Università degli Studi di Milano - Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di Laurea in Informatica

Piattaforma web per la ricerca automatica di vulnerabilità in file binari

Relatore: Prof. Claudio Ferretti

Correlatore: Dott.sa Martina Saletta

Tesi di Laurea di:
Andrea Consonni
Matricola 900116

Anno Accademico 2024-2025

Vorrei esprimere i miei più sentiti ringraziamenti al ***prof. Claudio Ferretti****, alla* ***Dott.sa Martina Saletta*** *e al* ***prof. Giovanni Denaro*** *per il costante supporto, la disponibilità e i preziosi consigli offerti durante tutto il percorso di tesi.*

Un ringraziamento speciale va alla ***mia famiglia****, che con il suo supporto incondizionato mi ha permesso di affrontare con serenità questo percorso di studi.*

Sono profondamente grato anche ***a tutti i miei amici****, il quale costante sostegno mi ha accompagnato e incoraggiato lungo tutto il percorso universitario*

Abstract

La sicurezza informatica è sempre più centrale nello sviluppo software, soprattutto quando si analizzano applicazioni senza accesso al codice sorgente. Questa tesi presenta Binoculars, una piattaforma web pensata per semplificare l'analisi di sicurezza di file binari ELF per architettura x86. Basata su angr, Flask e SvelteKit, la piattaforma offre un'interfaccia intuitiva che consente anche ad analisti non specialisti di effettuare una prima valutazione automatizzata, facilitando l'identificazione di potenziali vulnerabilità.

Indice

Elenco delle figure

Listings

Capitolo 1

Introduzione

In un mondo sempre più digitalizzato ed interconnesso, la tematica della sicurezza informatica ha assunto sempre più un'importanza chiave in ogni processo di sviluppo software. La potenziale presenza e lo sfruttamento di una vulnerabilità all'interno di un'applicazione da parte di un'attaccante potrebbe avere conseguenze disastrose: dall'escalation di privilegi all'accesso non autorizzato a dati sensibili, compromettendo quindi l'integrità e la confidenzialità di quest'ultimi. È quindi fondamentale che i potenziali rischi per la sicurezza siano considerati sin dai primi momenti del processo di sviluppo. Effettuare un'analisi di sicurezza approfondita risulta quindi fondamentale nell'evitare che potenziali vulnerabilità persistano all'interno del programma; tuttavia, questo processo si complica notevolmente quando l'analista è in **solo possesso del file binario** e non ha accesso al codice sorgente dell'applicazione. In questo caso, l'analista non solo dovrà avere ampie competenze specifiche in ambito di reverse engineering, ma dovrà essere in grado di utilizzare tool e framework che potrebbero avere un'interfaccia a primo impatto ostica, richiedere conoscenze di scripting o di tematiche di sicurezza avanzate oppure avere un costo elevato, il quale potrebbe non rientrare nei limiti di budget prefissati. Questa tesi propone l'implementazione di una piattaforma web per l'analisi di file binari denominata **Binoculars**; la quale si prefigge l'obbiettivo di semplificare il processo di analisi di sicurezza su file binari ELF compilati per architettura x86 tramite un'interfaccia semplice ed intuitiva, permettendo anche ad analisti con competenze di sicurezza non specialistiche di effettuare una prima valutazione del programma, la quale potrà poi essere approfondita tramite analisi più specifiche. La piattaforma si basa su **angr**, un toolkit open-source multi-architettura per l'analisi binaria, per eseguire automaticamente diverse tipologie di analisi statiche e dinamiche, sul framework python **Flask** per l'implementazione di una REST API progettata per comunicare i risultati dell'analisi e sul framework javascript **SvelteKit**, il quale si occupa della strutturazione delle pagine web della piattaforma e della presentazione dei risultati dell'analisi all'utente.

1.1 Differenza tra debolezza e vulnerabilità

Spesso il termine "vulnerabilità" è utilizzato per riferirsi ad una qualsiasi problematica di sicurezza all'interno del software sotto analisi. Tuttavia, è fondamentale distinguere il concetto di **vulnerabilità** da quello di **debolezza**. Per delineare con precisione questa distinzione, adotteremo le definizioni fornite dal glossario compilato dal MITRE [1]:

- **Debolezza**: Una condizione nel software, firmware, hardware o in una componen-

te di servizio che, sotto certe circostanze, potrebbe contribuire all'introduzione di vulnerabilità

- **Vulnerabilità**: Un errore nel software, firmware, hardware o componente di servizio **derivante dalla presenza di una debolezza** che può essere sfruttata da un'attaccante, causando un impatto negativo sull'integrità, la confidenzialità e la disponibilità dei componenti impattati

Una vulnerabilità è quindi **un'istanza sfruttabile di una debolezza**. Per riferirci alle categorie di difetti che le tecniche di analisi automatica offerte dalla piattaforma sono in grado di rivelare, questa tesi adotterà la tassonomia **Common Weakness Enumeration** (CWE), anch'essa compilata dal MITRE.

1.2 Struttura della relazione

La relazione è articolata nei seguenti capitoli:

- **Capitolo 2: Stato dell'arte**: Questo capitolo presenta una rassegna di alcune tecniche, metodologie e soluzioni esistenti per l'analisi di file binari. Verrà evidenziato l'approccio adottato per affrontare il problema della ricerca di vulnerabilità e i rispettivi limiti di ogni soluzione presentata.
- **Capitolo 3: Metodologie utilizzate**: Questo capitolo discute i fondamenti teorici che costituiscono la base delle analisi implementate dalla piattaforma. Saranno discussi in dettaglio sia i concetti di **disassembling** e **decompiling** sia le metodologie di analisi statica e dinamica utilizzate per effettuare la ricerca delle vulnerabilità. Verranno inoltre forniti degli esempi per illustrarne il funzionamento.
- **Capitolo 4: Tecnologie utilizzate**: Questo capitolo presenta in dettaglio le tecnologie e i framework scelti per l'implementazione della piattaforma. Saranno presentati sia i componenti del backend sia le tecnologie adottate per lo sviluppo del frontend.
- **Capitolo 5: Analisi implementate**: Questo capitolo illustra nel dettaglio le analisi implementate all'interno della piattaforma. Verrà descritto come ciascuna tecnica di analisi porti al rilevamento di una vulnerabilità e verrà fornita una lista comprensiva di tutte le debolezza software che ogni tecnica è capace di rilevare.
- **Capitolo 6: Architettura della soluzione**: Questo capitolo descrive l'architettura generale della piattaforma Binoculars. Verrà illustrato il modello architetturale della soluzione, illustrando le interazioni fra i vari componenti e come essi collaborano per presentare all'utente il risultato dell'analisi richiesta
- **Capitolo 7: Sperimentazione** Questo capitolo presenta le varie sperimentazioni effettuate sulla piattaforma al fine di validarne l'accuratezza. Per ogni tecnica di analisi implementata, verranno presentati i programmi che sono stati utilizzati al fine di validare l'efficacia e l'accuratezza dell'analisi e i risultati prodotti da quest'ultima
- **Capitolo 8: Conclusioni**: Questo capitolo presenterà le conclusioni finali del lavoro. Saranno inoltre esposte le limitazioni e le problematiche incontrate durante l'implementazione della piattaforma e i suoi possibili sviluppi futuri

Capitolo 2

Stato dell'arte

Questo capitolo tratta una rassegna di alcune metodologie, tecniche e soluzioni attualmente disponibili per risolvere il problema della ricerca automatica di vulnerabilità in file binari. Verranno in particolare approfonditi alcuni approcci basati su analisi statica, analisi dinamica e su tecniche di apprendimento automatico. Per ciascuna metodologia presentata, verranno dettagliati il suo funzionamento generale, le sue capacità di analisi e le sue limitazioni

2.1 Metodologie basate su tecniche di analisi statica

L'analisi statica di un programma consiste in un'insieme di metodologie, tool e algoritmi che permettono l'analisi del codice sorgente o della sua rappresentazione binaria (per esempio, un file eseguibile) senza che il programma venga effettivamente eseguito [2]. Questa tecnica è ampiamente adottata nell'ambito della ricerca delle vulnerabilità, in quanto consente di inferire e determinare se certe proprietà sono soddisfatte (per esempio, le condizioni che possono portare ad una certa vulnerabilità) senza direttamente eseguire il programma. Tuttavia, l'analisi statica condotta direttamente su un file binario è intrinsecamente più complessa rispetto all'analisi statica del codice sorgente: le principali difficoltà risiedono nella mancanza di informazioni riguardante i tipi e la struttura ad alto livello del codice [3] e nella necessità di gestire e rappresentare adeguatamente le operazioni riguardanti la memoria [4]. Nonostante queste sfide, nel corso degli anni sono stati sviluppati diversi approcci e metodologie di analisi statica progettati per effettuare la ricerca di vulnerabilità all'interno di file binari. Queste tecniche, tuttavia, possono produrre un elevato numero di falsi positivi e falsi negativi : poiché non effettuano un'esecuzione concreta del programma, esse devono effettuare diverse assunzioni sul suo stato a runtime. Ciò potrebbe quindi portare i tool basati su questa tipologia di analisi a segnalare vulnerabilità in porzioni di programma non vulnerabili.

2.1.1 Taint analysis statica

La *taint analysis* (o *taint checking*) è una tecnica di analisi che mira a tracciare e monitorare la propagazione di flussi di dati inaffidabili o potenzialmente dannosi all'interno del programma. La taint analysis si compone di tre elementi chiave:

1. **Sorgenti** (Sources): Sono i punti del programma dove si origina un flusso di dati inaffidabile. Una sorgente potrebbe per esempio essere l'input di un utente oppure i dati letti da un file.

2. **Propagazione**: Viene effettuato un monitoraggio continuo della propagazione nel programma dei dati provenienti da una sorgente

3. **Sink**: Sono i punti del programma che effettuano operazioni sensibili, come per esempio l'accesso al filesystem o la chiamata ad operazioni di libreria non sicure.

Una possibile vulnerabilità verrà quindi rilevata quando il programma permette ad un dato "tainted" di raggiungere un sink; ciò può avvenire quando, per esempio, il dato non viene adeguatamente sanificato.
Bintaint è un tool di parsing capace di effettuare taint analysis statica su file binari [5]. Il taint analyzer proposto è basato sul tool commerciale di reverse engineering *IDA*, il quale viene utilizzato per recuperare il codice assembly dal codice binario, ed è implementato utilizzando il linguaggio funzionale *OCaml*. Bintaint è composto da quattro moduli distinti:

- **Decoder module**: Questo modulo si occupa di tradurre il codice assembly recuperato da IDA in una rappresentazione in un linguaggio intermedio chiamato *REIL* (Reverse Engineering Intermediate Language), le quali espressioni verranno a loro volta convertite in espressioni simboliche.

- **Taint Processing Configuration Module**: Questo modulo gestisce la configurazione per l'inizializzazione della taint analysis, leggendo la configurazione fornita dall'utente in formato XML, la quale dovrà contenere tutte le informazioni necessarie per effettuare la taint analysis. Questo modulo si occupa inoltre di stabilire una relazione tra l'input esterno e le varie sorgenti definite

- **Expression Parsing Module**: Questo modulo si occupa di definire come avviene la propagazione dei flussi di dati tainted all'interno del programma

- **TCFG Generation Module**: Questo modulo si occupa di generare una struttura a grafo diretta chiamata *Taint Control Flow Graph*, la quale rappresenterà tutte le possibili aree del programma che un determinato flusso tainted può raggiungere. L'analisi del TCFG permetterà quindi di evincere se un determinato sink dipende dai dati generati da una determinata sorgente.

L'approccio proposto dal tool permette di ridurre il numero di falsi positivi e falsi negativi rilevati rispetto ad una tain analysis tradizionale; inoltre, l'utilizzo del linguaggio intermedio REIL permette al tool di essere facilmente integrabile in sistemi di analisi più complessi, a patto che anch'essi utilizzino lo stesso linguaggio di rappresentazione intermedia. Tuttavia il tool risulta comunque dipendente dall'input dell'analista; l'accuratezza dell'analisi dipenderà quindi dalla corretta definizione di sorgenti, sink e propagazione da parte di quest'ultimo. Infine, Bintaint si basa sul framework commerciale IDA, il quale non offre tutte le sue funzionalità nella sua versione gratuita.

2.1.2 Binary Code Similarity Detection

Quando l'obbiettivo dell'analisi è ricercare una vulnerabilità nota (per esempio, una debolezza già documentata), è possibile adottare una strategia chiamata *Binary Code Similarity Detection* (BCSD). Questo approccio si basa sul confronto il codice binario del programma in esame con la firma (il codice binario) della vulnerabilità. Se l'algoritmo di analisi rileva segmenti di codice con un elevato grado di somiglianza con la firma della

vulnerabilità, allora è altamente probabile che il programma contenga quella vulnerabilità. Un algoritmo di decisione determinerà se il programma contiene effettivamente la vulnerabilità. Tuttavia, poiché il codice contente la vulnerabilità spesso richiede solo piccole modifiche per fare in modo che esso non sia più vulnerabile (l'aggiunta di un controllo, l'impostazione di permessi aggiuntivi, ...), il codice binario del programma corretto e il programma originale saranno molto simili; potenzialmente portando l'analizzatore a segnalare dei falsi positivi [6].
VulneraBin [7] è un tool che effettua un'analisi BCSD attraverso una metrica di similarità basata su hashing, strutturando il processo nelle seguenti fasi:

1. **Re-ottimizzazione del linguaggio di rappresentazione intermedia (IR)**: Il codice assembly viene dapprima tradotto nel linguaggio di rappresentazione intermedia VEX-IR, il quale utilizzo mira ad appiattire le eventuali differenze sintattiche derivanti dall'utilizzo di registri diversi, istruzioni diverse per l'assegnamento o metodologie di ottimizzazione introdotte dai vari compilatori. Successivamente, viene applicata un'ulteriore ottimizzazione sul codice intermedio per eliminare le differenze residue che potrebbero ancora persistere a causa delle diverse tecniche di ottimizzazione dei compilatori.

2. **Program Slicing**: Il program slicing è una tecnica di anali statica che, partendo da un sottoinsieme dei comportamenti di un programma, ne produce una versione minimale, chiamata "slice", la quale mantiene esattamente lo stesso sottoinsieme di comportamenti. Poiché questa tecnica è alla base delle analisi offerte dalla piattaforma, verrà ulteriormente approfondita nel *Capitolo 3* di questa tesi.

3. **Strand normalization**: Una "strand" è definita come l'insieme di istruzioni contigue richieste per computare il valore di una specifica variabile [8]. Gli strand vengono normalizzati rinominando i registri utilizzati durante le varie operazioni, andando così ad eliminare eventuali differenze sintattiche introdotte dai compilatori

4. **Similarity evaluation**: Viene effettuato un confronto fra gli hash MD5 calcolati sugli strand normalizzati e gli hash delle vulnerabilità contenute in un database. Se la similarità supera una certa soglia definita manualmente, allora il binario sarà considerato vulnerabile.

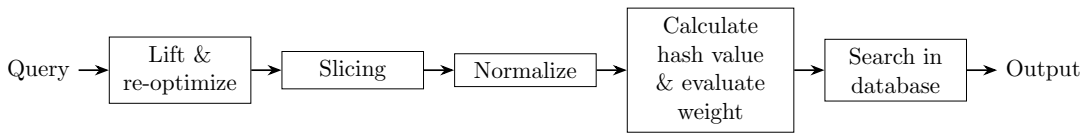

Figura 2.1: Schema di funzionamento di VulneraBin. Immagine proveniente da [7]

Nonostante l'approccio proposto porti ad un miglioramento della complessità computazionale dell'analisi e alla mitigazione del numero di falsi positivi e negativi rilevati, l'affidabilità dell'analisi rimane comunque legata ad una soglia scelta manualmente dall'analista. Sarà quindi necessario che quest'ultimo imposti una soglia ottimale per ogni specifico binario o classe di vulnerabilità, compromettendo quindi l'automazione del processo.

2.2 Metodologie basate su tecniche di analisi dinamica

L'analisi dinamica consiste nell'osservazione del comportamento di un programma mentre esse viene eseguito in un determinato ambiente d'esecuzione. Per consentire questo tipo di analisi, i tool che implementano questo tipo di tecniche devono effettuare un processo chiamato *instrumentation*, il quale consiste nell'aggiungere codice di analisi all'interno del programma da analizzare in modo tale che venga eseguito insieme a quest'ultimo senza modificarne il normale flusso di esecuzione [9]. I risultati ottenuti tramite l'analisi dinamica sono generalmente più precisi rispetto ai risultati ottenuti effettuando un'analisi statica del programma, poiché non vi è più la necessità di effettuare un'astrazione riguardo i valori computati o il cammino intrapreso dal programma sotto analisi. Tuttavia, poiché l'esecuzione concreta di un programma richiede la scelta di un insieme di input concreti con il quale eseguirlo, i risultati ottenuti tramite queste tecniche non sono generalizzabili, in quanto l'insieme di input scelto potrebbe non essere rappresentativo di tutti i possibili cammini d'esecuzione del programma [10].

2.2.1 Fuzzing

Il fuzzing è una tecnica di analisi dinamica che consiste nell'osservare il comportamento del programma quando esso riceve degli input casuali o malformati. Se un input provoca un blocco dell'esecuzione o un crash, allora il programma potrebbe allora contenere una problematica di implementazione oppure una debolezza software, la quale, sotto certe circostanze, potrebbe risultare sfruttabile da un potenziale attaccante. Questa tecnica viene implementata attraverso programmi specializzati, chiamati *fuzzers*; un esempio noto è **American Fuzzy Lop** (AFL). Generalmente, i principali componenti del fuzzing (e di un fuzzer) sono[11]:

- **Programma obbiettivo**: Il programma da analizzare, il quale può essere rappresentato sia dal suo codice binario sia dal suo codice sorgente. Poiché l'accesso a quest'ultimo è a volte ostico in situazioni reali, i software di fuzzing hanno spesso come programma obbiettivo il solo codice binario.
- **Monitor**: Raccoglie informazioni riguardanti l'esecuzione del programma.
- **Input generator**: Si occupa della generazione degli input, la quale può avvenire in due modi distinti:
  - **Grammar-based**: Gli input vengono generati utilizzando una grammatica
  - **Mutation-based**: Gli input vengono generati usando dei file seed, i quali vengono mutati casualmente oppure utilizzando delle strategie di mutazione ben definite.
- **Bug detector**: Quando il programma va in crash o riporta degli errori, questo modulo recupera e analizza le informazioni rilevanti per determinare se vi è la presenza di un "bug" (una debolezza, una vulnerabilità, ...).
- **Bug filter**: Non tutti i "bug" sono effettivamente delle vulnerabilità; è quindi necessaria un'operazione di filtraggio per scartare tutte quelle problematiche che non risultano sfruttabili da un attaccante.

Inoltre, le tecniche di fuzzing possono essere divise in tre categorie [12]:

- **White-box fuzzing**: In questo tipo di fuzzing, si assume di avere accesso al codice sorgente del programma; la maggior parte delle informazioni per generare l'input viene quindi acquisita tramite l'analisi del codice sorgente
- **Black-box fuzzing**: Nel fuzzing black-box si effettua il fuzzing sul programma senza avere nessuna informazione sulla sua struttura interna
- **Gray-box fuzzing**: Questo tipo di fuzzer effettuano un'analisi del programma (come taint analysis o tramite instrumentation) per ottenere le informazioni sulla struttura interna di quest'ultimo

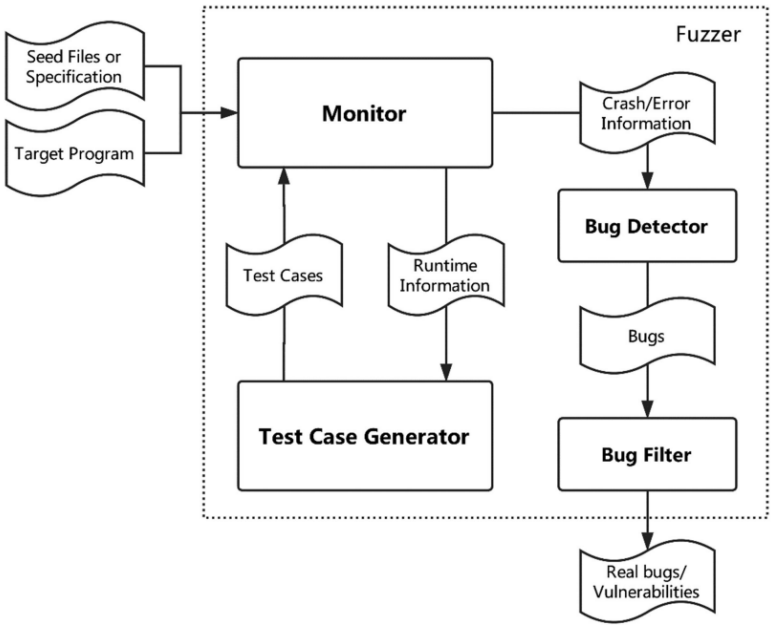

Figura 2.2: Funzionamento generale di un fuzzer. Immagine proveniente da [11]

Seppur sia una tecnica efficiente e ben conosciuta per effettuare l'analisi di un programma, il fuzzing risente di diverse problematiche, come la necessità, nei fuzzer gray-box e black-box, di generare un input che passi i controlli di sanificazione del programma senza avere informazioni su quest'ultimo, permettendo così un'analisi più approfondita del programma. Altra problematica è quella legata alla definizione, nei fuzzer mutation-based, di una buona tecnica di mutazione dell'input, in modo da poter analizzare il maggior numero possibile di cammini di esecuzione interessanti. Per i tool di fuzzing, quindi, la problematica principale da superare è quella di **implementare una buona strategia di generazione e mutazione dell'input**, in modo tale che essa permetta all'analisi di essere il più approfondita possibile.

2.3 Tecniche basate su modelli di apprendimento automatico

Negli ultimi anni, la ricerca nell'ambito dell'intelligenza artificiale ha compiuto enormi progressi, portando i modelli disponibili ad essere sempre più accurati ed efficienti. Questo

rapido avanzamento ha avuto un impatto significativo anche nel campo della sicurezza informatica, dove le capacità predittive dell'intelligenza artificiale possono essere sfruttate l'identificazione automatica di vulnerabilità. Le tecnica di apprendimento automatico possono dividere in base a come avviene l'addestramento del modello:

- **Apprendimento supervisionato**: Si sviluppa un modello predittivo tramite l'addestramento su dati etichettati.
- **Apprendimento non supervisionato**: Il modello viene applicato su un'insieme di dati non etichettati con lo scopo di trovare una qualche struttura intrinseca del dataset
- **Apprendimento per rinforzo**: Il modello apprende come raggiungere un dato obbiettivo, ricevendo una ricompensa o una penalità in base a quanto al scelta che ha compiuto lo avvicina all'obbiettivo

In generale, le tecniche di machine learning per la ricerca di vulnerabilità prevedono l'estrazione di feature significativo dal file binario sotto analisi; le quali verranno poi codificate in un formato idoneo e utilizzate dal modello per l'identificazione di potenziali percorsi di esecuzione vulnerabili [13]. Sono state proposti diversi approcci basati su machine learning, per esempio *Aumpansub & Huang* [14] propongono di estrarre le informazioni sintattiche dal codice assembly e di addestrare due modelli per il riconoscimento, mentre *Li et al.* [15] propongono invece di usare come input di addestrare una rete neurale al riconoscimento di vulnerabilità tramite l'utilizzo di tracce di esecuzione del programma ottenute tramite fuzzing. La ricerca automatica di vulnerabilità in file binari tramite machine learning è un campo relativamente nuovo e, come tale, soffre di alcune problematiche [16]:

- **Mancanza della struttura ad alto livello del codice**: Come per l'analisi statica, la mancanza di informazioni sui tipi o sulle funzioni chiamate rende difficile l'applicazione di questo tipo di tecniche
- **Selezione delle feature**: È necessario definire quali sono le feature rilevanti e sviluppare una metodologia per estrarle
- **Selezione del modello**: Bisogna selezionare un modello che permetta di ottenere un grado accettabile di accuratezza. Questo compito è reso particolarmente difficile dal fatto che diversi modelli possono ottenere un'accuratezza comparabile a parità di analisi da effettuare.

2.4 Tecniche ibride

I vari approcci all'analisi di sicurezza presentati fino ad ora non devono essere pensati come insiemi disgiunti. Infatti, la combinazione di queste tecniche è una pratica estremamente diffusa e proficua, visto che permette di controbilanciare i punti deboli di ciascuna metodologia e di ottenere risultati più accurati. Per esempio, combinare tecniche di analisi statica e dinamica permette di mitigare il numero di falsi positivi ottenuti dalla prima effettuando un'esplorazione mirata tramite la seconda. Oppure le tecniche di analisi statica e dinamica possono essere usate per ottenere ed estrarre le feature necessarie all'addestramento del modello di riconoscimento (come abbiamo già visto con DeepVL [15]).

Capitolo 3

Metodologie utilizzate

Questo capitolo è dedicato all'esposizione delle diverse metodologie statiche e dinamiche alla base delle tecniche di analisi rese disponibili dalla piattaforma. In particolare, verranno illustrati i concetti teorici alla loro base e verranno forniti esempi per illustrare il funzionamento di alcune tecniche su casi concreti.

3.1 Control Flow Graph (CFG)

Considerare adeguatamente il flusso di controllo di un programma, cioè quali istruzioni vengono eseguite dato un certo input, è fondamentale per effettuare un'analisi di sicurezza accurata. Risulterebbe infatti inutile segnalare una problematica di sicurezza data da un segmento irraggiungibile del codice di un programma. Possiamo notare che quando vi sono due istruzioni in sequenza, l'esecuzione della prima implica l'esecuzione della seconda. Chiamiamo quindi *basic block* una **sequenza massimale contigua di statement del programma**. Per rappresentare in modo esaustivo tutti i possibili cammini di esecuzione che un programma può intraprendere, possiamo ricorrere ad una struttura a grafo chiamata **Control-Flow Graph** (CFG). Dato un programma P, un CFG per P è un grafo G **diretto e orientato** dove:

- I nodi di G sono i basic block del programma
- Gli archi di G connettono i basic block che sono in una relazione di sequenza (uno segue l'altro). Gli archi che derivano da una scelta condizionale (es. *if*) sono etichettati con "true" e "false"

Durante la costruzione di un CFG, è importante gestire correttamente le istruzioni che modificano il flusso di controllo, come *if* e *while*:

- **If**: Il controllo condizionale termina il basic block a cui appartiene lo statement immediatamente precedente. Due archi etichettati "true" e "false" connettono il basic block contenente la condizione rispettivamente ai rami *then* e *else*. Gli archi uscenti dai basic block dei due rami sono diretti verso il basic block contenente gli statement che seguono l'intera struttura dell'istruzione condizionale.
- **While**: Questo statement crea un **basic block a se stante**, il quale avrà due archi uscenti etichettati rispettivamente "true", verso il basic block del corpo del ciclo, e "false", verso il basic block degli statement successivi al ciclo.

Supponiamo, per esempio, di avere il seguente programma e di volerne costruire il CFG:

```
int main(int argc, char** argv) {
    printf("Inserisci la lunghezza del vettore");
    int n = 0;
    scanf("%d", &n);
    if(n <= 0)
        exit(1);
    int V[n];
    printf("Inserisci %d numeri", n);
    int i = 0;
    int input;
    while(i < n) {
        scanf("%d", &input);
        V[i] = input;
        i++;
    }
    i = 1;
    int max = V[i];
    while(i < n) {
        if(V[i] > max)
            max = V[i];
        i++;
    }
    printf("Max: %d", max);
}
```

Listing 3.1: Un programma in C che calcola il massimo numero in un array di cinque elementi

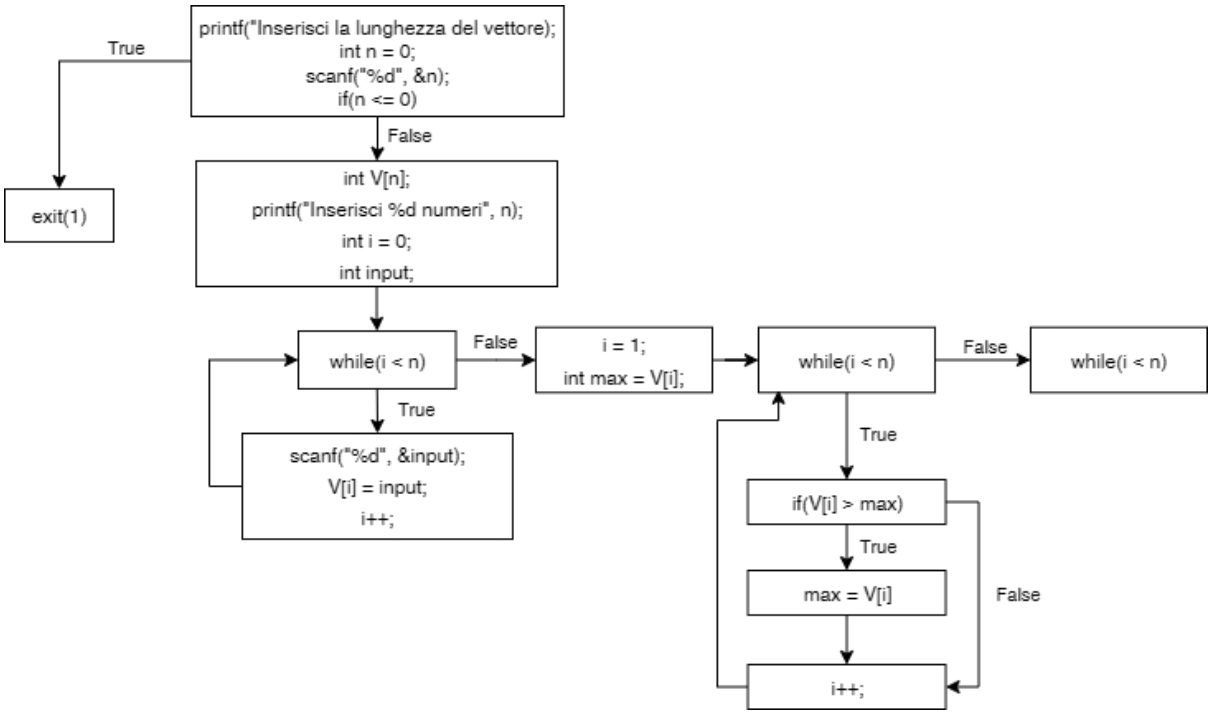

Figura 3.1: CFG del programma illustrato nel listing 3.1

3.2 Data Dependence Graph (DDG)

Quando si effettua l'analisi di un programma, oltre alla rappresentazione del flusso di controllo tramite CFG, risulta a volte utile tracciare le relazioni di dipendenza che sussistono tra le istruzioni del programma. Questo tipo di analisi permette infatti di individuare l'origine e l'utilizzo di variabili potenzialmente inquinate da dati malevoli. Per rappresentare queste relazioni possiamo usare una struttura a grafo che prende il nome di **Data-Dependence Graph** (DDG). Possiamo derivare un DGG direttamente dal CFG di un programma, tuttavia dobbiamo avere prima una definizione chiara di **dipendenza** fra gli statement; una possibile definizione è quella che prende il nome di **dipendenza per flusso** (flow-dependence) [17]: Sia $G = (V, E)$ il CFG per un programma P e siano $DEF(i)$ e $REF(i)$ gli insiemi che denotano rispettivamente le variabili definite e referenziate in un nodo $i \in V$ del CFG. Allora, un nodo $j \in V$ è **dipendente** dal nodo i rispetto ad una certa variabile se e solo se esiste una variabile x tale che:

1. $x \in DEF(i)$
2. $x \in REF(j)$
3. Esiste un cammino da i a j senza definizioni intermedie della variabile x (es. Altri assegnamenti ad x ecc...)

Applicando quindi la definizione di dipendenza data, sussiste una relazione di dipendenza tra due statement $S1$ e $S2$ se:

1. $S1$ definisce una variabile x
2. $S2$ contiene un riferimento ad x
3. Esiste un cammino da $S1$ a $S2$ dove x non viene ridefinita. Ciò significa che la definizione di x data in $S1$ viene utilizzata in $S2$.

Quindi, un DDG D per un programma P è un grafo **diretto e orientato** dove:

- I nodi di D rappresentano gli statement del programma
- Gli archi di D rappresentano le relazioni di dipendenza tra due statement

Supponiamo, per esempio, di avere il seguente programma e di volerne costruire il DDG:

```
int prince(int argc, char** argv) {
    int n; // S1
    scanf("%d", &n); // S2
    if(n < 0) // // S3
        n = -n; //S4
    int i = 1; //S5
    int tax = 0; //S6
    int price = 1; //S7
    while(i < n ) { //S8
        tax = tax +1; //S9
        price = price *i; //S10
        i = i+1; //S11
    }
    return price; //S12
}
```

Listing 3.2: Un programma in C che calcola il prezzo di un prodotto. Esempio proventiente da [17]

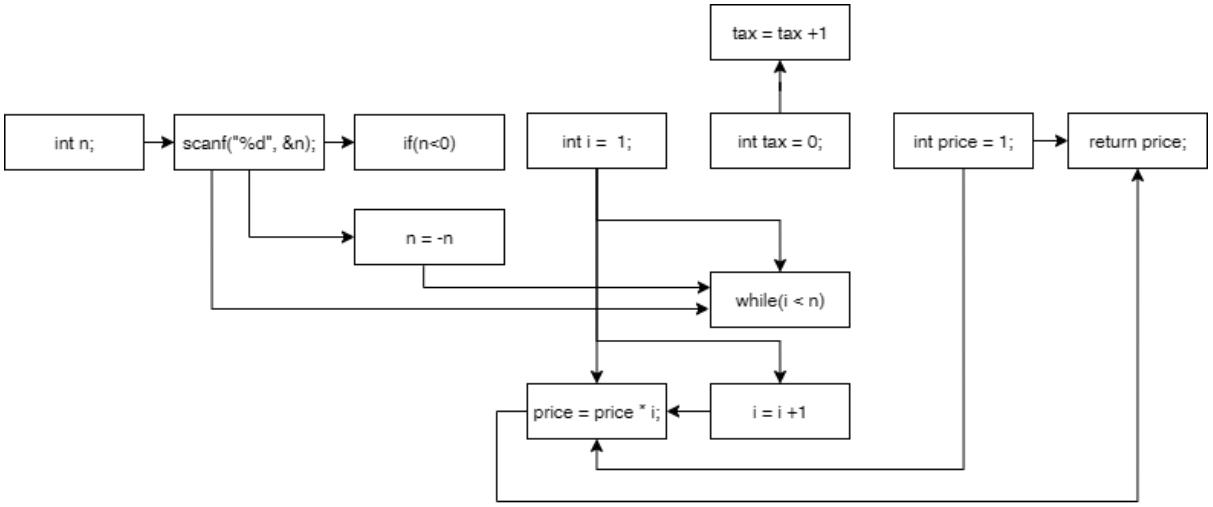

Figura 3.2: Data Dependence Graph per il listing 3.2. Adattato da [17]

3.3 Program slicing

La ricerca di vulnerabilità su programmi di grandi dimensioni può consumare una grande quantità di tempo e risorse. Inoltre, non tutte le computazioni effettuate da un programma sono potenzialmente vulnerabili. In questi casi, possiamo ridurre la dimensione delle spazio delle istruzioni da analizzare utilizzando una particolare tecnica di decomposizione chiamata **program slicing**. Questa tecnica produce un nuovo sottoprogramma, chiamato "slice", rilevante per una determinata computazione effettua dal programma principale. Il sottoprogramma prodotto da questa tecnica è comunque un programma eseguibile ed è prodotto a rispetto ad un **criterio di slicing**. Un criterio di slicing semplice utilizza due principali parametri [18]:

- **L'insieme delle variabili** rilevanti per la computazione di interesse
- Le **locazioni d'interesse** nel programma

Vi sono diverse tecniche per effettuare program slicing; alcuni esempi sono: [18]:

- **Program slicing statico (Backwards Slicing)**: Questa tecnica produce uno slice del programma senza tenere in considerazione l'input del programma. Fu la prima tecnica di slicing ad essere presentata [19].
- **Program slicing dinamico**: Questa tecnica, proposta da Korel and Laski [20], si basa sulla computazione delle slide tenendo in considerazione l'input ricevuto dal programma, il suo percorso di esecuzione e le relazioni di dipendenza tra gli statement.
- **Conditioned slicing**: Questo tipo di slicing forma un ponte tra slicing dinamico e statico. Il criterio di slicing condizionato usato da questa tecnica è una tripla (p, V, n) dove p è una qualche condizione iniziale di interesse, V l'insieme delle variabili di interesse e n l'insieme delle locazioni di interesse

Supponiamo di avere il seguente programma e di voler produrre uno slice, utilizzando la tecnica dello **slicing statico**, rispetto al criterio (*product*, 13):

```
int main(int argc, char** argv) {
    int n;
    scanf("%d", &n);
    int i = 1;
    int sum = 0;
    int product = 1;
    while(i <= n) {
        sum = sum + i;
        product = product *i;
        i++;
    }
    printf("%d\n", sum);
    printf("%d\n", product);
}
```

Listing 3.3: Un programma in C che calcola il fattoriale di un numero n e la somma da 1 a n. Esempio proventiente da [18]

```
int main(int argc, char** argv) {
    int n;
    scanf("%d", &n);
    int i = 1;
    int product = 1;
    while (i <= n) {
        product = product *i;
        i++;
    }
    printf("%d", product);

}
```

Listing 3.4: Slice ottenuta applicando slicing statico rispetto al criterio (*product*, 13)

3.4 Disassembling

Generalmente, la catena di compilazione di un linguaggio ad alto livello prevede una fase di *assemblaggio*, in cui il codice assembly generato dal compilatore viene tradotto nel linguaggio macchina specifico dell'architettura della CPU su cui il programma dovrà essere eseguito. Questa traduzione stabilisce quindi una relazione uno-a-uno tra le istruzioni macchina prodotte dall'*assemblatore* e le istruzioni assembly definita dalla *Instruction Set Architecture* (ISA) dell'architettura del processore. Questa relazione permette di effettuare anche la traduzione inversa e recuperare il codice assembly dall'insieme di istruzioni macchina presenti in un file binario. Questo processo è noto con il nome di **disassembling**. Effettuare il disassembly di un programma è una pratica fondamentale nell'ambito del reverse engineering, poiché permette di analizzare, in un formato leggibile dall'essere umano (assembly), le operazioni di basso livello che verranno eseguite dal calcolatore, rendendo possibile l'individuazione di potenziali problematiche di sicurezza sfruttabili da un attaccante. Seppur sembri un processo relativamente semplice, effettuare il disassembly di un codice macchina richiede di gestire diverse problematiche [21]:

- **Jump tables**: una *jump-table* è un array di indirizzi comunemente usata per per implementare trasferimenti del flusso di controllo multi-direzionali (ad esempio,

la trasposizione a basso livello del costrutto *switch* del linguaggio *C*). L'idea alla base dell'utilizzo di una jump table è quella di recuperare l'indirizzo a cui saltare indicizzando l'array con il valore dell'espressione per poi effettuare un jump indiretto verso l'indirizzo recuperato. Il codice che si occupa di questo processo è di solito preceduto da un controllo sul valore dell'espressione (*bound check*) per assicurarsi che non si stia cercando di accedere ad un indice non presente nell'array. Un disassembler dovrà quindi necessariamente stimare correttamente la grandezza della jump table per garantire la qualità del disassembly prodotto.

- **Position-Independent Code (PIC)**: Molti compilatori generano codice che può essere caricato ed eseguito indipendentemente dalla specifica sezione dello spazio di indirizzamento in cui vene caricato il programma. Questo tipo di codice viene detto *Position-Independent Code* (PIC). Quando viene prodotto PIC, il compilatore tipicamente crea delle *jump tables*, anch'esse indipendenti dalla posizione e formate da una serie di offset, le quali vengono inserite all'interno della sezione dell'eseguibile dedicata al codice (la "*text*" section). L'offset presente all'interno di queste tabelle verrà poi sommato all'indirizzo caricato al momento pe raggiungere la posizione desiderata tramite un jump indiretto. La presenza di PIC introduce due criticità che complicano il processo di disassembly:
  - Le tabelle sono **indistinguibili dai dati presenti nell'eseguibile**
  - Le sezioni di codice che effettuano i jump indiretti sono spesso **complesse** e non aderiscono a pattern di codice facilmente riconoscibili

  Considerate insieme, queste caratteristiche rendono il disassembly di sequenze di PIC contenenti jump table più problematiche rispetto all'analisi di codice standard.

3.5 Decompiling

Dato un programma binario, è possibile **ricostruire il codice ad alto livello** in cui è stato scritto attraverso un processo noto come *decompiling*. Lo scopo di un *decompiler* (o *reverse compiler*) è quindi quello di recuperare, partendo dal codice macchina, un programma scritto in un linguaggio ad alto livello che effettua le stesse operazioni del programma binario dato in input [22].

Bibliografia

[1] MITRE, *Common Weakness Enumeration Glossary*, Accesso effettuato il 26 settembre 2025, 2024. indirizzo: https://cwe.mitre.org/documents/glossary/

[2] P. Thomson, «Static analysis,» *Commun. ACM*, vol. 65, n. 1, pp. 50–54, dic. 2021, ISSN: 0001-0782. DOI: 10.1145/3486592 indirizzo: https://doi.org/10.1145/3486592

[3] Y. Xu et al., «A Review of Code Vulnerability Detection Techniques Based on Static Analysis,» in *Computational and Experimental Simulations in Engineering*, S. Li, cur., Cham: Springer Nature Switzerland, 2024, pp. 251–272, ISBN: 978-3-031-44947-5.

[4] G. Balakrishnan, R. Gruian, T. Reps e T. Teitelbaum, «CodeSurfer/x86—A platform for analyzing x86 executables,» in *Proceedings of the 14th International Conference on Compiler Construction*, ser. CC'05, Edinburgh, UK: Springer-Verlag, 2005, pp. 250–254, ISBN: 3540254110. DOI: 10.1007/978-3-540-31985-6_19 indirizzo: https://doi.org/10.1007/978-3-540-31985-6_19

[5] Z. Feng, Z. Wang, W. Dong e R. Chang, «Bintaint: A Static Taint Analysis Method for Binary Vulnerability Mining,» in *2018 International Conference on Cloud Computing, Big Data and Blockchain (ICCBB)*, 2018, pp. 1–8. DOI: 10.1109/ICCBB.2018.8756383

[6] W. Qingyang, H. Quanrui, N. Yuqiao, B. Chenya, G. Zhen e S. Shiwen, «A Survey of Binary Code Security Analysis,» in *2023 6th International Conference on Data Science and Information Technology (DSIT)*, 2023, pp. 42–49. DOI: 10.1109/DSIT60026.2023.00015

[7] Z. Tai, H. Washizaki, Y. Fukazawa, Y. Fujimatsu e J. Kanai, «Binary Similarity Analysis for Vulnerability Detection,» in *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*, 2020, pp. 1121–1122. DOI: 10.1109/COMPSAC48688.2020.0-110

[8] Y. David, N. Partush e E. Yahav, «Statistical similarity of binaries,» in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16, Santa Barbara, CA, USA: Association for Computing Machinery, 2016, pp. 266–280, ISBN: 9781450342612. DOI: 10.1145/2908080.2908126 indirizzo: https://doi.org/10.1145/2908080.2908126

[9] N. Nethercote, «Dynamic binary analysis and instrumentation,» University of Cambridge, Computer Laboratory, rapp. tecn. UCAM-CL-TR-606, nov. 2004. DOI: 10.48456/tr-606 indirizzo: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-606.pdf

[10] M. Ernst, «Static and Dynamic Analysis: Synergy and Duality,» mag. 2003.

[11] H. Liang, X. Pei, X. Jia, W. Shen e J. Zhang, «Fuzzing: State of the Art,» *IEEE Transactions on Reliability*, vol. 67, n. 3, pp. 1199–1218, 2018. DOI: 10.1109/TR.2018.2834476

[12] J. Li, B. Zhao e C. Zhang, «Fuzzing: a survey,» *Cybersecurity*, vol. 1, dic. 2018. DOI: 10.1186/s42400-018-0002-y

[13] H. Xue, S. Sun, G. Venkataramani e T. Lan, «Machine Learning-Based Analysis of Program Binaries: A Comprehensive Study,» *IEEE Access*, vol. 7, pp. 65 889–65 912, 2019. DOI: 10.1109/ACCESS.2019.2917668

[14] A. Aumpansub e Z. Huang, «Learning-based Vulnerability Detection in Binary Code,» in *Proceedings of the 2022 14th International Conference on Machine Learning and Computing*, ser. ICMLC '22, Guangzhou, China: Association for Computing Machinery, 2022, pp. 266–271, ISBN: 9781450395700. DOI: 10.1145/3529836.3529926 indirizzo: https://doi.org/10.1145/3529836.3529926

[15] R. Li, C. Zhang, C. Feng, X. Zhang e C. Tang, «Locating Vulnerability in Binaries Using Deep Neural Networks,» *IEEE Access*, vol. 7, pp. 134 660–134 676, 2019. DOI: 10.1109/ACCESS.2019.2942043

[16] P. Xu, Z. Mai, Y. Lin, Z. Guo e V. S. Sheng, «A Survey on Binary Code Vulnerability Mining Technology,» *Journal of Information Hiding and Privacy Protection*, vol. 3, n. 4, pp. 165–179, 2021, ISSN: 2637-4226. indirizzo: http://www.techscience.com/jihpp/v3n4/47056

[17] X. Wang, J. Sun, X. Yang, Z. He e S. Maddineni, «Automatically identifying domain variables based on data dependence graph,» in *2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No.04CH37583)*, vol. 4, 2004, 3389–3394 vol.4. DOI: 10.1109/ICSMC.2004.1400866

[18] B. Xu, J. Qian, X. Zhang, Z. Wu e L. Chen, «A brief survey of program slicing,» *SIGSOFT Softw. Eng. Notes*, vol. 30, n. 2, pp. 1–36, mar. 2005, ISSN: 0163-5948. DOI: 10.1145/1050849.1050865 indirizzo: https://doi.org/10.1145/1050849.1050865

[19] M. Weiser, «Program Slicing,» in *Proceedings of the 5th International Conference on Software Engineering (ICSE)*, IEEE Press, 1981, pp. 439–449.

[20] B. Korel e J. W. Laski, «Dynamic Program Slicing,» *Information Processing Letters*, vol. 29, n. 3, pp. 155–163, 1988.

[21] B. Schwarz, S. Debray e G. Andrews, «Disassembly of executable code revisited,» in *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, 2002, pp. 45–54. DOI: 10.1109/WCRE.2002.1173063

[22] C. G. Cifuentes e K. J. Gough, «Decompilation of binary programs,» *Software: Practice and Experience*, vol. 25, 1995. indirizzo: https://api.semanticscholar.org/CorpusID:8229401