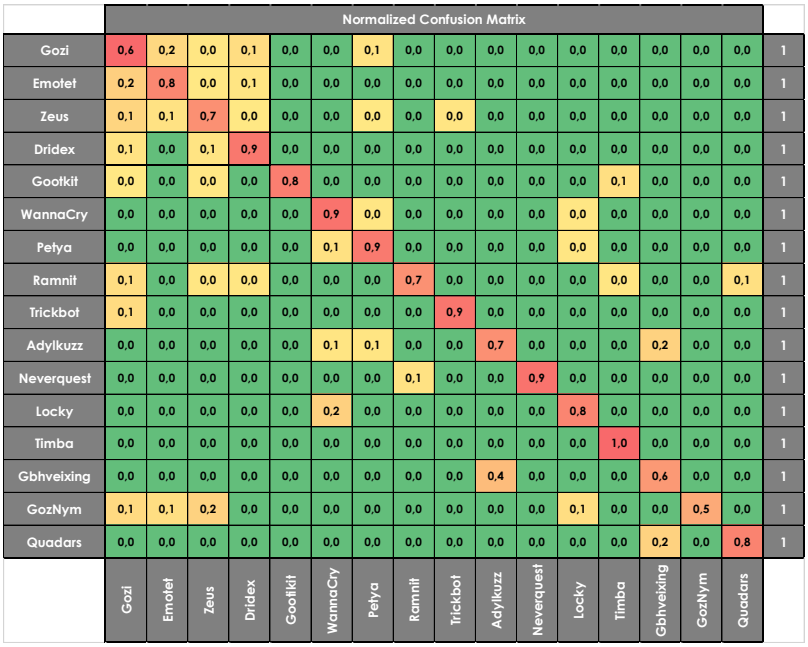


Figura 5.12. Matrice di confusione normalizzata di SAFE

In Fig. 5.13 e In Fig. 5.14 si riportano i risultati di classificazione ottenuti con l'algoritmo Diaphora. Complessivamente l'algoritmo ha classificato in maniera corretta il 59% dei campioni sottoposti ed ha sbagliato nel restante 41% dei casi.

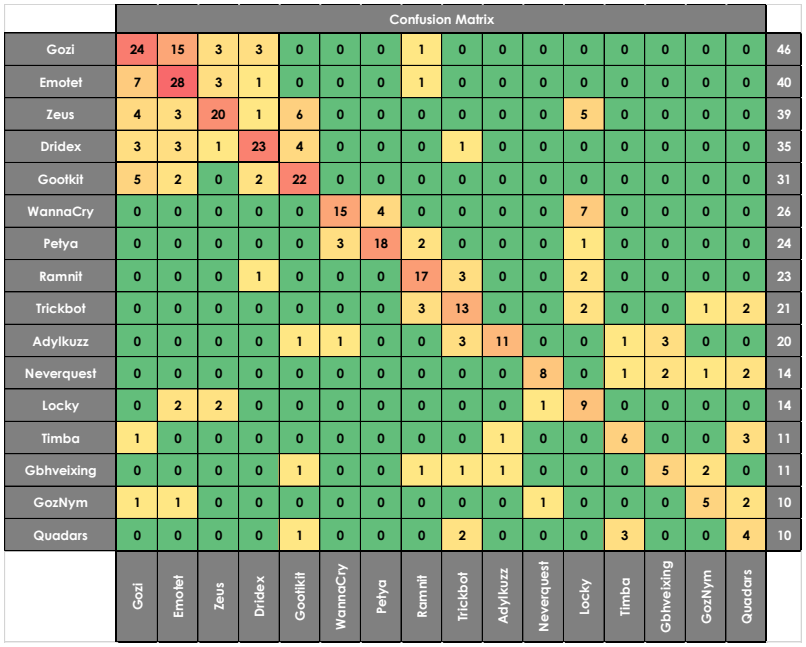


Figura 5.13. Matrice di confusione di Machoc

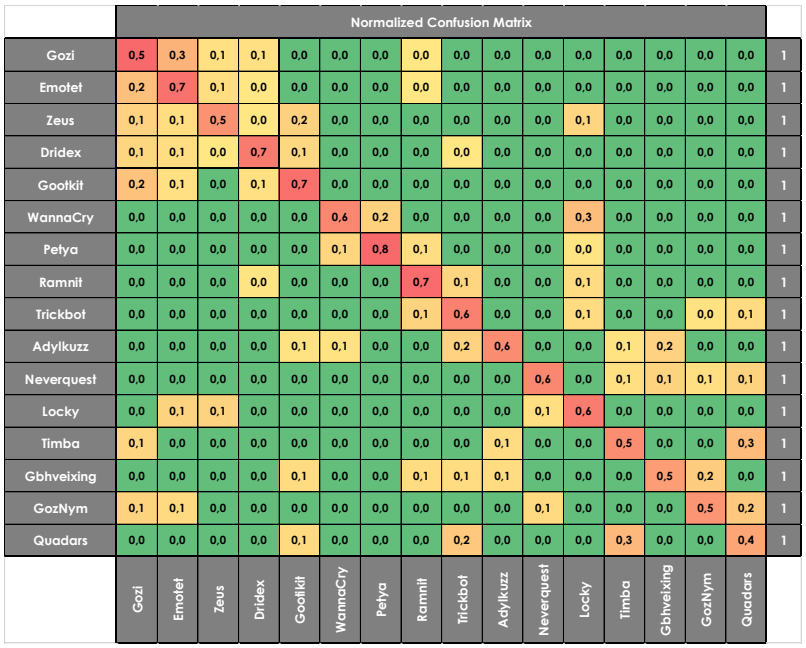


Figura 5.14. Matrice di confusione normalizzata di Machoc

Analizzando i risultati ottenuti, risulta particolarmente evidente che tutti gli algoritmi presi in considerazione hanno riscontrato un maggiore *"grado di confusione"* nella parte in alto a sinistra della matrice. Infatti, i malware appartenenti alle famiglie *Gozi*, *Emotet*, *Zeus* e *Dridex* presentano delle caratteristiche simili all'interno

del codice binario. In particolare, durante l'analisi manuale dei malware, vi sono evidenze del fatto che le famiglie identificate presentano dei *loader* condivisi. La parte di codice definita come loader (o packer) permette di decifrare e caricare in memoria il vero payload malevolo. Come mostra la Fig. 5.15, la struttura complessiva del codice relativo alla componente del loader risulta identica per tutte e quattro le famiglie malware.

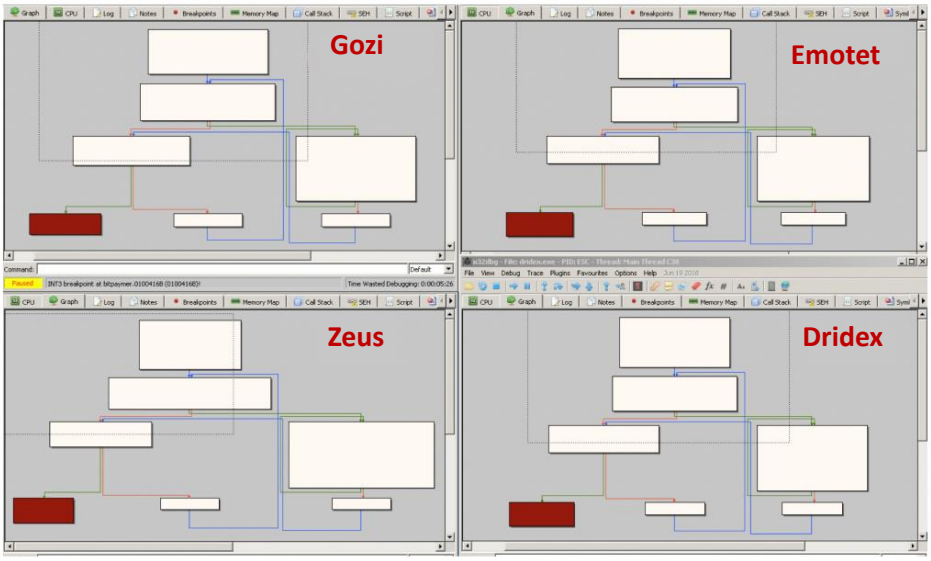


Figura 5.15. Struttura dei loader di Gozi, Emotet, Zeus e Dridex.

Inoltre, dalle analisi della struttura Fig 5.16, è emersa un'organizzazione identica in termini di dimensione del payload che viene utilizzata dal malware e di indirizzo di caricamento utilizzato a runtime (rispettivamente offset *0x34* e *0x38*).

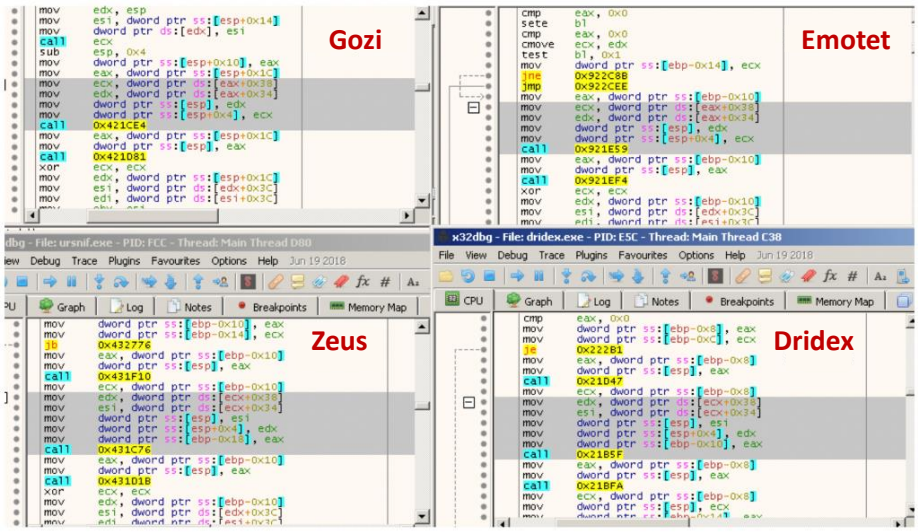


Figura 5.16. La struttura dati mostra dimensioni del payload utile e indirizzamento identici.

Tali similarità nel codice sono state evidenziate anche dai ricercatori di Trend Micro [30]. Essi sospettano che le organizzazioni criminali, che stanno dietro allo sviluppo delle quattro famiglie malware potrebbero essere in contatto tra di loro e scambiarsi informazioni e risorse. Tale ipotesi spiegherebbe le grandi somiglianze tra i campioni di queste famiglie.

Alla luce, di quanto detto in precedenza, in Tab. 5.4, si riportano i risultati ottenuti, suddivisi in totali e parziali. I risultati totali indicano i coefficienti di corretta e di errata classificazione ottenuti utilizzando tutte le famiglie di malware prese in considerazione. Mentre, i risultati parziali escludono il caso particolare delle famiglie *Gozi*, *Emotet*, *Zeus* e *Dridex*.

| Risultati | | | | |
|---|---|---|---|---|
| | totale | | parziale | |
| algoritmo | true positive | false positive | true positive | false positive |
| Kam1n0 | 0,53 | 0,47 | 0,58 | 0,42 |
| Genius | 0,82 | 0,18 | 0,85 | 0,15 |
| Gemini | 0,83 | 0,17 | 0,87 | 0,13 |
| Diaphora | 0,83 | 0,17 | 0,84 | 0,16 |
| SAFE | 0,79 | 0,21 | 0,82 | 0,18 |
| Machoc | 0,59 | 0,41 | 0,66 | 0,34 |

Tabella 5.4. Risultati di classificazione degli algoritmi.

I risultati parziali evidenziano un miglioramento del coefficiente di corretta classificazione per tutti gli algoritmi analizzati. Nelle Tab. 5.5, Tab. 5.6 e Tab. 5.7 si riportono i risultati suddivisi per le macro categorie analizzate: *Banking Trojan*, *Cryptojacking* e *Ransmware*. In questo caso gli algoritmi sono stati applicati ai soli sample appartenenti alla macro categoria di riferimenti. Tale analisi permette di evidenziare la percentuale di corretta classificazione tra campioni di famiglie appartenenti alla stessa macro categoria di interesse.

| Risultati dei Banking Trojan | | |
|---|---|---|
| algoritmo | true positive | false positive |
| Kam1n0 | 0,52 | 0,48 |
| Genius | 0,80 | 0,20 |
| Gemini | 0,81 | 0,19 |
| Diaphora | 0,82 | 0,18 |
| SAFE | 0,75 | 0,25 |
| Machoc | 0,55 | 0,45 |

Tabella 5.5. Risultati di classificazione relativi alla macro categoria Banking Trojan.

I risultati mostrano che i *Banking Trojan* sono la macro tipologia con la percentuale di errata classificazione più alta tra le famiglie costituenti. Ciò è indice di grosse analogie all'interno del codice di queste famiglie malware.

Invece le altre due macro categorie prese in considerazione, presentano delle caratteristiche bene delineate nella strattura e nell'organizzazione dei sample. Ciò permette di determinare con maggiore certezza la famiglia malware da associare.

| Risultati dei Cryptojacking | | |
|---|---|---|
| algoritmo | true positive | false positive |
| Kam1n0 | 0,69 | 0,31 |
| Genius | 0,88 | 0,12 |
| Gemini | 0,86 | 0,14 |
| Diaphora | 0,87 | 0,13 |
| SAFE | 0,80 | 0,20 |
| Machoc | 0,62 | 0,38 |

Tabella 5.6. Risultati di classificazione relativi alla macro categoria Cryptojacking.

| Risultati dei Ransomware | | |
|---|---|---|
| algoritmo | true positive | false positive |
| Kam1n0 | 0,65 | 0,35 |
| Genius | 0,96 | 0,04 |
| Gemini | 0,95 | 0,05 |
| Diaphora | 0,93 | 0,07 |
| SAFE | 0,83 | 0,17 |
| Machoc | 0,79 | 0,21 |

Tabella 5.7. Risultati di classificazione relativi alla macro categoria Ransomware.

# 6 Conclusioni

Il lavoro descritto nei capitoli precedenti, rappresenta una valutazione dello stato dell'arte delle tecniche di Binary Code Similarity e Binary Diffing applicate alla classificazione dei malware.

I risultati mostrano che gli approcci presi in considerazione, sono in grado di classificare un gran numero di sample, attribuendogli la corretta famiglia malware. In particolare si è stimato che il livello di precisione varia dal 56% al 83% in base alla tipologia di algoritmo utilizzati. Questi risultati evidenziano che gli algoritmi analizzati rappresentano un strumento aggiuntivo a supporto degli analisti di sicurezza. Nello specifico, sono in grado di classificare una grande quantità di campioni simultaneamente, garantendo così un grosso aiuto nell'analisi dei malware.

Tuttavia, bisogna sottilineare che gli algoritmi di machine learning non sono ancora in grado di sostituirsi ad un'attenta analisi, che risulta necessaria in numerosi casi. Infatti, dallo studio condotto l'intervento umano risulta fondamentale in diverse fasi del processo di apprendimento degli algoritmi.

In primo luogo, l'estrazione delle features dai binari è un passo rilevante per ottenere degli ottimi risultati, e spesso sono scelti sulla base dell'esperienza di esperti del settore. Un altro aspetto fondamentale, è la scelta dei campioni malevoli che costituiscono il dataset di training. Un dataset costituito da numerose varianti della stessa famiglia, permette di classificare con maggiore efficacia i campione sottoposti. Infine, l'analista deve validare i risultati generati dal motore di machine lerning. Questo ultimo passaggio è di estrama importanza, in quanto statisticamente l'introduzione di strumenti di machine learning aumenta di circa il 20% il numero di falsi positivi.

# A Il Formato Windows PE

Al fine di capire come estrapolare le informazioni salienti da un binario eseguibile, è necessario avere una comprensione di base della sua struttura. Questa sezione tratterà la struttura del formato eseguibile di Windows: il ***Portable Execution (PE)***.

Il formato PE risulta essere estremamente articolato nei sui campi, pertanto in tale sezione verranno trattati solo gli aspetti essenziali e i campi più significativi [1] utilizzati nella binary code analisys e nella malware analisys.

Una delle peculiarità più importanti dei file eseguibili è la caratteristica di essere rilocabili. Ciò significa che potrebbero essere caricati in memoria a un indirizzo virtuale differente ogni volta che viene mandato in esecuzione. Inoltre, oltre all'eseguibile principale, ogni programma ha un certo numero di file binari aggiuntivi caricati nel suo spazio di indirizzamento. Tali eseguibili aggiuntivi molto spesso sono ***Dynamic Link Library (DLL)*** che dipendono dalle funzionalità del programma stesso.

Poiché in ogni spazio degli indirizzi vengono caricati più eseguibili, risulta di fondamentale importanza assegnare ad ognuno di questi programmi uno spazio di indirizzamento virtuale. Sulla base di questo concetto il formato PE presenta al suo interno puntatori di tipo ***Relative Virtual Address (RVA)***. L'indirizzo RVA non è altro che un offset che viene aggiunto dal loader all'indirizzo di base del programma per ottenere i ***Virtual Address (VA)***.

Un'altra caratteristica dei file eseguibili è quella di essere divisi in sezioni. Ciò deriva dal fatto che la gestione della memoria di Windows è di tipo segmentata. La tecnica della segmentazione presuppone una suddivisione logica del programma che verrà eseguito in sezioni chiamati segmenti o sezioni. I segmenti sono necessari perché aree diverse del programma, vengono trattate in maniera diversa dal gestore della memoria al caricamento di un modulo.

Una divisione comune consiste nell'avere un segmento di codice (chiamato anche segmento di testo) ***.text*** contenente il codice dell'eseguibile e un segmento di dati ***.data*** contenente i dati dell'eseguibile. Al tempo di caricamento, il gestore della memoria imposta i diritti di accesso sulle pagine di memoria nei diversi segmenti. Questo determina se una determinata sezione è leggibile, scrivibile o eseguibile.

[1] Per un elenco completo dei singoli campi, è possibile visionare il sito MSDN di Microsoft.

Il formato PE può essere suddiviso in due aree principali: la parte relativa alle intestazioni chiamata ***Header File*** e la parte relativa ai segmenti chiamata ***Sections File***.

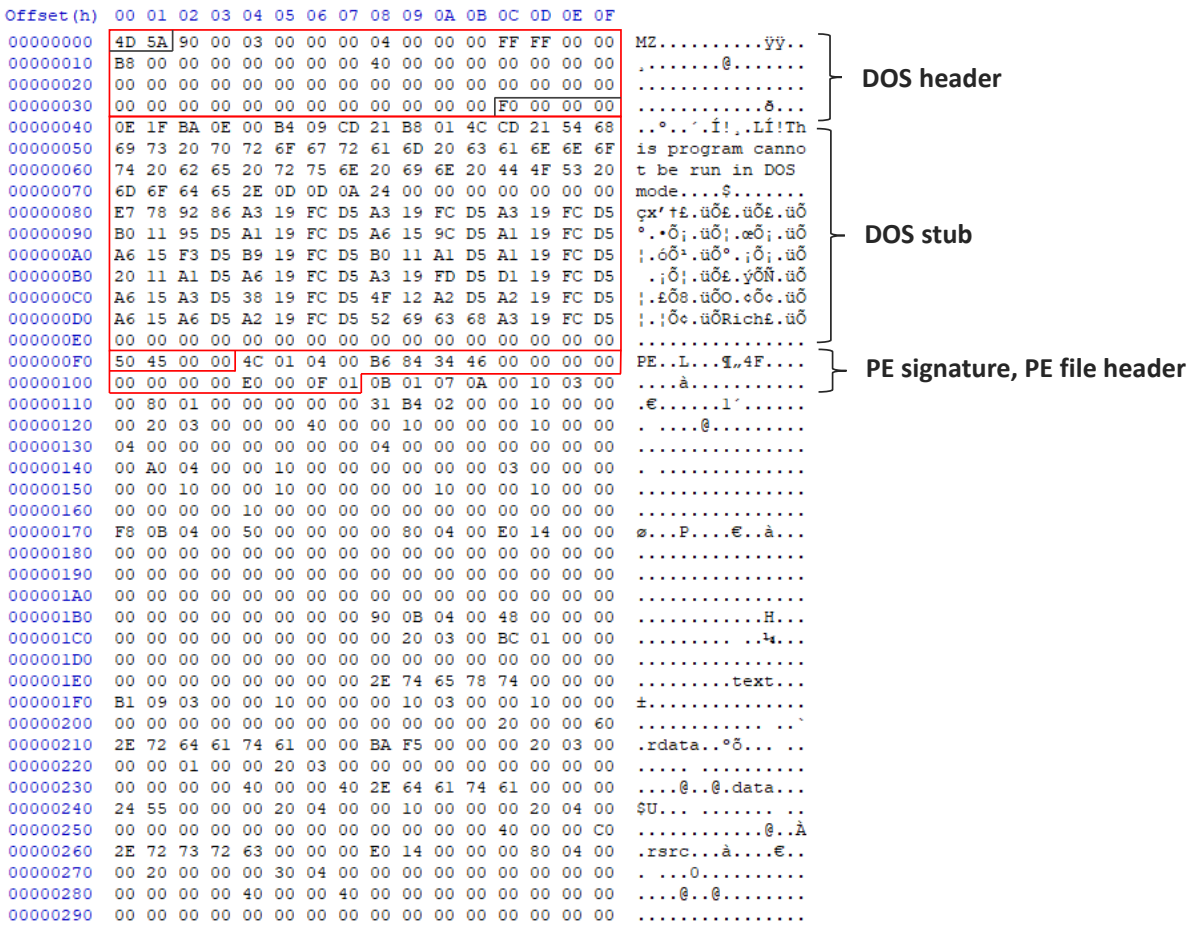


Figura A.1. Struttura del Formato Portable Execution (PE).

Qualsiasi file eseguibile presenta al suo interno quattro intestazioni differenti. I primi 64 byte di ogni file identifcano il ***DOS header***, caratterizzato dal magic code **"MZ"**[2] **(0x4D 0x5A)**. Ciò permette al sistema operativo di riconoscere questo file come un qualsiasi eseguibile (un .exe o .dll). La parte importante di questa intestazione è il puntatore al ***PE header*** presente all'offset 0x0C. Tra l'intestazione DOS e l'intestazione PE è presente il ***DOS stub***. Il DOS stub, nei Portable Execution, è un programma predefinito che contiene il codice necessario per stampare il messaggio ***"This program cannot be run in DOS mode"*** quando si tenta di eseguire tale programma in ambiente MS-DOS.

Il PE header è identificato dal magic code **"PE\0\0" (0x50 0x45 0x00 0x00)** e preseta una dimensione fissa di 24 byte. In questa intestazione sono presenti le seguenti informazioni sull'eseguibile:

[2] Iniziali dell'ingegnere di Microsoft Mark "Zibo" Joseph Zbikowski che brevettò il formato MS-DOS executable.

- ***Machine***: un numero che identifica la tipologia di processore per il quale il programma è stato progettato (*offset: 0, size: 2B*).
- ***NumberOfSections***: indica la dimensione della Section Table presente subito dopo l'intestazione dell'eseguibile (*offset: 2, size: 2B*).
- ***TimeDataStamp***: indica la data e l'ora di compilazione del programma (*offset: 4, size: 4B*).
- ***PointerToSymbolTable***: : indica il numero di voci nella tabella dei simboli COFF. Questo valore dovrebbe essere zero per un PE in quanto le informazioni di debug COFF sono deprecate (*offset: 12, size: 4B*).
- ***NumberOFSymbols***: indica l'offset del file all'interno della tabella dei simboli COFF o zero se tale tabella non è presente (*offset: 8, size: 4B*).
- ***SizeOfOptionalHeader***: indica la dimensione dell'intestazione opzionale (*offset: 16, size: 2B*).
- ***Characteristics***: i flag che indicano gli attributi del file (*offset: 18, size: 2B*).

L'intestazione PE è subito seguita da un'intestazione variabile, che non verrà trattata in quanto poco rilevanti ai fini dell'analisi. Tutte queste intestazioni sono definite nel Microsoft Platform SDK all'interno del file *WinNT.h*.

Al termine di tutte le intestazioni inizia la ***Section Table***. Tale tabella contiene per ogni riga il puntatore RVA alla sezione e la dimensione delle sezioni. Il numero di righe della tabella è indicato dal campo *NumberOfSections* del PE Header e sono numerati a partire da uno. La Section Table è seguita dalle intestazioni di sezione, una per ogni sezione del file. Le principali sezioni di un eseguibile sono:

- ***.data*** Initialized data
- ***.rdata*** Read-only initialized data
- ***.xdata*** Exception information
- ***.text*** Executable code
- ***.edata*** Export tables
- ***.idata*** Import tables

Per un elenco più dettagliato delle sezioni che si possono trovare all'interno di un eseguibile si rimanda al sito MSDN di Microsoft.

# B Strutture a Grafo

Prima di descrivere cos'è un grafo delle chiamate a funzione è opportuno riportare la definizione di grafo, sottografo e isomorfismo tra grafi.

**Definizione 7** *(Grafo). Un grafo G = (V, E) è costituito da un insieme di vertici (o nodi) V(G) e da un insieme di archi E(G). Gli archi definiscono una relazione tra vertici (u, v). Pertanto, tutti gli archi (u, v) ∈ E(G) se e solo se u,v ∈ V(G).*

**Definizione 8** *(Grafo Orientato). Un grafo G = (V, E) si definisce orientato se l'insieme degli archi E(G) sono un insieme di coppie ordinate.*

**Definizione 9** *(Sottografo). Un grafo H si definisce sottografo del grafo G se e solo se |V(H)| ≤ |V(G)| ed esisteste una funzione iniettiva f : V(H) ↦ V(G), tale che per tutti gli archi (u, v) ∈ E(H) risulta verificata la seguente relazione (f(u),f(v)) ∈ E(G).*

**Definizione 10** *(Isomorfismo). Un grafo H è isomorfico con il grafo G se e solo se |V(H)| = |V(G)| ed esiste una funzione biiettiva f : V(H) ↦ V(G), tale che qualsiasi arco (u, v) ∈ E(H) verifica la seguente relazione (f(u), f(v)) ∈ E(G).*

I grafi possono essere suddivisi in: *grafi semplici* e *grafi multipli*. I *grafi semplici* hanno la caratteristica di avere al massimo un arco che collega due nodi, mentre i *grafi multipli* permettono di rappresentare più archi tra due coppie di nodi.

Un grafo può essere *etichettato*. Una funzione di etichettatura può essere definita come L : S ↦ Z, dove Z è l'insieme di etichette, S è un sottoinsieme di nodi di V(G). Un insieme di etichette può essere rappresentato da qualsiasi tipo di oggetto (tipicamente sono interi positivi, colori o stringhe). Inoltre, le etichette possono essere associati anche agli archi, in questi casi solitamente vengono definiti come pesi.

Un path in un grafo non orientato G è una seguenza di archi ($e_1$, $e_2$, ..., $e_n$), ∀ $e_i$ ∈ E(G), i ∈ {1, 2, ..., n} che connette una seguenza di vertici ($v_1$, $v_2$, ..., $v_n$, $v_{n+1}$), dove ∀ $v_i$ ∈ V(G), $e_j$ = ($v_j$, $v_{j+1}$), j ∈ {1, 2, ..., n}. La lunghezza del path è rappresentata dal numero di archi che contiene.

# Bibliografia


[1] A. Schulman, *Finding binary clones with opstrings and function digests*, Dr Dobb's Journal-Software Tools for the Professional Programmer, 2005.

[2] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, Z. Su*Detecting Code Clones in Binary Executables*, University of California, Indiana University, Lawrence Livermore National Laboratory, 2016

[3] T. Dullien, R. Rolles, *Graph-based comparison of executable objects*, SSTIC, 2005.

[4] H. Chen, *The Influences of Compiler Optimization on Binary Files Similarity Detection*, PhD thesis. School of Computer Science and Technology Shandong Yingcai University, Jinan, China, 2014

[5] M. Jurczyk, *Using Binary Diffing to Discover Windows Kernel Memory Disclosure Bugs*, Google Project Zero, 2017.

[6] J. Oh ,*Fight against 1-day exploits: Diffing Binaries vs Anti-diffing Binaries*, Blackhat USa, 2009.

[7] M. Charikar. *Similarity estimation techniques from rounding algorithms*. In Proceedings on 34th Annual ACM STOC'02, 2002.

[8] S. Har-Peled, P. Indyk, and R. Motwani. *Approximate nearest neighbor*, "Towards removing the curse of dimensionality. Theory of Computing", 2012.

[9] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. *Locality-sensitive hashing scheme based on p-stable distributions*. In Proc. of ACM SoCG'04, 2004.

[10] P. Yosifovich, A. Ionesco, M. E. Russinovich, D. E. Solomon, *Windows Internals*, "System architecture, processes, threads, memory managment and more", vol. 1, pp. 45-99, 2017.

[11] E. Eilam, *Reversing*, "Secrets of Reverse Engineering", pp. 93-103, 2005.

[12] C. Eagle. *The IDA Pro Book*, "The unofficial Guide to the World's Most Popular Disassembler", pp.59-98, pp.415-499 2008.

[13] A. Rousseau, R. Seymour, *Xori*, "Finding Xori, Malware Analysis Triage with Automated Disassembly", https://sites.google.com/secured.org/malwareunicorn/xori, Blackhat2018, Defcon 26, USA, 2018.

[14] K. Zakkahttps, *A Complete Guide to K-Nearest-Neighbors with Applications in Python and R*, Kevin's Blog https//kevinzakka.github.io/2016/07/13/k-nearest-neighbor/, 2016.

[15] T. Liu, A. W. Moore, A. Gray and K. Yang, *Fast Approximate Nearest Neighbors With Automatic Algorithm Configuration*, Computer Science Department, University of British Columbia, Vancouver, B.C., Canada

[16] T. Liu, A. W. Moore, A. Gray and K. Yang, *An Investigation of Practical Approximate Nearest Neighbor Algorithms*, School of Computer Science Carnegie-Mellon University, Pittsburgh, USA, 2016.

[17] S. H. H. Ding, B. C. M. Fung, P. Charland, *Kam1n0: MapReduce-based Assembly Clone Search for Reverse Engineering*, Research Article. School of Information Studies, McGill University, Montreal, QC, Canada, 2016.

[18] W. M. Khoo, A. Mycroft, and R. J. Anderson. *Rendezvous: a search engine for binary code.* 2013.

[19] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, H. Yin, *Scalable Graph-based Bug Search for Firmware Images* Research Article. Air Force Research Lab,University of California, Riverside, Department of EECS, Syracuse University, USA, 2016.

[20] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, D. Song, *Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection*, Research Article. Shanghai Jiao Tong University, University of California, Berkeley, University of California, Riverside, Georgia Institute of Technology, Samsung Research America, 2017.

[21] J. Koret, *Diaphora - An IDA Python BinDiffing plugin* User Guide: https://github.com/joxeankoret/diaphora/blob/master/doc/diaphora_help.pdf, 2015.

[22] L. Massarelli, G. A. Di Luna, F Petroni, L. Querzoni, R. Baldoni, *SAFE: Self-Attentive Function Embeddings for Binary Similarity*, Research Article. University of Rome Sapienza, CINI, National Laboratory of Cyber Security, 2018.

[23] Z. Lin, M. Feng, C. Nogueira dos Santos, M. Yu, B. Xiang, B. Zhou, and Y. Bengio, *A structured self-attentive sentence embedding* Research Article. Montreal, Institute for Learning Algorithms (MILA), Universite de Montreal, 2017.

[24] S. L. Berre, A. Chevalier, T. Pourcelot, *Démarche d'analyse collaborative de codes malveillants*, Research Article. Agence Nationale de la Sécurité des Systèmes d'Informations (ANSSI), 2017.

[25] Proofpoint, *Quarterly Threat Report*, Q1 report: https://www.proofpoint.com/sites/default/files/pfpt-us-tr-q118-quarterly-threat-report.pdf, 2018.

[26] Proofpoint, *Quarterly Threat Report*, Q1 report: https://www.proofpoint.com/sites/default/files/pfpt-us-tr-q218-quarterly-threat-report.pdf, 2018.

[27] M. Sebastiàn, R. Rivera, P. Kotzias, J. Caballero, *AVCLASS: A Tool for Massive Malware Labeling*, Research Article. IMDEA Software Institute, Universidad Politecnica de Madrid, 2016.

[28] F. Skulason, A. Solomon, *CARO Virus Naming Convention.* http://www.caro.org/articles/naming.html, 1991.

[29] D. Beck, J. Connolly, *The Common Malware Enumeration Initiative*,Virus Bulletin Conference, 2006.

[30] Trend Micro, *Exploring Emotet: Examining Emotet's Activities, Infrastructure*, https://blog.trendmicro.com/trendlabs-security-intelligence/exploring-emotet-examining-emotets-activities-infrastructure/

# Ringraziamenti

Dopo 4 anni dall'iscrizione al Politecnico di Torino, finalmente il giorno è arrivato: scrivere queste frasi di ringraziamento è il momento conclusivo della mia tesi. È stato un periodo di profondo apprendimento e cambiamento. Vorrei spendere due parole di ringraziamento nei confronti di tutte le persone che mi hanno sostenuto e aiutato durante questo periodo.

In primo luogo, desidero ringraziare il relatore di questa tesi, il professore Giovanni Squillero, e il corelatore Andrea Marcelli, per la disponibilità, la professionalità e la gentilezza dimostrate durante la stesura del lavoro.

Un doveroso ringraziamento va ai miei genitori, perché mi sono sempre stati accanto e non mi hanno mai fatto mancare il loro sostegno durante questi anni di università. Grazie papà Vincenzo e mamma Maria.

Un grazie sincero va ai miei amici di vecchia data Marco, Santi, Stefano e Toti, che mi hanno dimostranto che il sostegno dei veri amici supera anche le distanze. Ringrazio di cuore anche Enrico e Simone per essermi stati vicino nelle ultime fasi di questo percorso di studio (non sempre semplice).

Un ringraziamento speciale va al collega di lavoro Max (The Wizard) per avermi trasferito la passione per la sicurezza informatica e l'entusiasmo nell'imparare ogni giorno qualcosa di nuovo.

Infine, un ultimo ringraziamento voglio farlo a me stesso. Perché nei momenti difficili ho trovato la forza per andare avanti, anche quando ero convinto di non farcela. Questi ringraziamenti voglio che siano di incoraggiamento per il mio futuro.