



UNIVERSITÀ DEGLI STUDI DI MILANO - BICOCCA
Dipartimento di Informatica, Sistemistica e Comuni-
cazione
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

1	Introduzione	2
1.1	Differenza tra debolezza e vulnerabilità	2
1.2	Struttura della relazione	3
2	Stato dell'arte	4
2.1	Metodologie basate su tecniche di analisi statica	4
2.1.1	Taint analysis statica	4
2.1.2	Binary Code Similarity Detection	5
2.2	Metodologie basate su tecniche di analisi dinamica	7
2.2.1	Fuzzing	7
2.3	Tecniche basate su modelli di apprendimento automatico	8
2.4	Tecniche ibride	9
3	Metodologie utilizzate	10
3.1	Control Flow Graph (CFG)	10
3.2	Data Dependence Graph (DDG)	12
3.3	Program slicing	13
3.4	Esecuzione simbolica	14
3.5	Disassembling	16
3.6	Decompiling	17
4	Tecnologie utilizzate	19
4.1	Capstone	19
4.2	Ghidra	20
4.3	angr	21
5	Analisi implementate	23
5.1	VulnDetect	23
5.1.1	Buffer overflow e Stack-Based Buffer Overflow	23
5.1.2	Strategia di analisi	24
5.2	Arbiter	25
5.2.1	Property-Compliant Vulnerabilities	26
5.2.2	Vulnerability descriptions	26
5.2.3	Strategia di analisi	27
6	Implementazione	29
6.1	Architettura	29
6.2	Backend	29
6.2.1	Configurazione	31

6.2.2	Implementazione delle tecniche di analisi	31
6.3	Frontend	32
6.3.1	Pagina principale	32
6.3.2	Pagina di analisi	33
Bibliografia		37
A Codice relativo al backend		40
A.1	Server web	40
A.2	Classi wrapper per le analisi	43
A.2.1	VulnDetect	43
A.2.2	Arbiter	47

Elenco delle figure

2.1	Schema di funzionamento di VulneraBin. Immagine proveniente da [7] . . .	6
2.2	Funzionamento generale di un fuzzer. Immagine proveniente da [11] . . .	8
3.1	CFG del programma illustrato nel listing 3.1	11
3.2	Data Dependence Graph per il listing 3.2. Adattato da [17]	13
3.3	Flusso completo del processo di decompilazione. Immagine proventiente da [24]	18
4.1	Struttura interna di cs_isns. Immagine proveniente da [25]	19
4.2	Architettura del framework Ghidra	20
4.3	Architettura di angr. Immagine proveniente da [28]	22
5.1	Processo di analisi impiegato da VulnDetect. Immagine proveniente da [30]	25
5.2	Processo di analisi implementato da Arbiter. Figura proveniente da [31] .	28
6.1	Pagina principale dell'applicazione	33
6.2	Pagina di analisi dell'applicazione senza alcun risultato	34
6.3	Pagina di analisi dell'applicazione con i risultati ottenuti da <i>VulnDetect</i> .	34
6.4	Pagina di analisi dell'applicazione con i risultati ottenuti da <i>Arbiter</i> . . .	35

Listings

3.1	Un programma in C che calcola il massimo numero in un array di cinque elementi	11
3.2	Un programma in C che calcola il prezzo di un prodotto. Esempio proveniente da [17]	12
3.3	Un programma in C che calcola il fattoriale di un numero n e la somma da 1 a n . Esempio proveniente da [18]	14
3.4	Slice ottenuta applicando slicing statico rispetto al criterio (<i>product</i> , 13) .	14
3.5	Un programma in C che controlla se l'utente ha scritto un numero tra cinque e sette	15
6.1	Struttura del file JSON ritornato dall'endpoint /disassemble	30
6.2	Struttura del file JSON ritornato dall'endpoint /decompile	30
6.3	Struttura del file JSON ritornato da vulndetect	30
6.4	Struttura del file JSON ritornato da arbiter	31
6.5	Proxy per l'operazione di caricamento di un file	33
6.6	Struttura di un proxy per le operazioni di analisi. <i>Analysis</i> può assumere il valore <i>vulndetect</i> oppure <i>arbiter</i>	35
6.7	Struttura di un proxy per le operazioni di disassembly e decompiling. <i>Operation</i> può assumere il valore <i>disassemble</i> oppure <i>decompile</i>	35
6.8	Funzione <i>sendData</i> utilizzata dal codice riportato nel listing 6.9	36
6.9	Codice che svolge il recupero del disassembly e del decompiling tramite l'interrogazione del backend. Questo codice viene eseguito immediatamente dopo il caricamento della pagina di analisi	36
A.1	Endpoint /upload	40
A.2	Endpoint /decompile	40
A.3	Endpoint /disassemble	41
A.4	Endpoint /analyses/vulndetect	42
A.5	Endpoint /analyses/arbiter	42
A.6	Funzione <i>find_vd_files</i>	43
A.7	Funzione <i>save_file</i>	43
A.8	Funzioni considerate potenzialmente pericolose da <i>VulnDetect</i>	43
A.9	Metodo init della classe wrapper per <i>VulnDetect</i>	44
A.10	Metodo static_analysis della classe wrapper per <i>VulnDetect</i>	45
A.11	Metodo vulnerability_detection della classe wrapper per <i>VulnDetect</i> . . .	46
A.12	Metodo analyze della classe wrapper per <i>VulnDetect</i>	47
A.13	Metodo init della classe wrapper per <i>Arbiter</i>	47
A.14	Metodo enable_logging della classe wrapper per <i>Arbiter</i>	48
A.15	Metodo setup della classe wrapper per <i>Arbiter</i>	48
A.16	Metodo analyze della classe wrapper per <i>Arbiter</i>	49

Elenco delle tabelle

5.1	Vulnerability description attualmente disponibili in Arbiter	27
5.2	Classi di vulnerabilità per le quali è possibile scrivere una VD	27

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'obiettivo 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 compo-

nente 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 delle tecniche di analisi rese disponibili dalla piattaforma. Verrà approfondito il funzionamento interno e le funzionalità per ogni tecnologia impiegata.
- **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 debolezze 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 taint 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'obiettivo 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 contenente 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 analisi 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.

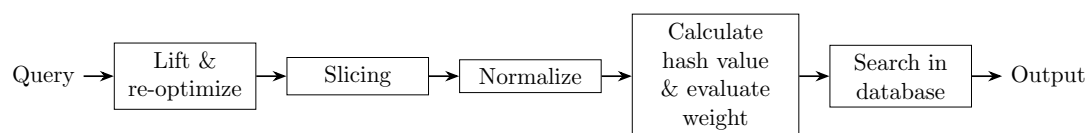


Figura 2.1: Schema di funzionamento di VulneraBin. Immagine proveniente da [7]

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

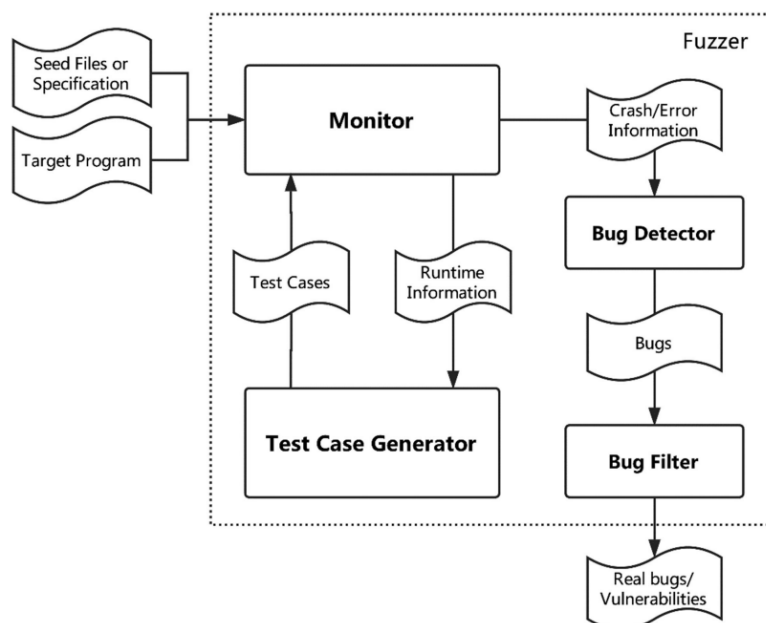


Figura 2.2: Funzionamento generale di un fuzzer. Immagine proveniente da [11]

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 per l'identificazione automatica di vulnerabilità. Le tecniche di apprendimento automatico possono dividersi 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 obiettivo, ricevendo una ricompensa o una penalità in base a quanto la scelta che ha compiuto lo avvicina all'obiettivo

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 stati 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 addestramento 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:

```

1 int main(int argc, char** argv) {
2     printf("Inserisci la lunghezza del vettore");
3     int n = 0;
4     scanf("%d", &n);
5     if(n <= 0)
6         exit(1);
7     int V[n];
8     printf("Inserisci %d numeri", n);
9     int i = 0;
10    int input;
11    while(i < n) {
12        scanf("%d", &input);
13        V[i] = input;
14        i++;
15    }
16    i = 1;
17    int max = V[i];
18    while(i < n) {
19        if(V[i] > max)
20            max = V[i];
21        i++;
22    }
23    printf("Max: %d", max);
24 }

```

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

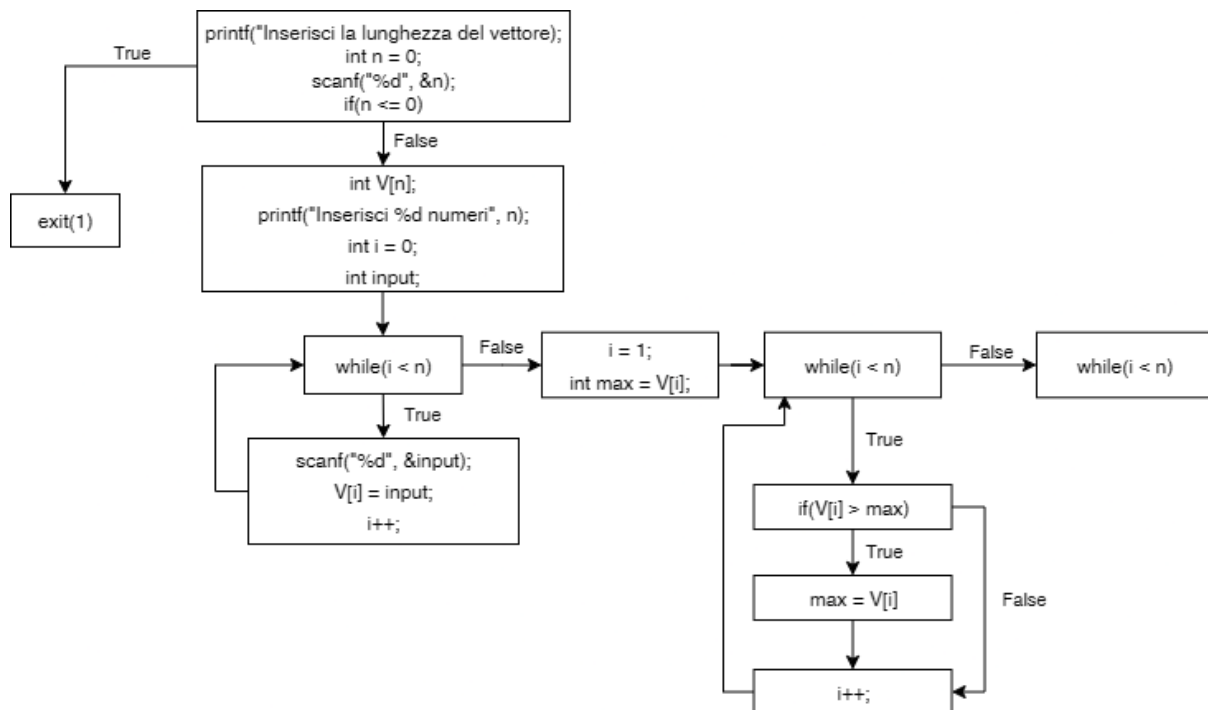


Figura 3.1: CFG del programma illustrato nel listing 3.1

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 DDG 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:

```

1 int prince(int argc, char** argv) {
2     int n; // S1
3     scanf("%d", &n); // S2
4     if(n < 0) // // S3
5         n = -n; //S4
6     int i = 1; //S5
7     int tax = 0; //S6
8     int price = 1; //S7
9     while(i < n) { //S8
10         tax = tax + 1; //S9
11         price = price * i; //S10
12         i = i + 1; //S11
13     }
14     return price; //S12
15 }
```

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

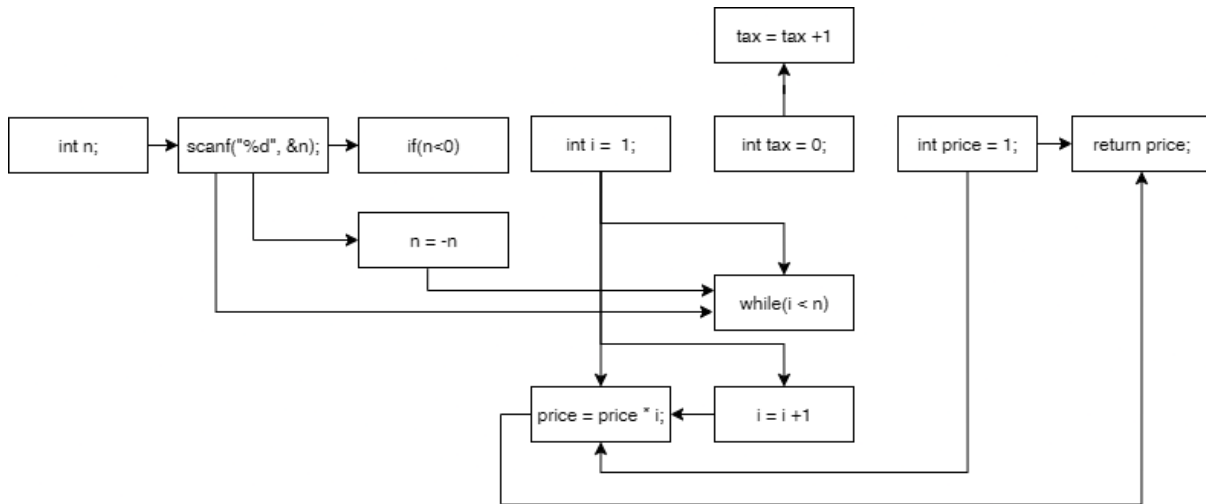


Figura 3.2: Data Dependence Graph per il listing 3.2. Adattato da [17]

3.3 Program slicing

La ricerca di vulnerabilità in software di grandi dimensioni può consumare una grande quantità di tempo e risorse computazionali. Inoltre, non tutte le computazioni effettuate da un programma sono rilevanti o potenzialmente vulnerabili. In questi casi, è possibile ridurre la dimensione dello spazio delle istruzioni da analizzare utilizzando una particolare tecnica di decomposizione chiamata **program slicing**. Questa tecnica produce un nuovo sottoprogramma, denominata *slice*, che include solamente le istruzioni di interesse per una determinata computazione del programma. Lo slicing di un programma produce il più **piccolo sottoprogramma eseguibile** che mantiene il comportamento del programma originale rispetto a quella computazione ed è prodotto rispetto ad un **criterio di slicing**. Un criterio di slicing è una coppia $C = (V, n)$ dove [18]:

- V è l'insieme delle **variabili** rilevanti per la computazione di interesse
- n è l'insieme dell **locazioni di interesse** nel programma

Esistono diverse metodologie per effettuare lo slicing di un programma; 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:** Proposta da Korel e Laski [20], questo tipo di slicing si basa sulla computazione dello slice tenendo in considerazione l'input ricevuto dal programma, il suo percorso di esecuzione e le relazioni di dipendenza tra gli statement.
- **Conditioned slicing:** Questa tecnica funge da ponte tra slicing dinamico e statico. Nel conditioned slicing, una condizione di slicing è una tripla $C_{cond} = (p, V, n)$, dove p è una qualche condizione iniziale di interesse.

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

```
1 int main(int argc, char** argv) {
2     int n;
3     scanf("%d", &n);
4     int i = 1;
5     int sum = 0;
6     int product = 1;
7     while(i <= n) {
8         sum = sum + i;
9         product = product * i;
10        i++;
11    }
12    printf("%d\n", sum);
13    printf("%d\n", product);
14 }
```

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

```
1 int main(int argc, char** argv) {
2     int n;
3     scanf("%d", &n);
4     int i = 1;
5     int product = 1;
6     while (i <= n) {
7         product = product * i;
8         i++;
9     }
10    printf("%d", product);
11
12 }
```

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

3.4 Esecuzione simbolica

Quando avviene la costruzione di un control flow graph per un programma, l'obiettivo d'interesse è quello di rappresentare tutti i suoi possibili flussi di esecuzione. Tuttavia, ciò che non viene considerato durante la costruzione di un CFG è l'effettiva praticabilità dei suoi cammini; infatti alcuni cammini presenti sul CFG potrebbero essere **irraggiungibili** da un qualsiasi input dato al programma. Possiamo classificare i cammini su un CFG in due diverse categorie:

- **Praticabili** (feasible): Se esiste un input concreto di valore V tale che se il programma è eseguito con V , allora esso esegue tutte le istruzioni sul cammino considerato.
- **Impraticabili** (infeasible): Se non esiste nessun input di valore V che porta il programma ad eseguire il cammino considerato. La presenza di cammini non praticabili sul CFG **non implica necessariamente la presenza di codice morto**, tuttavia la presenza di quest'ultimo implica l'esistenza di cammini non praticabili. Spesso, una grossa porzione di tutti i cammini di esecuzione di un programma sono impraticabili.

L'**esecuzione simbolica** funge da ponte tra il comportamento di un programma e la sua logica. A differenza dell'esecuzione concreta di un programma, la quale utilizza un input concreto e permette di analizzare solamente un determinato cammino di esecuzione del programma, l'esecuzione simbolica permette di **esplorare simultaneamente tutti i cammini praticabili** che il programma potrebbe intraprendere al variare dell'input, eseguendo il programma con valori simbolici [21]. Un framework di esecuzione simbolica è tendenzialmente composto dalle seguenti componenti [21]:

1. **Esecutore simbolico:** Esegue il programma utilizzando input simbolici. Per ogni cammino di esecuzione esplorato, mantiene le seguenti informazioni:
 - **Path condition:** Una formula booleana nella logica del primo ordine che caratterizza lo stato d'esecuzione del programma. Essa esprime, sotto forma di vincoli logici, le condizioni soddisfatte dall'input durante l'esecuzione simbolica del programma. Ogni istruzione di branch che viene eseguita dall'esecutore aggiorna la path condition per un determinato cammino con un nuovo vincolo.
 - **Memoria simbolica:** Mappa le variabili a delle espressioni o valori simbolici. Le istruzioni di assegnamento del programma aggiornano la memoria simbolica dell'esecutore.
2. **Model checker:** Un model checker, tipicamente un *Satisfiability modulo theories* (SMT) solver, viene utilizzato per controllare se vi sono delle violazioni logiche nella path condition rispetto ad una proprietà d'interesse (per esempio, una proprietà di sicurezza) e se il cammino è praticabile, cioè se la path condition che lo caratterizza è una formula **soddisfacibile**. In generale, il problema della soddisfacibilità booleana (conosciuto con il nome di SAT) è un problema **indecidibile**; tuttavia, può essere risolto in un gran numero di casi pratici; per esempio, i vincoli booleani lineari (es. $X > Y \wedge X + Y \leq 10$) sono spesso risolvibili in maniera efficiente.

L'esecuzione simbolica di un programma può, inoltre, avvenire in due diversi modi:

- **Esecuzione simbolica statica:** L'esecuzione avviene senza effettivamente eseguire il programma. Vengono esplorati tutti i possibili cammini praticabili del CFG.
- **Esecuzione simbolica dinamica (analisi concolica):** l'esecuzione simbolica del programma viene affiancata ad un'esecuzione; vengono quindi esplorati ed eseguiti simbolicamente solo i cammini raggiunti dal valore concreto dell'input.

Supponiamo, per esempio, di voler eseguire simbolicamente (in maniera statica) il seguente programma:

```
1 int main() {  
2     int x;  
3     scanf("%d", &x);  
4     if (x >= 5 && x <= 7) {  
5         printf("Numero tra 5 e 7!\n"); // Percorso A  
6     } else {  
7         printf("Numero minore di 5 o inferiore di 7! \n") // Percorso B  
8     }  
9     return 0;  
10 }
```

Listing 3.5: Un programma in C che controlla se l'utente ha scritto un numero tra cinque e sette

L'esecutore simbolico esplorerà tutti i cammini praticabili del programma:

- **Variabili simboliche:** $x = X$ con X valore simbolico
- **Path condition:**
 - **Percorso A:** $X \geq 5 \wedge X \leq 7$
 - **Percorso B:** $X < 5 \vee X > 7$

3.5 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 [22]:

- **Jump tables:** una *jump-table* è un array di indirizzi comunemente usata 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 viene 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 per 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.

3.6 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. Per effettuare il recupero del codice di un programma, un *decompiler* deve gestire una molteplicità di problemi sia teorici che pratici, tra cui [23]:

- **Problemi di rappresentazione:** Nell'architettura di Von Neumann, sia le istruzioni che i dati sono rappresentati nello stesso modo. Dato un byte in memoria, è impossibile determinare se esso rappresenta un'istruzione, un dato (o entrambi) fino a quando quel byte non verrà estratto dalla memoria, posto in un registro e utilizzato di conseguenza.
- **Codice auto-modificante:** Il *codice auto-modificante* (self-modifying code) si riferisce ad istruzioni o dati preimpostati in memoria che vengono modificati durante l'esecuzione del programma. Un byte in una posizione di memoria può essere modificato per rappresentare un'altra istruzione o un'altro dato. Questa tecnica era particolarmente rilevante durante gli anni '60 e '70; quando i limiti sulla grandezza della memoria dei calcolatori imponevano un utilizzo efficiente della stessa. Un modo per efficientarne l'uso era di modificare i byte caricati in una certa parte della memoria in modo tale che essi potessero essere riutilizzati come istruzioni o come dati a runtime. La presenza di codice auto-modificante rende più complesso effettuare il decompiling del programma, poiché il decompiler dovrà tenere in considerazione la modifiche introdotte da questa tecnica per recuperare il codice ad alto livello corretto.
- **Subroutine incluse da compilatori e linker:** Quando un programma viene compilato, il compilatore e il linker inseriscono una grande quantità di subroutine legate alle funzioni che queste due componenti svolgono. In particolare, il compilatore includerà sempre delle routine di avvio che effettuano il setup dell'ambiente e, se richieste, delle subroutine di supporto al motore d'esecuzione. Queste routine sono di solito scritte in assembly e quindi risultano spesso **intraducibili** in un linguaggio ad alto livello. Ad esempio, un programma in C che stampa a schermo "hello world", quando compilato, conterrà nel suo file binario oltre 25 subroutine; mentre un programma in Pascal che effettua la stessa operazione ne conterrà addirittura più di 40.

Concettualmente, un *decompiler* è strutturato in maniera simile ad un compilatore e prevede l'esecuzione di diverse fasi per effettuare la trasformazione da codice macchina a codice ad alto livello [23]. Queste fasi possono essere strutturate in tre moduli distinti [24]:

1. **Front-end:**

- Prende in input il file binario per una specifica architettura e scritto in uno specifico linguaggio, lo carica in uno spazio di memoria virtuale
- Effettua il parsing delle istruzioni presenti nel programma
- Produce sia una rappresentazione del programma in un linguaggio intermedio a basso livello sia il Control Flow Graph del programma.

2. **Universal decompiling machine (UDM):**

- È un modulo intermedio **completamente indipendente** dall'architettura del processore e dal linguaggio di programmazione utilizzato per scrivere il programma
- Effettua due tipi di analisi, rispettivamente *data flow analysis* e *control flow analysis*, per raffinare la rappresentazione intermedia a basso livello in una rappresentazione intermedia più ad alto livello.
- Inoltre, il modulo si occupa di ristrutturare il CFG prodotto dal primo modulo in un insieme di **costrutti** generalmente presenti in un linguaggio ad alto livello.

3. **Back-end:**

- Questo modulo si occupa della generazione effettiva del codice decompilato. Questo modulo è quindi **dipendente dal linguaggio di programmazione in cui si vuole avere il risultato finale**
- Effettua, opzionalmente, un'ulteriore ristrutturazione del CFG, in modo tale che i costrutti specifici presenti nel linguaggio obbiettivo ma non presenti nel generico insieme di costrutti generato durante la fase precedente vengano utilizzati.
- Infine, il modulo genera il codice ad alto livello (codice decompilato)

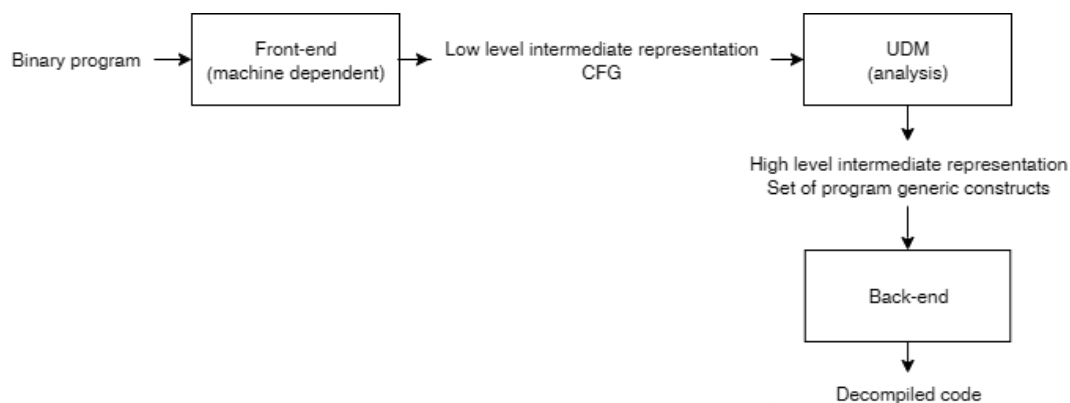


Figura 3.3: Flusso completo del processo di decompilazione. Immagine proveniente da [24]

Capitolo 4

Tecnologie utilizzate

Questo capitolo è dedicato ad un'analisi dettagliata delle principali tecnologie adottate per lo sviluppo della piattaforma Binoculars. Verrà approfondito il funzionamento e la struttura interna per ogni tecnologia di analisi impiegata e quali funzionalità essa offre nell'ambito dell'analisi di file binari.

4.1 Capstone

Capstone [25] è una framework leggero e multi-piattaforma per effettuare il disassembly di codice macchina. Si basa sulla famiglia di framework per compilatori *LLVM*, in particolare sul modulo *LLVM-Machine Code* (LLVM-MC), il quale contiene un disassemblatore interno con supporto per molteplici architetture. Questo supporto è dato dalla presenza di molteplici *tabelle descrittive* (file *.TD*), le quali descrivono in maniera astratta tutte le caratteristiche dell'ISA di una determinata architettura. Capstone sfrutta queste tabelle per estrarre le seguenti **informazioni semantiche**:

- **Registri impliciti:** I registri letti e/o scritti **implicitamente dall'istruzione**; questi registri infatti non appaiono esplicitamente nella stringa della operazione.
- **Gruppi di istruzioni:** A quale categoria funzionale (es. aritmetica, logica, ...) appartiene l'istruzione

Per ogni istruzione macchina, Capstone produce una struttura dati di output denominata *cs_insn*, la quale contiene due tipi di informazioni:

- **Informazioni di base (Indipendenti dall'architettura):** ID, dimensioni (in byte), mnemonic, e una stringa contenete gli operandi dell'istruzione
- **Informazioni dettagliate:** Contiene le informazioni estratte dal file *.TD*

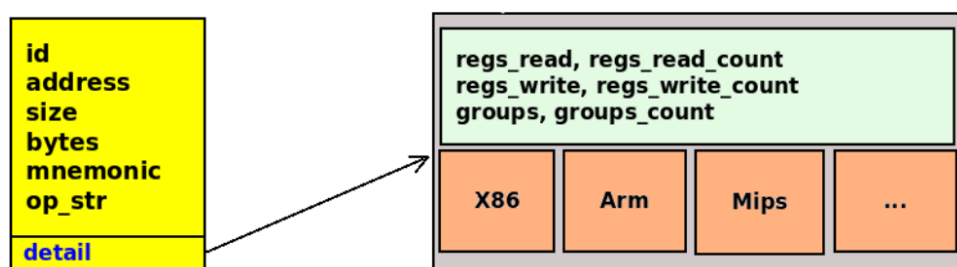


Figura 4.1: Struttura interna di *cs_insn*. Immagine proveniente da [25]

4.2 Ghidra

Ghidra è un framework di reverse engineering sviluppato dalla *National Security Agency* degli Stati Uniti d’America [26]. La piattaforma che il framework offre contiene un disassembler, un decompiler e un vasto insieme di tool e script di analisi. Inoltre, può essere estesa tramite l’aggiunta di plugin e script di analisi personalizzati scritti in python o Java (il linguaggio in cui Ghidra è scritto). In particolare, è possibile interagire con le API messe a disposizione da Ghidra direttamente da python, utilizzando la libreria *pyghidra*, originariamente sviluppata dal *Department of Defense Cyber Crime Center* (DC3) (un dipartimento del governo americano) e ora parte integrante della piattaforma. L’architettura di Ghidra è estremamente complessa e formata da diversi moduli software, di cui i principali sono [26]:

- **Loaders:** Questi moduli si occupano dell’importazione dei file binari all’interno della piattaforma. Ogni loader è pensato per un singolo tipo di file eseguibile (ELF, Windows PE, ...). Ghidra permette inoltre di estendere l’insieme di loader disponibili tramite l’aggiunta di loader scritti dall’utente stesso.
- **Processors:** I Ghidra processors sono i moduli più complessi dell’intera piattaforma e si occupano di tutte le operazioni riguardanti il disassembly del file binario. Similmente ai file *.TD* visti per Capstone, anche il processo di disassembly di Ghidra utilizza dei file di specifica dell’ISA per le diverse architetture supportate. Questi file sono scritti usando il linguaggio specifico per Ghidra denominato *SLEIGH*.
- **Ghidra decompiler:** Questo modulo si occupa del processo di decompilazione del file binario in una rappresentazione ad alto livello ispirata al linguaggio *C*. Il processo di decompilazione di ghidra avviene in tre fasi distinte:
 1. Il decompiler usa i file scritti in SLEIGH per creare una bozza del codice ad alto livello nel linguaggio di rappresentazione intermedia *p-code*. Inoltre, in questa fase viene derivato il CFG del programma
 2. Il codice viene raffinato, eliminando gli statement irraggiungibili o inutili. A seguito di questa operazione, avviene anche una ristrutturazione del CFG in modo da riflettere i potenziali cambiamenti introdotti al flusso di controllo del programma
 3. Viene generato il codice decompilato in un linguaggio *C-like*

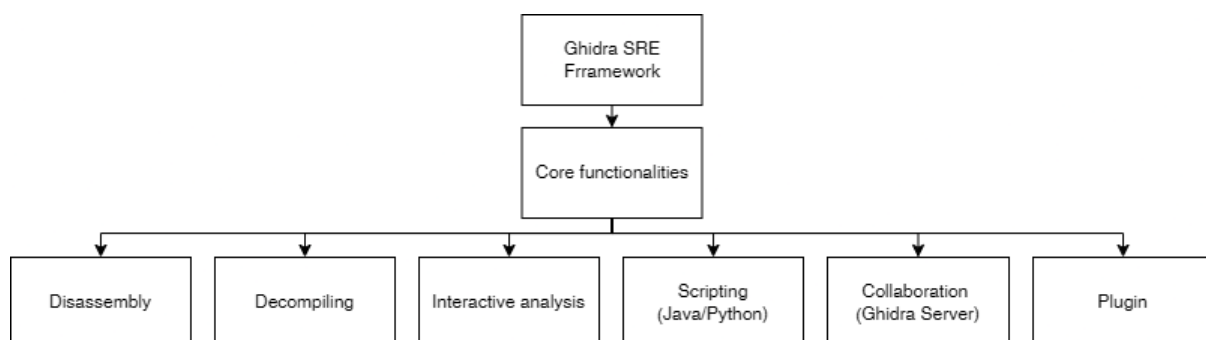


Figura 4.2: Architettura del framework Ghidra

4.3 angr

angr [27] è una piattaforma per l'analisi binaria scritta in occasione della *DARPA Cyber Grand Challenge 2016*. Essa offre un supporto multi-architettura per l'esecuzione di un'ampia gamma di tecniche di analisi binaria [27]:

- **CFG Recovery:** angr permette di effettuare il recupero del CFG del programma caricato. Per svolgere questa funzione, angr implementa un'algoritmo iterativo per la costruzione del CFG e sfrutta una combinazione di tecniche di analisi per effettuare il recupero, quando possibile, degli indirizzi obbiettivo per ogni salto indiretto presente nel programma. Per ottimizzare i tempi di calcolo dell'algoritmo di recupero, angr effettua una serie di assunzioni riguardo al programma:
 1. Tutto il codice del programma può essere distribuito in funzioni differenti.
 2. Tutte le funzioni vengono chiamate esplicitamente oppure sono precedute da un'istruzione di salto in coda.
 3. La procedura di pulizia dello stack è prevedibile e non dipende da quale funzione (chiamata o chiamante) viene invocata.
- **Value-Set Analysis:** Value-Set Analysis è una tecnica di analisi statica che permette di approssimare il valore di ogni registro e locazione di memoria ad ogni punto dell'esecuzione di un programma. Questa tecnica analizza il programma fino a quando l'approssimazione raggiunge un *punto fisso*, cioè una sovra-approssimazione stretta di tutti i valori che ogni registro o locazione di memoria può avere ad ogni punto nel programma.
- **Esecuzione simbolica dinamica:** La tecnica di analisi principale messa a disposizione da angr è quella dell'analisi simbolica dinamica. La piattaforma mette a disposizione un potente motore di esecuzione simbolica, il quale sfrutta le funzionalità messe a disposizione dalla libreria *claripy* e dal SMT solver *Z3* per popolare la memoria simbolica ed effettuare il controllo di soddisfacibilità delle path condition per ogni cammino analizzato.
- **Under Constrained Symbolic Execution (UCSE):** angr mette a disposizione un'ulteriore tipo di esecuzione simbolica, chiamata *Under Constrained Symbolic Execution (UCSE)*. Essa effettua un'analisi separata per ogni funzione presente nel programma. Poiché ogni funzione viene analizzata senza la presenza di variabili globali e degli argomenti con cui viene chiamata, questo tipo di analisi non è particolarmente accurata e produce falsi positivi. UCSE etichetta tutti i dati mancanti a causa della mancanza di "contesto" (variabili globali e parametri) come *under-constrained*. Quando l'analisi rileva una violazione di sicurezza, viene effettuato un controllo su tutti i valori coinvolti: se presentano tutti lo stato di *under-constrained*, allora la violazione rilevata viene filtrata come falso positivo.
- **Automatic Exploit Generation (AEG):** angr mette a disposizione dei moduli per effettuare la generazione automatica di exploit, i quali permettono ad un'analista di confermare se la vulnerabilità rilevata può essere effettivamente sfruttata da un potenziale attaccante.

La piattaforma è suddivisa nei seguenti sotto-moduli [27]:

- **Intermediate Representation:** Per permettere il supporto per molteplici architetture, il codice macchina del programma analizzato viene tradotto nel linguaggio di rappresentazione intermedia *VEX*, sviluppato per il progetto *Valgrind* e specificatamente studiato per l'analisi binaria.
- **CLE:** Il compito di caricare il file binario viene gestito dal modulo *CLE*, acronimo ricorsivo per *CLE Loads Everything*. CLE astrae il processo di caricamento per una numerosa quantità di formati di file eseguibili, gestendo la risoluzione dinamica dei simboli, la rilocalizzazione della memoria e la corretta inizializzazione dello stato del programma
- **SimuVEX:** Il modulo *SimuVEX* si occupa di effettuare la rappresentazione dello stato del programma (valori dei registri, file aperti, ...). Il modulo, inoltre, permette di rappresentare i cambiamenti semantici sullo stato del programma dati dall'esecuzione di un particolare blocco di codice. In particolare, SimuVEX permette di processare uno *stato di input*, attraverso un blocco di codice VEX, e di generare uno *stato di output* (o un'insieme di stati di output, in caso il blocco di codice contenga un'istruzione condizionale).
- **Data Model:** I valori presenti nei registri sono rappresentati tramite le astrazioni messe a disposizione dalla libreria *Claripy*. La libreria astrae ogni valore in una rappresentazione simbolica chiamata *espressione*, la quale traccia tutte le operazioni in cui essa viene utilizzata. Le espressioni vengono rappresentate mediante strutture ad albero chiamate *expression trees*, nelle quali i nodi intermedi contengono l'operazione svolta, mentre le foglie contengono i valori usati come argomenti dell'operazione. Per esempio, se all'espressione x viene aggiunta l'espressione 5, la nuova espressione risultante sarà $x + 5$. Il suo *expression tree*, quindi, avrà l'operazione di somma alla radice, mentre le foglie conterranno le espressioni x e 5.
- **Full program analysis:** Questo modulo mette a disposizione dell'analista un entry point (la classe *Project*) per accedere sia alle funzionalità dei moduli descritti sopra sia alle tecniche di analisi binaria messe a disposizione dalla piattaforma.

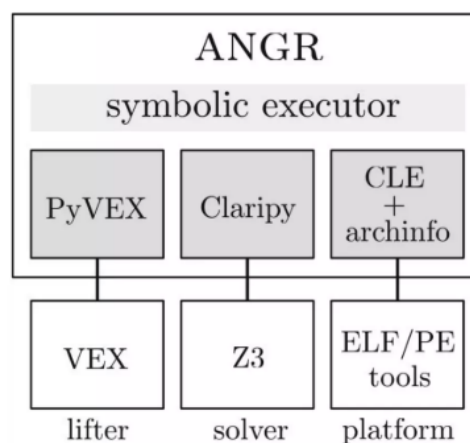


Figura 4.3: Architettura di angr. Immagine proveniente da [28]

Capitolo 5

Analisi implementate

In questo capitolo, verranno descritte nel dettaglio le tecniche di ricerca automatica delle vulnerabilità rese disponibili dalla piattaforma. In particolare, verranno illustrate quali metodologie di analisi ogni tecnica utilizza e come esse concorrono al rilevamento di particolari classi di vulnerabilità.

5.1 VulnDetect

Le tecniche che si basano sull'esecuzione simbolica devono tenere conto del problema noto con il nome di *path explosion*: l'incremento delle dimensioni di un programma porta ad un aumento esponenziale dei cammini praticabili presenti sul suo CFG, i quali possono crescere fino a diventare potenzialmente infiniti se il programma contiene cicli non vincolati da una condizione di terminazione [29]. Per evitare questo problema, l'esecutore simbolico può essere guidato in modo tale che esplori solamente una parte del CFG del programma, in modo tale da evitare che esso esplori cammini potenzialmente inutili alla ricerca di vulnerabilità. **VulnDetect** [30] è una tecnica di ricerca per basata su esecuzione simbolica per la ricerca di vulnerabilità di tipo **stack-based buffer-overflow**. L'implementazione di VulnDetect si basa sulle funzionalità offerte dalla piattaforma *angr*. Per evitare il problema della *path explosion*, VulnDetect utilizza un'insieme di metodologie di analisi statica per guidare l'esecutore simbolico ad analizzare solo i cammini potenzialmente vulnerabili del programma.

5.1.1 Buffer overflow e Stack-Based Buffer Overflow

Una delle classi di vulnerabilità software più diffuse nel panorama software moderno è quella che prende il nome di **buffer overflow** (CWE-120). Questo tipo di vulnerabilità si presenta quando non avviene nessun controllo sulla lunghezza massima dei dati che devono essere salvati all'interno di un buffer allocato in memoria. Il dato, quindi, può potenzialmente andare a sovrascrivere i dati posti in sezioni di memoria adiacenti all'area di allocazione del buffer, potenzialmente quindi andando a sovrascrivere dati fondamentali per la corretta esecuzione del programma. Questa problematica è particolarmente rilevante nei linguaggi non *memory safe*, dove il compito di gestione della memoria del programma è affidato al programmatore. Quando il buffer-overflow può avvenire su un buffer allocato nello stack del programma, la vulnerabilità prende il nome di **stack-based buffer overflow** (CWE-121). La presenza di questo tipo di vulnerabilità può portare a

conseguenze catastrofiche: un attaccante, per esempio, potrebbe sovrascrivere l'indirizzo di ritorno della funzione in cui viene dichiarato il buffer, potenzialmente portando il programma ad eseguire codice malevolo presente nell'input (shellcode).

5.1.2 Strategia di analisi

VulnDetect struttura la strategia di analisi in due fasi distinte [30]:

1. **Analisi statica:** Le vulnerabilità di tipo *buffer overflow* sono principalmente causate dall'utilizzo improprio di funzioni pericolose (in C, per esempio *gets*, *scanf*,...) e dalla mancanza di controlli sulla lunghezza dell'input dell'utente. Per migliorare l'efficienza dell'esecuzione simbolica, VulnDetect applica una sequenza di tecniche di analisi statica sul programma:
 - (a) **Identificazione di operazione sensibili:** Innanzitutto, viene caricato il programma tramite la classe *Project* di angr. Successivamente, si procede al recupero degli indirizzi delle funzioni di libreria potenzialmente pericolose, accedendo alle informazioni contenute nella *Procedure Linkage Table* (PLT). Per determinare quali funzioni recuperare dalla PLT, il programma di analisi mantiene al suo interno una **lista di nomi di funzioni potenzialmente pericolose**. Viene infine recuperato, sempre tramite angr, il CFG del programma, il quale viene attraversato in modo tale da individuare **gli indirizzi nel programma in cui avvengono chiamate a funzioni pericolose**.
 - (b) **Program slicing:** Per prima cosa, viene creato il Data Dependency Graph del programma. Dopodiché, viene effettuato *program slicing* per ogni indirizzo di chiamata a funzione pericolosa individuato nel passo precedente, in modo tale da estrarre i segmenti di codice correlati ad ogni chiamata. Chiameremo questi segmenti *hotspot code segments*.
2. **Vulnerability detection:** La ricerca di vulnerabilità viene effettuata eseguendo simbolicamente il programma, simulando l'input dell'utente con un **input simbolico**. Per evitare *path explosion*, l'esecutore simbolico viene guidato durante il processo di esecuzione usando gli *hotspot code segments* ottenuti durante la fase di analisi statica. Così facendo, l'esecuzione simbolica esplorerà solamente i cammini legati alla ricerca della vulnerabilità, invece di tutti i cammini praticabili del CFG. Quando l'esecuzione arriva ad un punto del programma dove avviene una chiamata ad una funzione pericolosa, è necessario continuare da quel punto l'esecuzione simbolica del programma per assicurarsi che l'operazione che svolge porti ad una vulnerabilità. In caso di presenza di buffer overflow, il registro *EIP* (Extended Instruction Pointer) conterrà un valore simbolico. Questo stato non permette quindi all'esecutore di determinare quale sarà il prossimo stato in cui si troverà il programma e quindi l'esecuzione viene interrotta. In angr, questo tipo di stati del programma prendono il nome di **unconstrained states**. Sarà quindi presente nel programma una vulnerabilità di tipo *stack-based buffer overflow* se l'esecutore simbolico trova almeno uno stato *unconstrained* durante l'esecuzione.

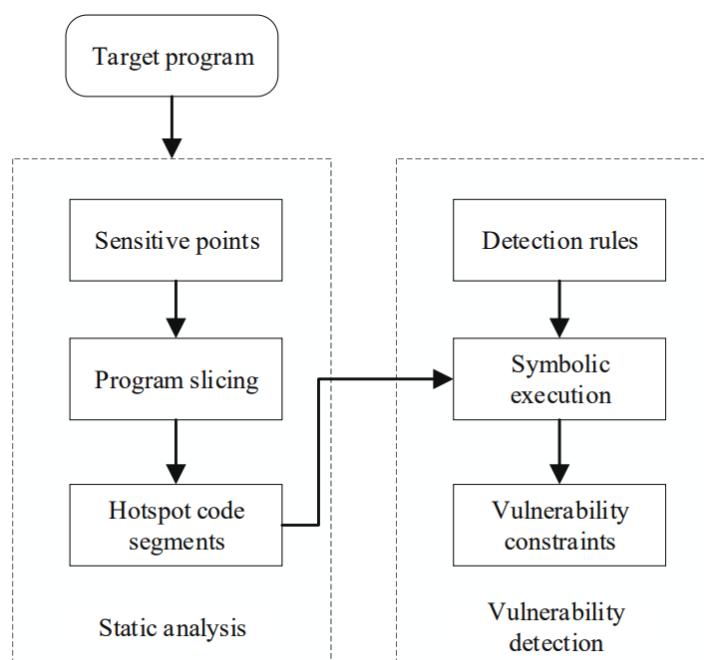


Figura 5.1: Processo di analisi impiegato da VulnDetect. Immagine proveniente da [30]

5.2 Arbiter

Oltre alle problematiche già discusse nel *capitolo 2*, le tecniche di analisi dinamica soffrono anche di un significativo problema di *scalabilità*. In particolare, l'esecuzione simbolica dinamica presenta una pessima scalabilità quando applicata su file binari di grandi dimensioni. A questo si aggiunge il problema, presente in tutte le tecniche di analisi dinamica, della bassa copertura in termini di codice analizzato. Per essere utile all'analista, una tecnica di ricerca automatica delle vulnerabilità deve soddisfare due requisiti cruciali: la **generalità**, ossia la capacità di poter essere impiegata su un ampio spettro di programmi, e l'**accuratezza**, la quale richiede di introdurre un numero di falsi positivi quanto più minimale possibile. Le tecniche di **analisi ibrida** si configurano quindi come una scelta particolarmente adatta a questo scopo, in quanto combinano i punti di forza sia di analisi dinamica e statica, permettendo così di mitigarne le problematiche. **Arbiter** [31] è una tecnica ibrida di ricerca automatica di vulnerabilità, progettata per individuare la presenza di diverse classi di vulnerabilità attraverso una combinazione di analisi statiche e dinamiche. La metodologia di analisi proposta da *Arbiter* permette di mitigare il numero di falsi positivi introdotti dalle tecniche di analisi statica, mantenendo al contempo una buona scalabilità al crescere delle dimensioni del programma analizzato. L'implementazione delle tecniche di analisi usate da *Arbiter* sfrutta le funzionalità messe a disposizione dal framework di analisi *angr*.

5.2.1 Property-Compliant Vulnerabilities

Arbiter è capace di rilevare tutte quelle vulnerabilità che rispettano la seguente serie di proprietà [31]:

- **(P1): Sensibilità al flusso dei dati:** Le vulnerabilità che sono sensibili al flusso dei dati nel programma (*data-flow sensitive*) possono essere rilevate effettuando un ragionamento riguardante il flusso dei dati dalle sorgenti di input ai sink vulnerabili. È necessario notare che il numero di questo tipo di vulnerabilità è strettamente maggiore rispetto al numero di vulnerabilità rilevabili effettuando solamente una *taint-analysis*; quest'ultime, infatti, includono solamente quelle vulnerabilità causate da una **mancata sanificazione** dell'input dell'utente.
- **(P2): Sorgenti e sink facilmente identificabili:** Le sorgenti di input e i loro sink devono essere facilmente identificabili, cioè devono poter essere identificati tramite l'analisi di artefatti che è possibile produrre in maniera computazionalmente efficiente, come per esempio, un CFG.
- **(P3): Aliasing determinato dal flusso di esecuzione:** Le informazioni riguardanti l'accesso alla stessa area di memoria attraverso *aliasing* devono essere recuperabili semplicemente analizzando il flusso d'esecuzione del programma. Questa proprietà risulta vera solamente quando il programma non effettua nessuna **deferenziazione di puntatori** oppure quando la deferenziazione può essere **risolta ad un singolo oggetto** determinato interamente dal flusso di esecuzione del programma.

Chiameremo una vulnerabilità *property-compliant* (PC) quando essa rispetta tutte e tre le proprietà descritte sopra. Tuttavia, è necessario notare che la condizioni sulla deferenziazione dei puntatori descritte nella proprietà *P3* portano l'analisi a barattare ad una perdita di generalità in cambio di un aumento dell'efficienza.

5.2.2 Vulnerability descriptions

Per effettuare l'analisi per ogni classe di vulnerabilità *property-compliant*, *Arbiter* utilizza una descrizione, specifica per ogni vulnerabilità, delle proprietà descritte sopra chiamata *Vulnerability description* (VD). Una VD è una **rappresentazione programmatica in linguaggio python** dei vari artefatti statici e simbolici inerenti ad ogni proprietà che una vulnerabilità *property-compliant* deve rispettare. Una VD contiene le seguenti funzioni:

- **specify_sources(binary):** Questa funzione deve ritornare un dizionario contenente tutte le sorgenti di input (funzioni) potenzialmente malevole del programma. Ad ogni chiave del dizionario può essere associato il valore numerico 0, il quale indicherà ad *Arbiter* di tracciare il flusso di dati generato dal valore di ritorno della funzione, oppure un valore numerico $i > 0$, il quale indicherà ad *Arbiter* di tracciare il flusso di dati generato dal parametro i -esimo della funzione.
- **specify_sinks(binary):** Similmente a *specify_sources*, questa funzione deve ritornare un dizionario contenente tutti i potenziali sink (funzioni) dei flussi di dati vulnerabili. Ad ogni funzione nel dizionario è associata una lista di caratteri così strutturata:
 - Se la cella j -esima contiene il carattere n , allora *Arbiter* considererà i flussi di dati che arrivano ad essere utilizzati come parametro j -esimo della funzione.

- Se la cella j -esima contiene il carattere c , allora *Arbiter* ignorerà i flussi di dati che arrivano ad essere utilizzati come parametro j -esimo della funzione
- **apply_constraints(state, sources, sink)**: Questa funzione specifica, utilizzando le funzionalità di *claripy* tramite *angr*, quali sono le condizioni che il flusso di dati da sorgente a sink deve rispettare per essere considerato vulnerabile.

Le VD attualmente presenti in *Arbiter* coprono le seguenti classi di vulnerabilità:

CWE	Descrizione	Sorgente	Sink
CWE-131	Incorrect calculation of buffer size	Qualsiasi operazione aritmetica	1° argomento di malloc()
CWE-134	Controlled format string	Qualsiasi dato in input	Tutte le funzioni della famiglia di printf()
CWE-252	Unchecked return value	Valore di ritorno di setuid()	Qualsiasi operazione che usa il dato generato dalla sorgente
CWE-337	Predictable seed in pseudo-random number generator	Valore di ritorno di time()	1° argomento di srand()

Tabella 5.1: Vulnerability description attualmente disponibili in *Arbiter*

Inoltre, è possibile creare delle VD per le seguenti classi di vulnerabilità:

CWE	Descrizione	Sorgente	Sink
CWE-78	OS Command injection	1° argomento di sprintf()	1° argomento di system()
CWE-190	Integer overflow or Wraparound	Operazione aritmetica sul valore di ritorno di sizeof()	Qualsiasi parametro di funzione di tipo size_t
CWE-676	Use of potentially dangerous function	Qualsiasi funzione	1° argomento di strcpy()
CWE-120	Classic buffer overflow	2° argomento di read()	1° argomento di memcpy()

Tabella 5.2: Classi di vulnerabilità per le quali è possibile scrivere una VD

5.2.3 Strategia di analisi

Supponiamo di voler analizzare il programma P per controllare se esso presenta una certa vulnerabilità V . Allora, *Arbiter* struttura la strategia di analisi per V nelle seguenti fasi:

1. **Control Flow Recovery**: Viene innanzitutto effettuato il recupero del Control Flow Graph del programma tramite le funzionalità offerte da *angr*.
2. **Subject Caller Identification**: *Arbiter* utilizza il CFG e le informazioni presenti nella VD di V per identificare nel programma le sorgenti di input potenzialmente malevole e i loro sink.

3. **Data Dependency Recovery:** Viene costruito un DDG per ogni funzione che contiene almeno una sorgente o un sink descritto nella VD di V . Se la VD di V descrive sia sorgenti che sink, allora il DDG costruito per una certa funzione f non conterrà alcuna informazione riguardante le funzioni che chiamano f . Altrimenti, verrà costruito un DDG per ogni funzione, sia chiamata che chiamante, che contiene almeno un sink descritto nella VD, invece di costruire in DDG che copra tutte le potenziali sorgenti di input. Questo meccanismo rende la generazione del DDG per il programma più scalabile, al costo, tuttavia, di una perdita di precisione; la quale verrà corretta durante la fase di verifica finale dei risultati.
4. **Backward Slicing e Filtering:** L'identificazione dei flussi potenzialmente vulnerabili avviene tracciando i cammini che collegano le sorgenti ai sink sul DDG e generando i **sotto-grafi** corrispondenti a tali cammini. Tale processo è equivalente all'effettuare un *program slicing* statico sul programma per ogni flusso di interesse.
5. **Under Constrained Symbolic Execution (UCSE):** I flussi di dati raccolti durante le fasi precedenti potrebbero non essere realmente vulnerabili: la path condition che caratterizza un determinato flusso di dati potrebbe infatti risultare **insoddisfacibile** oppure potrebbe **prevenire il manifestarsi della vulnerabilità V di interesse**. Per ogni flusso di dati individuato, *Arbiter* esegue una *UCSE* sulla *slice* corrispondente in modo da recuperare le relazioni simboliche tra i dati nel cammino dalla sorgente al sink. Se le relazioni recuperate soddisfano i vincoli specificati nella VD di V , allora viene segnalata una potenziale vulnerabilità.
6. **Adaptive False Positive Reduction:** Come già discusso nel capitolo 4, uno dei problemi della UCSE è la mancanza di informazione riguardante il contesto di esecuzione della funzione analizzata. Per mitigare il numero di falsi positivi introdotti da questa problematica, *Arbiter* effettua la seguente serie di operazioni:
 - (a) Quando viene trovata una potenziale vulnerabilità in una funzione f , *Arbiter* effettua una ricerca tutti i punti nel programma dove f viene chiamata
 - (b) Per ogni punto di chiamata C individuato, viene ripetuta la UCSE includendo il contesto dato da C

Questa serie di operazioni viene ripetuta **ricorsivamente** fino a raggiungere una **profondità massima di ricorsione**, la quale è configurabile dall'analista in base al tempo disponibile per effettuare l'analisi.

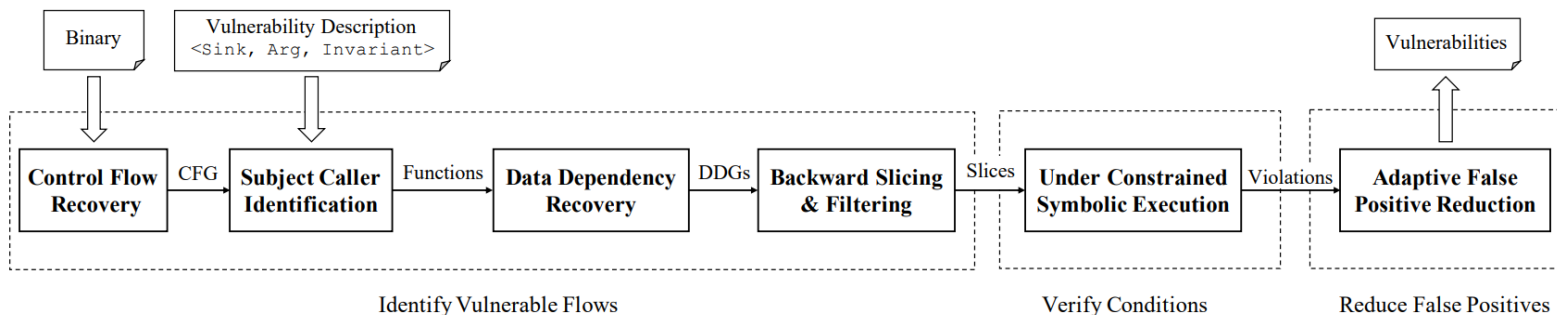


Figura 5.2: Processo di analisi implementato da Arbiter. Figura proveniente da [31]

Capitolo 6

Implementazione

In questo capitolo, si fornirà una descrizione dettagliata dell'architettura della piattaforma Binoculars. In particolare, verranno illustrate le implementazioni sia del frontend che del backend, illustrando come la loro collaborazione permette all'utente di usare semplicemente le funzionalità offerte dalla piattaforma.

6.1 Architettura

L'architettura della piattaforma si suddivide in due componenti principali:

- **Frontend:** Il frontend dell'applicazione comprende la logica di caricamento dei file binari e la presentazione dei risultati delle analisi. Questa componente è stata sviluppata utilizzando **SvelteKit**, un framework leggero e robusto per lo sviluppo di applicazioni web.
- **Backend:** Il backend dell'applicazione consiste in un server web sviluppato utilizzando il framework python **Flask**. In particolare, il server si occupa delle seguenti mansioni
 - Gestione dei file binari caricati dall'utente
 - Gestione della sessione HTTP
 - Decompiling e Disassembly del file binario
 - Applicazione delle diverse tecniche di analisi disponibili e generazione della risposta in formato *JSON*

6.2 Backend

Il server web è implementato in un singolo file *wsgi.py* e utilizza il protocollo di trasmissione *Web Server Gateway Interface* (WSGI) per comunicare con il frontend. In particolare, vengono esposti al frontend i seguenti endpoint:

- **/upload:** Questo endpoint gestisce l'upload di file binari da parte degli utenti. Il file caricato deve essere un eseguibile *ELF* valido; in caso affermativo, esso verrà memorizzato sulla piattaforma con un nome univoco. Questo meccanismo permette l'upload di file con nome uguale da parte di utenti diversi. Un'utente può caricare in piattaforma solamente un file alla volta.

- **/disassemble**: Questo endpoint gestisce l'attuazione del processo di disassembly sul file caricato. Questa operazione, come già esplicitato nel capitolo 4, si sull'utilizzo della libreria python *Capstone*. Alla termine del processo di disassembly, viene restituita una risposta HTTP contenete un file *JSON* con la seguente struttura:

```

1 {
2   "disassembly": [
3     {
4       "instructions": [
5         {
6           "addr": "INDIRIZZO DELL'ISTRUZIONE",
7           "mnemonic": "OPERANDO",
8           "op_str": "OPERANDI"
9         },
10        ...
11      ]
12    }
13  ]
14 }
```

Listing 6.1: Struttura del file JSON ritonrato dall'endpoint /disassemble

- **/decompile**: Questo endpoint gestisce il decompiling del file caricato tramite il framework *Ghidra*. L'utilizzo da parte di *Binoculars* di entrambi *Ghidra* e *angr* fa emergere un **conflitto sull'utilizzo dell'SMT solver Z3**, il quale è utilizzato da entrambi i framework e non può essere utilizzato in un processo da più di un oggetto software alla volta. Per risolvere questo problema, il processo di decompilazione del file viene eseguito in un processo separato, il quale verrà terminato una volta completato il decompiling. Al termine dell'operazione, viene restituita una risposta HTTP contenente un file *JSON* così strutturato:

```

1 [
2   {
3     "code": "CODICE DELLA FUNZIONE",
4     "functions": "NOME DELLA FUNZIONE"
5   },
6   ...
7 ]
```

Listing 6.2: Struttura del file JSON ritonrato dall'endpoint /decompile

- **/analyses/{Analysis}**: Questa famiglia di endpoint gestisce l'applicazione di una tecnica di analisi sul file eseguibile caricato. In particolare, *Analysis* può assumere due possibili valori:

- **vulndetect**: Verrà applicata la strategia di analisi implementata da *vulndetect*. Alla fine dell'analisi, verrà restituita una risposta HTTP contenente un file *JSON* con la seguente struttura:

```

1 {
2   "INDIRIZZO_OPERAZIONE_VULNERABILE": {
3     "Description": "DESCRIZIONE DELLA VULNERABILITA'",
4     "Vulnerability_found": "CWE-121"
5   }, ...
6 }
```

Listing 6.3: Struttura del file JSON ritornato da vulndetect

- **arbiter**: Verrà applicata la strategia di analisi implementata da *Arbiter*. Alla fine dell'analisi, verrà restituita una risposta HTTP contenente un file *JSON* con la seguente struttura:

```
1 {  
2   "CWE131": true/false ,  
3   "CWE134": true/false ,  
4   "CWE252": true/false ,  
5   "CWE337": true/false  
6 }
```

Listing 6.4: Struttura del file JSON ritornato da arbiter

Il codice del server web è riportato nell'*appendice A* di questo documento.

6.2.1 Configurazione

Per garantire una più ampia flessibilità alla piattaforma, l'analista può configurare i seguenti parametri del server web tramite un'apposito file *JSON* di configurazione:

- **Posizione cartella di upload**: Indica dove i file binari caricati dall'utente devono essere salvati.
- **Cartella di installazione di Ghidra**: In molte distribuzioni di Linux, *Ghidra* è reperibile direttamente tramite il package manager della distribuzione. Questo parametro permette quindi all'analista di configurare correttamente la posizione di installazione del framework.
- **Massima profondità di angr**: Indica ad angr la massima profondità di esplorazione del CFG.
- **Cartella dei risultati di arbiter**: Indica dove i file JSON contenenti i risultati ottenuti da *arbiter* devono essere salvati.
- **Cartella dei log di arbiter**: Indica dove i file di log di *arbiter* devono essere salvati.
- **Cartella VD di arbiter**: Indica la posizione in cui si trovano i file contenenti le VD per *arbiter*.
- **Regex per l'individuazione dei file VD**: Per permettere un recupero automatico dei file VD per *arbiter*, viene utilizzato un meccanismo di recupero basato su *regex*. Questo parametro permette all'analista di configurare come i nomi dei file contenenti le VD per *arbiter* debbano essere formattati.
- **Blacklist di funzioni per arbiter**: Indica una lista di funzioni che *arbiter* eviterà di analizzare.

6.2.2 Implementazione delle tecniche di analisi

Le due tecniche di analisi disponibili (*Arbiter* e *VulnDetect*) sono integrate nella piattaforma attraverso l'utilizzo di due **classi wrapper**. Queste classi sono state **progettate** per offrire un punto di accesso semplificato ad entrambe le metodologie. In particolare per

Arbiter, il quale utilizzo prevede un'esecuzione tramite linea di comando, questa scelta è stata essenziale per semplificare l'accesso e l'attivazione della strategia di analisi da parte della piattaforma. Il codice delle due classi wrapper è riportato nell'*appendice A* di questo documento.

6.3 Frontend

Il frontend della piattaforma consiste in due pagine sviluppate tramite il framework *Svelte*:

- **Pagina principale:** La pagina principale dell'applicazione. Essa permette all'utente di caricare un file binario tramite un apposito pulsante di upload.
- **Pagina di analisi:** Questa pagina è dedicata all'analisi del file binario caricato dall'utente. Essa mostra i risultati del disassembly e del decompiling del programma e permette all'utente di lanciare una della analisi presenti sulla piattaforma e di ottenerne i risultati.

Inoltre, le due pagine condividono una **navbar**, la quale è stata sviluppata come componente indipendente, in modo che essa possa essere riutilizzata facilmente. Il *routing* tra la pagina principale e la pagina di analisi è gestito direttamente da Svelte attraverso il suo meccanismo di filesystem-based routing, il quale definisce le route dell'applicazione in base alla posizione dei file nelle directory del progetto. Questo meccanismo è stato utilizzato anche per implementare un **proxy**, così da poter effettuare le chiamate necessarie al backend dell'applicazione senza la necessità di dover configurare manualmente le politiche *CORS* (*Cross-Origin Resource Sharing*) per ogni chiamata. Gli endpoint esposti dal *proxy* sono:

- **/api/upload:** Effettua una chiamata all'endpoint */upload*.
- **/api/disassemble:** Effettua una chiamata all'endpoint */disassembly*.
- **/api/decompile:** Effettua una chiamata all'endpoint */decompile*.
- **/api/analyses/{Analysis}:** Effettua una chiamata all'endpoint */analyses/{Analysis}*, dove *Analysis* può essere *vulndetect* oppure *arbiter*.

6.3.1 Pagina principale

La pagina principale dell'applicazione è la prima pagina che appare all'utente. Essa presenta in fondo una descrizione delle funzionalità della piattaforma (raggiungibile anche utilizzando il tasto *about* della navbar) e un pulsante, il quale permette all'utente di caricare un file eseguibile *ELF* in piattaforma. In particolare, effettuare il caricamento di un file binario genera una chiamata all'endpoint */api/upload*, il quale, a sua volta effettuerà una chiamata all'endpoint */upload* del backend. Se il caricamento del file ha successo (il file che l'utente sta cercando di caricare è un file *ELF* valido), allora l'utente verrà reindirizzato alla pagina di analisi del binario. Altrimenti, verrà mostrato un messaggio di errore all'utente, informandolo che il caricamento del file non è andato a buon fine.

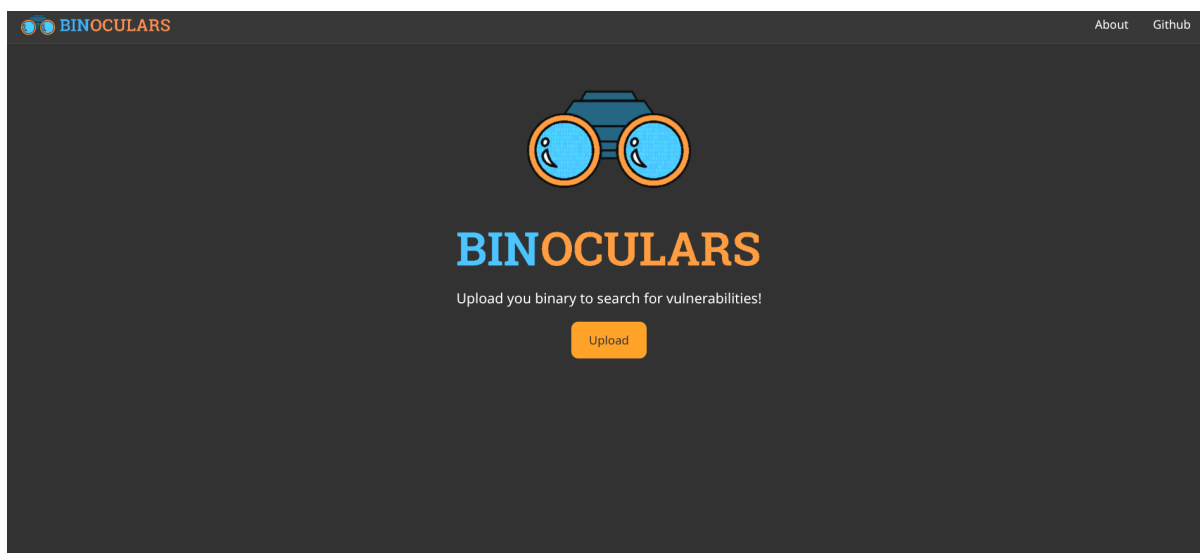


Figura 6.1: Pagina principale dell'applicazione

```
1 import { error, json } from "@sveltejs/kit";
2
3 export async function POST({ request }) {
4   // Gets the file from the form
5   const formData = await request.formData();
6   const file = formData.get("file");
7
8   // Creates a new formData since the previous can't be transmitted
9   // again due to it being
10  // tied to the form request
11  const forwardFormData = new FormData();
12  forwardFormData.append("file", file, file.name);
13
14  const backEndResponse = await fetch("http://localhost:5000/upload",
15  {
16    credentials: "include",
17    method: "POST",
18    body: forwardFormData
19  });
20
21  return new Response(backEndResponse.body, {
22    status: backEndResponse.status,
23    headers: backEndResponse.headers
24  })
25 }
```

Listing 6.5: Proxy per l'operazione di caricamento di un file

6.3.2 Pagina di analisi

La pagina di analisi è divisa in tre sezioni:

- **Disassembled code:** Questa sezione contiene il codice assembly ottenuto dal processo di disassembly del binario.
- **Decompiled code:** Questa sezione contiene il codice *C-Like* ottenuto dal processo di decompiling tramite *Ghidra*.

- **Vulnerability analyses:** Questa sezione presenta una maschera per la selezione della strategia di analisi da applicare. Dopo la selezione, la maschera mostra gli eventuali **parametri di input** che l'utente deve fornire prima di avviare l'analisi.

Quando la pagina viene caricata, vengono effettuate immediatamente due chiamate a `/api/disassembly` e `/api/decompile` per ottenere rispettivamente il *disassembly* e il codice *decompilato* del programma. Qualora una di queste due chiamate non vada a buon fine, verrà visualizzato un messaggio di errore all'utente nella sezione associata alla chiamata fallita. Altrimenti, verrà mostrato, nella sezione associata alla chiamata, il codice ottenuto. Infine, l'utilizzo, da parte dell'utente, di una qualsiasi delle tecniche di analisi disponibili genera una chiamata a `/api/{Analysis}`. Se la chiamata ha successo, al termine dell'analisi verranno presentati i risultati ottenuti all'utente. Altrimenti, verrà mostrato un messaggio di errore.

The screenshot shows the BINOCULARS application interface. It is divided into three main sections: Disassembled code, Decompiled code, and Vulnerability analyses. The Disassembled code section shows a table with columns for Address, Operation, and Operands. The Decompiled code section shows a list of lines of code. The Vulnerability analyses section has a dropdown menu with the text 'Choose one of the available analysis!'.

Figura 6.2: Pagina di analisi dell'applicazione senza alcun risultato

The screenshot shows the BINOCULARS application interface with the 'Vulnerability analyses' section populated. It includes input fields for 'Length of stdin input' (1000), 'Number of CMD parameters' (3), 'Parameter 1 length' (3000), 'Parameter 2 length' (3000), and 'Parameter 3 length' (3000). There are 'Start analysis!' and 'Go back' buttons. Below these is a table with columns for Address, Description, and Vulnerability. The table contains one entry: Address 0x401149, Description 'A potential instruction pointer hijack by user input was detected', and Vulnerability 'CWE-121: Stack-based buffer overflow'.

Figura 6.3: Pagina di analisi dell'applicazione con i risultati ottenuti da *VulnDetect*

The screenshot shows the BINOCULARS application interface. It is divided into three main sections:

- Disassembled code:** A table with columns 'Address', 'Operation', and 'Operands'. It shows assembly instructions for a function, starting with a note about GNU property and then listing instructions like 'add', 'xor', 'push', and 'return'.
- Decompiled code:** A table with columns 'Line' and 'Code'. It shows the corresponding C code for the assembly instructions, including function initialization, variable declarations, and conditional logic.
- Vulnerability analyses:** A section with buttons 'Start analysis!' and 'Go back'. Below it is a table with columns 'Vulnerability' and 'Result', showing results for CWE131, CWE134, CWE252, and CWE337, all marked as 'false'.

Figura 6.4: Pagina di analisi dell'applicazione con i risultati ottenuti da *Arbiter*

```

1 export async function POST({ request }) {
2   const backEndResponse = await fetch("http://localhost:5000/analyses
  /{Analysis}", {
3     method: "POST",
4     headers: request.headers,
5     body: request.body,
6     credentials: "include",
7     duplex: "half"
8   });
9   return new Response(backEndResponse.body, {
10     status: backEndResponse.status,
11     headers: backEndResponse.headers
12   });
13 }

```

Listing 6.6: Struttura di un proxy per le operazioni di analisi. *Analysis* può assumere il valore *vulndetect* oppure *arbiter*

```

1 export async function GET( { request }) {
2   const cookie = request.headers.get("cookie") || "";
3   const backEndResponse = await fetch("http://localhost:5000/{
  Operation}", {
4     headers: request.headers,
5     credentials: "include"
6   });
7   return new Response(backEndResponse.body, {
8     status: backEndResponse.status,
9     headers: backEndResponse.headers
10   });
11 }

```

Listing 6.7: Struttura di un proxy per le operazioni di disassemblaggio e decompilazione. *Operation* può assumere il valore *disassemble* oppure *decompile*

```
1 export async function sendData(endpoint, body, handleResultsFunction,
2   handleErrorFunction) {
3   let error = null;
4   let data = null;
5   try {
6     const response = await fetch(endpoint, {
7       method: "POST",
8       body: body,
9       credentials: "include"
10    });
11    data = await response.json();
12    return handleResultsFunction({ data });
13  } catch(e) {
14    console.error(`${TAG} Something went wrong!`, e);
15    return handleErrorFunction({ error });
16  }
17 }
```

Listing 6.8: Funzione *sendData* utilizzata dal codice riportato nel listing 6.9

```
1 import { onMount } from "svelte";
2 let disassemblyState = {
3   disassembly: null,
4   error: null,
5   isLoading: true
6 };
7 let decompileState = {
8   decompile: null,
9   error: null,
10  isLoading: true
11 };
12 function fetchData() {
13   getData("/api/disassemble",
14     ( {data} ) => {
15     disassemblyState.disassembly = data.disassembly;
16     disassemblyState.isLoading = false;
17   }, ( {error} ) => {
18     disassemblyState.error = error.message;
19     disassemblyState.isLoading = false;
20   }
21 );
22   getData("/api/decompile",
23     ( {data} ) => {
24     decompileState.decompile = highlightCodeLines(data, hijs);
25     decompileState.isLoading = false;
26   }, ( {error} ) => {
27     decompileState.error = error.message;
28     disassemblyState.isLoading = false;
29   }
30 );
31 }
32 onMount(async () => {
33   fetchData();
34 });
```

Listing 6.9: Codice che svolge il recupero del disassembly e del decompiling tramite l'interrogazione del backend. Questo codice viene eseguito immediatamente dopo il caricamento della pagina di analisi

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 (ICCCBB)*, 2018, pp. 1–8. DOI: 10.1109/ICCCBB.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] R. Baldoni, E. Coppà, D. C. D’elia, C. Demetrescu e I. Finocchi, «A Survey of Symbolic Execution Techniques,» *ACM Comput. Surv.*, vol. 51, n. 3, mag. 2018, ISSN: 0360-0300. DOI: 10.1145/3182657. indirizzo: <https://doi.org/10.1145/3182657>.
- [22] 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.
- [23] C. G. Cifuentes, «Reverse compilation techniques,» 1994. indirizzo: <https://api.semanticscholar.org/CorpusID:110021381>.
- [24] 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>.

- [25] Capstone, *Capstone documentation*, Accesso effettuato il 13 ottobre 2025, 2014. indirizzo: <https://www.capstone-engine.org/documentation.html>.
- [26] C. Eagle e K. Nance, *The Ghidra Book: The Definitive Guide*. No Starch Press, 2020, p. 608, ISBN: 978-1718501027.
- [27] Y. Shoshitaishvili et al., «SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis,» in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 138–157. DOI: 10.1109/SP.2016.17.
- [28] E. Coppa, *Symbolic Execution*, Accesso effettuato il 14 ottobre 2025. indirizzo: <https://www.slideshare.net/slideshow/symbolic-execution/124783109>.
- [29] S. Anand, P. Godefroid e N. Tillmann, «Demand-Driven Compositional Symbolic Execution,» in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan e J. Rehof, cur., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 367–381, ISBN: 978-3-540-78800-3.
- [30] W. Wang, P. Zhwng, G. Wei, Z. Ge, Z. Qin e X. Sun, «Buffer Overflow Vulnerability Detection Based on Static Analysis-assisted Symbolic Execution,» in *2023 4th International Symposium on Computer Engineering and Intelligent Communications (ISCEIC)*, 2023, pp. 546–550. DOI: 10.1109/ISCEIC59030.2023.10271194.
- [31] J. Vadayath et al., «Arbiter: Bridging the Static and Dynamic Divide in Vulnerability Discovery on Binary Programs,» in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA: USENIX Association, ago. 2022, pp. 413–430, ISBN: 978-1-939133-31-1. indirizzo: <https://www.usenix.org/conference/usenixsecurity22/presentation/vadayath>.

Appendice A

Codice relativo al backend

Questa appendice contiene il codice sviluppato per il backend della piattaforma.

A.1 Server web

Il codice del server web è suddiviso nelle seguenti funzioni:

```
1 @app.route("/upload", methods=["POST"])
2 def upload():
3     file = request.files.get("file")
4     try:
5         # Try to parse the file as ELF to validate it
6         ELFFile(file)
7         file.seek(0) # Reset file pointer after validation
8         save_file(file)
9         return make_response("Created", 201)
10    except ELFParseError as err:
11        print(err)
12        # Return 422 Unprocessable Entity if file is not valid ELF
13        return make_response("The uploaded file is not a valid ELF!",
422)
```

Listing A.1: Endpoint */upload*

```
1 @app.route("/decompile", methods=["GET"])
2 def decompile():
3     filepath = session.get("FILEPATH")
4     ghidra_install_folder = app.config["GHIDRA_INSTALL_FOLDER"]
5
6     # Launch external Python script to run Ghidra headless
7     decompilation
8     proc = subprocess.run(
9         [sys.executable, "./ghidra_decompile.py", filepath,
10         ghidra_install_folder],
11         capture_output=True,
12         text=True
13     )
14
15     if proc.returncode != 0:
16         # If subprocess failed, log stderr and return error
17         print(proc.stderr)
18         return make_response(f"{tag} Something went wrong", 500)
```

```

18     try:
19         # Parse the stdout produced by the subprocess as JSON
20         results = json.loads(proc.stdout)
21     except json.JSONDecodeError as e:
22         print(f"{tag} Failed to decode Ghidra output!")
23         return make_response(f"{tag} Failed to decode Ghidra output!",
24                               500)
25     return jsonify(results)

```

Listing A.2: Endpoint */decompile*

```

1 @app.route("/disassemble", methods=["GET"])
2 def disassemble():
3     filepath = session.get("FILEPATH")
4     try:
5         with open(filepath, "rb") as file:
6             elf_object = ELFFile(file)
7
8             # Ensure the binary has a .text section
9             text_section = elf_object.get_section_by_name(".text")
10            if not text_section:
11                return make_response(".text section is missing in the
12                ELF file!", 422)
13
14            # Initialize Capstone disassembler for x86-64
15            md = Cs(CS_ARCH_X86, CS_MODE_64)
16            disassembly = []
17
18            # Iterate through all ELF sections
19            for section in elf_object.iter_sections():
20                if section.name and section.data():
21                    disassembly_section = {
22                        "section": section.name,
23                        "instructions": []
24                    }
25
26                    code = section.data()
27                    addr = section["sh_addr"]
28                    # Disassemble instructions from this section
29                    for i in md.disasm(code, addr):
30                        disassembly_section["instructions"].append({
31                            "address": f"0x{i.address:x}",
32                            "mnemonic": i.mnemonic,
33                            "op_str": i.op_str
34                        })
35                    disassembly.append(disassembly_section)
36
37            # Return structured JSON response with disassembly
38            json_response = jsonify({"disassembly": disassembly})
39            return json_response, 200
40    except Exception as e:
41        print(e)
42        return f"{tag} Something went wrong!", 500;

```

Listing A.3: Endpoint */disassemble*


```

1 @app.route("/analyses/vulndetect", methods=["POST"])
2 def vuln_detect_analysis():
3     detector = VulnDetection(session.get("FILEPATH"))
4
5     # Get and parse parameter length list from request
6     param_len_list_raw = request.form.get("param_lengths")
7     param_len_list = json.loads(param_len_list_raw)
8
9     # Validate param_lengths is a list of integers
10    if not isinstance(param_len_list, list) or not all(isinstance(n,
11    int) for n in param_len_list):
12        return jsonify({"error": "param_lengths must be a list of
13    integers"}), 400
14
15    # Run VulnDetection analysis
16    results = detector.analyze(
17        stdin_input_len=int(request.form.get("stdin_input_length")),
18        max_depth=app.config["ANGR_MAX_DEPTH"],
19        length_of_argv_inputs=param_len_list
20    )
21    return jsonify(results)

```

Listing A.4: Endpoint */analyses/vulndetect*

```

1 @app.route("/analyses/arbiter", methods=["POST"])
2 def arbiter_analysis():
3     # Retrieve file path and name from session
4     filename = session.get("FILENAME")
5     filepath = session.get("FILEPATH")
6     # Get Arbiter configuration values
7     vd_dir = app.config["ARBITER_VD_FOLDER"]
8     regex = app.config["ARBITER_CWE_VD_REGEX"]
9
10    # Initialize Arbiter analysis object
11    analysis = ArbiterAnalysis(
12        os.path.join(app.config["ARBITER_LOGS_FOLDER"], filename),
13        os.path.join(app.config["ARBITER_JSON_RESULTS_FOLDER"],
14    filename),
15        BLACKLIST=app.config["ARBITER_FUNCTIONS_BLACKLIST"]
16    )
17    # Find vulnerability descriptor (VD) scripts matching the regex
18    vds = find_vd_files(vd_dir, regex)
19
20    print(f"Found VDs: {vds}")
21
22    results = {}
23
24    # Run Arbiter analysis for each VD script
25    for vd in vds:
26        try:
27            print(f"Analyzing for {vd}")
28            analysis.analyze(os.path.join(vd_dir, vd), filepath)
29            results[vd.replace(".py", "")] = True
30        except Exception:
31            print(f"This file does not present vulnerability {vd}")
32            results[vd.replace(".py", "")] = False
33    return results

```

Listing A.5: Endpoint */analyses/arbiter*

```

1 def find_vd_files(directory, regex):
2     matched_files = []
3     try:
4         regex = re.compile(regex)
5         for root, dirs, files in os.walk(directory):
6             for filename in files:
7                 if regex.match(filename):
8                     matched_files.append(filename)
9     except Exception as e:
10        print(e)
11    finally:
12        return matched_files

```

Listing A.6: Funzione *find_vd_files*

```

1 def save_file(file):
2     # Generate a unique filename to avoid collisions
3     filename = f"{uuid.uuid4().hex}_{secure_filename(file.filename)}"
4     # Ensure upload directory exists (consider moving this to server
5     # startup)
6     os.makedirs(app.config['BIN_UPLOAD_FOLDER'], exist_ok=True)
7     filepath = os.path.join(app.config["BIN_UPLOAD_FOLDER"], filename)
8     # Store file path and name in session
9     session["FILEPATH"] = filepath
10    session["FILENAME"] = filename
11    # Save the uploaded file
12    file.save(filepath)

```

Listing A.7: Funzione *save_file*

A.2 Classi wrapper per le analisi

Questa sezione contiene il codice delle classi wrapper che permettono l'accesso alle tecniche di analisi alla piattaforma.

A.2.1 VulnDetect

Come già trattato nel *capitolo 3*, VulnDetect mantiene una lista di funzioni vulnerabili:

```

1 import angr
2 import claripy
3 from enum import Enum
4 from angr import SimValueError
5
6 # A list of potentially dangerous functions
7 dangerous_functions = [
8     # Input functions
9     "gets", "getc", "getchar", "getwd", "fgets", "read",
10    "scanf", "fscanf", "sscanf", "vscanf", "vfscanf", "vsscanf",
11
12    # String copying functions (no bounds checking)
13    "strcpy", "stpcpy", "wcscpy", "memcpy", "wmemcpy", "strncpy", "
14    wcsncpy",
15
16    # String concatenation functions
17    "strcat", "wscat", "strncat", "wcsncat",

```

```

17
18     # Output functions (can be format string vulnerable)
19     "sprintf", "vsprintf", "snprintf", "vsnprintf", "fprintf", "
20     fprintf", "printf",
21
22     # Memory handling
23     "memmove", "bcopy", "alloca",
24
25     # Path handling
26     "realpath", "tempnam", "tmpnam", "mktemp", "mkstemp",
27
28     # Network or file IO (depending on context)
29     "recv", "recvfrom", "readlink", "open",
30
31     # Misc
32     "gets_s",
33     "strtok",
34     "setenv", "putenv",
35     "__builtin___sprintf_chk", "__builtin___memcpy_chk", # GCC
    builtins with overflow potential

```

Listing A.8: Funzioni considerate potenzialmente pericolose da *VulnDetect*

Il codice della classe wrapper per *VulnDetect* può essere suddiviso nei seguenti metodi:

```

1 class VulnDetection:
2     """
3     This class implements the "VulnDetect" detection of buffer
4     overflows that can overwrite the instruction pointer .
5     A detailed description of the analysis method can be found in the
6     associated paper:
7     (https://ieeexplore.ieee.org/document/10271194)
8     """
9
10    def __init__(self, binary_path: str):
11        """
12        Constructor for the VulnDetection class
13        :param binary_path: the path of the binary to analyze
14        """
15
16        # TAG for debugging purposes
17        self.TAG = "[VulnDetect]"
18        # Initialize the project
19        self.project = angr.Project(binary_path, auto_load_libs = False
20    )
21
22    # Create the Control Flow Graph (CFG) for the binary
23    self.CFG = self.project.analyses.CFGEmlated(keep_state = True)
24    # Create the Control Dependence Graph (CDG) for the binary
25    self.CDG = self.project.analyses.CDG(cfg = self.CFG)
26    # Create the Data Dependence Graph (DDG) for the binary
27    self.DDG = self.project.analyses.DDG(cfg = self.CFG)
28    # List to hold the points where the vulnerable functions are
29    called
30    self.unsafe_call_points = []
31    # List to hold the backward slicing results
32    self.backward_slicing_results = []

```

Listing A.9: Metodo init della classe wrapper per *VulnDetect*

```

1  def __static_analysis(self):
2      """
3      This function implements the static analysis module of
4      VulnDetect
5      """
6      # Get the Procedure-Linkage-Table (PLT)
7      plt = self.project.loader.main_object.plt
8
9      for function in dangerous_functions:
10         try:
11             # Get the function address from the PLT
12             unsafe_function_addr = plt[function]
13             # Get all the nodes whose address matches the unsafe
14             function one
15             unsafe_nodes = self.CFG.model.get_all_nodes(
16             unsafe_function_addr)
17             for node in unsafe_nodes:
18                 # Get the list of dangerous functions call point
19                 address from the precursors nodes
20                 for predecessor in node.predecessors:
21                     if predecessor is not None and isinstance(
22                     predecessor, angr.analyses.cfg.cfg_base.CFGNode):
23                         code_location = angr.code_location.
24                         CodeLocation(predecessor.addr, None)
25                         self.unsafe_call_points.append(
26                         code_location)
27         except KeyError:
28             continue
29
30     if not self.unsafe_call_points:
31         print(f"{self.TAG} No unsafe call points found. Nothing to
32         slice.")
33         return
34
35     print(f"{self.TAG} Found dangerous function call points: {self.
36     unsafe_call_points}")
37
38     for location in self.unsafe_call_points:
39         # Perform backward slicing
40         bs = self.project.analyses.BackwardSlice(
41             self.CFG,
42             self.CDG,
43             self.DDG,
44             [location]
45         )
46         result = {
47             "target": hex(location.block_addr),
48             "slice": bs
49         }
50         self.backward_slicing_results.append(result)

```

Listing A.10: Metodo static_analysis della classe wrapper per *VulnDetect*

```

1  def __vulnerability_detection(self, stdin_input_len: int, max_depth
: int, length_of_argv_inputs: list, ):
2      """
3      This function implements the vulnerability detection module of
VulnDetect
4      :param stdin_input_len The length of the symbolic input that
will be passed to the binary via stdin
5      :param max_depth Maximum depth of the exploration
6      :param length_of_argv_inputs The length of the various command
line arguments
7      """
8
9      vulnerable_states = []
10     for program_slice in self.backward_slicing_results:
11         # Create a symbolic input of stdin_input_len bytes
12         stdin_sym_input = claripy.BVS("stdin_input",
stdin_input_len * 8)
13
14         # Initialize symbolic command line arguments
15         argv_inputs = [self.project.filename]
16         for i in range(len(length_of_argv_inputs)):
17             argv_inputs.append(claripy.BVS(f"argv_input_{i+1}",
length_of_argv_inputs[i]))
18
19         # Create the initial state of the binary
20         init_state = self.project.factory.entry_state(stdin =
stdin_sym_input, args = argv_inputs)
21         # Get the sensitive point address
22         target = program_slice["target"]
23         # Create a simulation manager instance
24         simgr = self.project.factory.simulation_manager(init_state,
save_unconstrained = True)
25
26         # Start symbolic execution in order targeting the sensitive
point:
27         simgr.explore(find = int(target, 16))
28
29         # If the relevant state is found, make active only that
state
30         if simgr.found is not None:
31             simgr.drop(stash = "active")
32             simgr.move(from_stash = "found", to_stash = "active")
33
34         counter = 0
35         while len(simgr.unconstrained) == 0 and counter <=
max_depth:
36             counter += 1
37             simgr.step() # Execute one step of symbolic execution
38
39             if len(simgr.unconstrained) > 0:
40                 vulnerable_states.append((simgr.unconstrained[0],
target))
41         return vulnerable_states

```

Listing A.11: Metodo vulnerability_detection della classe wrapper per *VulnDetect*

```

1  def analyze(self, stdin_input_len: int, max_depth: int,
2  length_of_argv_inputs: list):
3      """
4      This function starts the analysis and returns its results
5      """
6      # Execute static analysis on the binary
7      self.__static_analysis()
8      # Get the vulnerable states using symbolic execution
9      vulnerable_states = self.__vulnerability_detection(
10     stdin_input_len, max_depth, length_of_argv_inputs)
11
12     response = dict()
13     for state_tuple in vulnerable_states:
14         state = state_tuple[0]
15         addr = state_tuple[1]
16         # Get the instruction pointer
17         ip = state.regs.ip
18         # Create a junk address the size of the IP
19         target_junk_addr = b"A" * (ip.size() // 8)
20
21         # Check if the program counter can point to the junk
22         address
23         if state.satisfiable(extra_constraints = [state.regs.ip ==
24         target_junk_addr]):
25             response[f"{addr}"] = {
26                 "Vulnerability_found": "CWE-121: Stack-based buffer
27                 overflow",
28                 "Description": "A potential instruction pointer
29                 hijack by user input was detected",
30             }
31     return response

```

Listing A.12: Metodo analyze della classe wrapper per *VulnDetect*

A.2.2 Arbiter

Il codice della classe wrapper per *Arbiter* può essere suddiviso nei seguenti metodi:

```

1  from arbiter.master_chief import *
2  class ArbiterAnalysis:
3      """
4      This class is a wrapper around Arbiter's analysis framework.
5      It offers an easily accessible entry point to setup the environment
6      , load
7      the VD, configuring logging and run the analysis
8      """
9      def __init__(self, LOG_DIR = None, JSON_DIR = None, CALL_DEPTH = 1,
10     STRICT_MODE = False, IDENTIFIER = None, CALLER_LEVEL = -1,
11     BLACKLIST = []):
12
13         # Directories for logs and JSON results
14         self.LOG_DIR = LOG_DIR
15         self.JSON_DIR = JSON_DIR
16
17         # Analysis configuration
18         self.CALL_DEPTH = CALL_DEPTH
19         self.STRICT_MODE = STRICT_MODE
20         self.IDENTIFIER = IDENTIFIER
21         self.LOG_LEVEL = logging.DEBUG

```

```

18     self.CALLER_LEVEL = CALLER_LEVEL
19
20     # Functions to ignore during analysis
21     self.BLACKLIST = BLACKLIST
22
23     # Suppress angr's internal logging (to reduce noise)
24     logging.getLogger('angr').setLevel(logging.CRITICAL)

```

Listing A.13: Metodo init della classe wrapper per *Arbiter*

```

1     def __enable_logging(self, vd, target):
2         """
3         Configures Arbiter logging system for different modules
4         """
5         vd = Path(vd).stem
6         target = Path(target).stem
7         # List of Arbiter internal loggers to configure
8         loggers = ['sa_recon', 'sa_advanced', 'symbolic_execution']
9         for logger in loggers:
10             l = logging.getLogger(f"arbiter.master_chief.{logger}")
11             # Log format: timestamp, logger name, log level, message
12             formatter = logging.Formatter('%(asctime)s - %(name)s - %(
levelname)s - %(message)s')
13             if self.LOG_DIR is not None:
14                 # Log file: arbiter_<vd>_<target>.log
15                 handler = logging.FileHandler(f"{self.LOG_DIR}/arbiter_
{vd}_{target}.log")
16                 handler.setFormatter(formatter)
17                 l.addHandler(handler)
18
19             l.setLevel(self.LOG_LEVEL)

```

Listing A.14: Metodo enable_logging della classe wrapper per *Arbiter*

```

1     def __setup(self, vd_path: str, target_path: str):
2         """
3         Prepares the analysis environment
4         """
5         vd = Path(vd_path)
6         target = Path(target_path)
7
8         # Check if vulnerability definition (VD) exists
9         if not vd.exists():
10             sys.stderr.write(f"Error: {vd} does not exist\n")
11             raise Exception("VD does not exist")
12         # Check if target binary exists
13         elif not target.exists():
14             sys.stderr.write(f"Error: {target} does not exist\n")
15             raise Exception("Target does not exist")
16
17         # Dynamically import the VD module
18         try:
19             spec = util.spec_from_file_location(vd.stem, vd.absolute().
as_posix())
20             template = util.module_from_spec(spec)
21             spec.loader.exec_module(template)
22         except:
23             sys.stderr.write(f"Error could not import VD: {vd}\n")
24             raise Exception("Error importing the VD")

```

```

25
26     # Create LOG_DIR if necessary
27     if self.LOG_DIR:
28         Path(self.LOG_DIR).mkdir(parents=True, exist_ok=True)
29         if Path(self.LOG_DIR).exists():
30             self.LOG_DIR = Path(self.LOG_DIR).resolve().as_posix()
31         else:
32             sys.stderr.write(f"Directory {self.LOG_DIR} does not
exist and we could not create it\n")
33             self.__enable_logging(vd, target)
34
35     # Create JSON_DIR if necessary
36     if self.JSON_DIR:
37         Path(self.JSON_DIR).mkdir(parents=True, exist_ok=True)
38         if Path(self.JSON_DIR).exists():
39             self.JSON_DIR = Path(self.JSON_DIR).resolve().as_posix
()
40         else:
41             sys.stderr.write(f"Directory {self.JSON_DIR} does not
exist and we could not create it\n")
42
43     return template, target

```

Listing A.15: Metodo setup della classe wrapper per *Arbiter*

```

1     def analyze(self, vd, target):
2         """
3         Main entry point to run Arbiter's vulnerability analysis:
4         """
5         # Initialize angr project without loading external libraries
6         project = angr.Project(target, auto_load_libs=False)
7         # Setup VD, logging, and directories
8         template, target = self.__setup(vd, target)
9         # Reconnaissance and sink identification
10        sink_map = template.specify_sinks()
11        sa = SA_Recon(project, sinks=sink_map.keys(), maps=sink_map,
json_dir=self.JSON_DIR)
12        if self.IDENTIFIER is None:
13            sa.analyze(ignore_funcs=self.BLACKLIST)
14        else:
15            sa.analyze_one(self.IDENTIFIER)
16        # Advanced static analysis and source discovery
17        sources = template.specify_sources()
18        sb = SA_Adv(sa, checkpoint=sources, require_dd=self.STRICT_MODE
, call_depth=self.CALL_DEPTH, json_dir=self.JSON_DIR)
19        sb.analyze_all()
20        # Symbolic execution
21        constrain = template.apply_constraint
22        se = SymExec(sb, constrain=constrain, require_dd=self.
STRICT_MODE, json_dir=self.JSON_DIR)
23        se.run_all()
24        # Post-processing
25        return template.get_results(se.postprocessing(pred_level=self.
CALLER_LEVEL))

```

Listing A.16: Metodo analyze della classe wrapper per *Arbiter*