UNIVERSITÀ
DEGLI STUDI
DI BRESCIA

DIPARTIMENTO DI INGEGNERIA
DELL'INFORMAZIONE

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea

Analisi sperimentale del fuzzing con AFL: riproduzione e valutazione di vulnerabilità software

Laureando:
Bergamini Brian

Anno Accademico 2024/2025

Indice

Capitolo 1

Introduzione

La crescente complessità del software moderno e la sua diffusione in ambiti critici, come i sistemi embedded, le applicazioni web e le librerie multimediali, rendono la sicurezza un requisito fondamentale. Una singola vulnerabilità in una libreria ampiamente utilizzata può avere conseguenze critiche su un'enorme quantità di applicazioni, aprendo la strada a crash imprevisti o, nei casi più gravi, ad attacchi mirati. Per questo motivo, negli ultimi anni, la ricerca di tecniche efficaci per individuare vulnerabilità nel software è diventata una priorità. Tra i metodi più promettenti per il testing automatico del software troviamo il **fuzzing**, una tecnica che prevede l'immissione di dati non validi, imprevisti o casuali come input tramite appositi programmi detti *fuzzer*.

I fuzzer sono impiegati tipicamente per testare software che elaborano input strutturati, ossia dati o istruzioni organizzati secondo un formato predefinito (ad esempio, file JSON, immagini PNG, pacchetti di rete). La funzione del codice di parsing[1] è quella di validare tali input, separando quelli conformi (accettabili) da quelli non conformi. Un fuzzer efficace deve generare input simili al formato atteso ma con lievi alterazioni, assicurando così che non vengano immediatamente scartati dal parser. Superando il filtro iniziale, questi input possono creare comportamenti imprevisti nella logica interna del programma e far emergere i "corner case" (o casi limite) che non sono stati gestiti correttamente dagli sviluppatori.

[1]Parser: processo che analizza un flusso continuo di dati in ingresso, in modo da determianre la corretezza della sua struttura grazie ad una grammatica formale.

1.1 Contesto

Il lavoro effettuato per questa tesi rientra nell'ambito della verifica della sicurezza del software, con particolare attenzione alla libreria libpng, che si occupa di immagini in formato png. Questa tipologia di librerie è ampiamente usata in progetti open-source. Il fuzzing negli anni si è dimostrato uno degli approcci più efficaci per scoprire bug di tipo memory corruption, crash e condizioni non gestite, perchè combina mutazioni automatiche con informazioni di copertura per esplorare percorsi di esecuzione rilevanti. L'obiettivo di questa tesi è quello di condurre una campagna sperimentale di fuzzing mirata a libpng usando American Fuzzy Lop (AFL) all'interno del framework MAGMA [1], raccogliendo i crash e gli input rilevanti prodotti durante l'esecuzione. Una volta ottenuti i crash più significativi, analizzarli in dettaglio mediante tecniche standard di debugging (ad esempio utilizzando GDB [2]) per l'identificazione della causa scatenante. Lo scopo finale è quello di documentare almeno un caso di Root Cause Analysis (RCA) completo che metta in relazione la mutazione/splicing dell'input generato dal fuzzer con il comportamento anomalo della libreria libpng[2].

1.2 Struttura tesi

La tesi è strutturata nei seguenti capitoli:

- **Capitolo 1 -** *Introduzione:* capitolo appena affrontato che tratta il contesto, le motivazioni e gli obiettivi della tesi.
- **Capitolo 2 -** *Background teorico:* panoramica sul testing del software con approfondimenti sul fuzzer AFL e la libreria libpng.
- **Capitolo 3 -** *Ambiente sperimentale:* descrizione dell'ambiente sperimentale adottato, comprendente il sistema operativo Debian, il framework di valutazione MAGMA, la piattaforma Docker e le configurazioni necessarie all'avvio della campagna di fuzzing.

[2]GNU Debugger: è uno strumento a riga di comando gratuito per il debugging di programmi, che consente agli sviluppatori di esaminare il comportamento di un software in esecuzione per identificare e correggere bug.

- **Capitolo 4 -** *Fuzzing:* presentazione e analisi quantitativa dei risultati della campagna di fuzzing.
- **Capitolo 5 -** *Discussione:* valutazione critica della campagna e suggerimenti per migliorare la robustezza della libreria target libpng.
- **Capitolo 6 -** *Conclusioni:* sintesi dei contributi, lezioni apprese e possibili sviluppi futuri.

Capitolo 2

Background teorico

Prima di descrivere nel dettaglio la metodologia e gli esperimenti condotti, è utile introdurre i concetti teorici e gli strumenti fondamentali che costituiscono le basi di questa tesi. In questo capitolo verranno presentati i principi del fuzzing e poi verrà analizzato in dettaglio il funzionamento del fuzzer American Fuzzy Lop. Verranno inoltre spiegate le caratteristiche principali della libreria libpng, scelta come target del fuzzing, e la struttura interna del formato PNG, che servirà per comprendere al meglio il crash analizzato nella Root Cause Analysis.

2.1 Fuzzing

A differenza delle tradizionali metodologie di test del software basati su casi costruiti manualmente, il fuzzing[3] è caratterizzato da un'elevata automazione e dalla capacità di generare input nel tentativo di provocare un arresto anomalo e poter quindi individuare errori che altrimenti non verrebero scoperti. Tali errori di codice rappresentano aree potenzialmente ad alto rischio per le minacce alla sicurezza. Un elemento centrale del fuzzing è la natura degli input prodotti. L'idea originaria era quella di inviare dati completamente casuali, mentre i fuzzer moderni adottano strategie più sofisticate: partono da un corpus iniziale di file validi (seed corpus) e applicano mutazioni mirate affinchè gli input risultanti siano sufficientemente stutturati da attraversare le prime fasi del parsing, ma al tempo stesso abbastanza diversi da portare il programma in stati non previsti.

Il fuzzing può essere classificato in tre categorie:

- **Black-box fuzzing**:

  il fuzzer non riceve alcuna informazione sull'esecuzione interna del programma. L'input viene generato in modo casuale o basato su semplici euristiche e si osservano solo eventualmente crash o timeout. È il metodo meno efficiente, poiché difficilmente riesce a produrre input che progressivamente esplorano parti significative del codice.

- **White-box fuzzing**:

  Sfrutta completamente il codice sorgente del programma e tecniche di analisi per generare input che soddisfino vincoli specifici. Offre un controllo molto fine sul percorso di esecuzione, ma è estremamemnte costoso in termini computazionali e difficilmente applicabile a software esteso o ricco di branch.

- **Grey-box fuzzing**:

  Rappresenta il compromesso più diffuso e utilizzato nei fuzzer moderni, tra cui AFL. Il fuzzer ha accesso a una quantità minima di informazioni interne, tipicamente la code coverage, ossia una misura delle porzioni di codice effettivamente eseguite durante l'elaborazione di un input. Tale informazione viene utilizzata come meccanismo di feedback per guidare l'evoluzione dei casi di test. Se una mutazione permette di coprire un nuovo ramo o un nuovo percorso, l'input viene considerato interessante e aggiunto al corpus. Questo modello permette un'esplorazione molto più efficace rispetto al black-box, mantenendo al contempo una complessità computazionale ridotta rispetto al white-box.

2.2 American Fuzzy Lop / libAFL

AFL[4] è un software libero che impiega algoritmi genetici[1] per individuare automaticamente casi di test in grado di attivare nuovi stati interni del software di destinazione. Rilasciato inizialmente nel novembre del 2013, AFL è rapidamente diventato uno dei fuzzer più utilizzati nella ricerca sulla sicurezza. Il codice sorgente è pubblicato su GitHub e il suo nome fa riferimento a una razza di conigli.

AFL richiede che l'utente fornisca un comando per l'esecuzione dell'applicazione da testare e almeno un input valido iniziale. Tale input può essere fornito al programma tramite standard input oppure come file specificato nella linea di comando. Ad esempio, nel caso di un programma che processa file multimediali, un breve file correttamente formattato può essere utilizzato come seed iniziale. Una volta verificato che il comando fornito funzioni correttamente, il fuzzer tenta di ridurre l'input nella forma più semplice possibile, mantenendo lo stesso comportamento osservato. Terminata questa fase preliminare, AFL avvia il processo di fuzzing vero e proprio applicando diverse mutazioni agli input disponibili. La generazione di un crash o di un blocco del programma rappresenta l'indicatore principale della scoperta di un bug o di una potenziale vulnerabilità di sicurezza; in tali casi, l'input responsabile viene salvato per consentire un'analisi successiva.

Per massimizzare l'efficacia del fuzzing, AFL richiede che il programma bersaglio venga compilato tramite specifiche utility di strumentazione, le quali inseriscono nel codice meccanismi per il tracciamento del flusso di controllo a basso livello. Questo approccio consente al fuzzer di rilevare con precisione quando un input induce l'esecuzione di nuovi percorsi di codice, guidando così il processo di generazione dei test in modo coverage-guided.

È opportuno distinguere AFL da `libAFL`[5], un framework di fuzzing più recente che rappresenta un'evoluzione concettuale dell'approccio introdotto da AFL. Mentre AFL è un fuzzer monolitico e pronto all'uso, `libAFL` è una libreria modulare progettata per consentire la realizzazione di fuzzer personalizzati. libAFL

[1] Algoritmo genetico: sono una classe di algoritmi di ottimizzazione ispirati ai meccanismi dell'evoluzione biologica. Partono da una popolazione iniziale di soluzioni candidate e, attraverso operazioni come mutazione, crossover (ricombinazione) e selezione, producono nuove soluzioni potenzialmente migliori.

fornisce componenti riutilizzabili per la gestione di mutazioni, feedback, scheduler ed esecuzione dei target, lasciando allo sviluppatore la definizione delle politiche di fuzzing più appropriate. In questo lavoro, tuttavia, l'attenzione è rivolta esclusivamente all'utilizzo di AFL come strumento di analisi sperimentale, senza ricorrere allo sviluppo di fuzzer custom basati su `libAFL`.

2.2.1 Corpus e seed

Il corpus rappresenta l'insieme degli input utilizzati dal fuzzer come base per la generazione dei casi di test. Esso costituisce una componente centrale del fuzzing coverage-guided, poiché determina lo spazio di partenza da cui vengono derivate le mutazioni successive. All'avvio della campagna di fuzzing, il corpus è inizializzato a partire da uno o più *seed*, ovvero input validi forniti dall'utente o dal benchmark, che rispettano il formato atteso dal programma target. I seed svolgono un ruolo fondamentale nelle fasi iniziali del fuzzing: input ben formati consentono al programma di superare i controlli di parsing e di raggiungere porzioni di codice più profonde, aumentando le probabilità di esplorare nuovi percorsi di esecuzione. Ogni seed può essere considerato un singolo elemento del corpus iniziale. Durante l'esecuzione della campagna, il corpus viene aggiornato dinamicamente. Ogni input generato dal fuzzer che consente di raggiungere un nuovo percorso di esecuzione o di incrementare la copertura del codice viene promosso ed inserito nella queue[2] di AFL, ovvero la struttura dati che rappresenta operativamente il corpus attivo. In questo modo, il corpus evolve progressivamente, adattandosi al comportamento del programma target e guidando l'esplorazione di nuove aree del codice.

2.2.2 Mutator e generator

Un *mutator* è il componente del fuzzer responsabile della modifica degli input esistenti al fine di produrre nuovi casi di test. Le mutazioni possono essere semplici, come la modifica di singoli bit o byte, oppure più complesse, come l'inserimento di valori particolari o la combinazione di parti di input differenti. AFL mette a

[2]Queue: cartella contenente tutti gli input che sono stati salvati dal fuzzer

disposizione un insieme di mutazioni predefinite, progettate per esplorare in modo sistematico e casuale lo spazio degli input.
Di seguito sono descritte le principali tipologie di mutazione utilizzate dal fuzzer:

- **Flip1/2/4 (Bit Flipping)**: mutazioni che invertono rispettivamente 1, 2 o 4 bit consecutivi lungo l'input. Questa tecnica consente di individuare rapidamente condizioni di errore legate a singoli bit o a campi di dimensioni ridotte.

- **Arith8/16/32 (Arithmetic Increment/Decrement)**: il fuzzer seleziona rispettivamente un byte (8 bit), una word (16 bit) o una double word (32 bit) e applica incrementi o decrementi di piccoli valori interi. Questo tipo di mutazione è particolarmente efficace nel sollecitare controlli sui limiti e verificare la corretta gestione dei valori numerici.

- **Int8/16/32 (Interesting Values)**: porzioni dell'input vengono sostituite con valori predefiniti noti per essere critici, come valori di soglia, massimi o minimi rappresentabili. Tali valori sono frequentemente associati a errori di overflow, underflow o accessi a memoria non validi.

- **EXT_AO (Extras / Dictionary)**: il fuzzer inserisce o sostituisce sequenze di byte specifiche, dette *magic bytes*, che il programma target si aspetta di elaborare. Nel caso dei file PNG, esempi tipici includono marker di formato come *IDAT*.

Il concetto di *generator*, invece, si riferisce alla creazione di input completamente nuovi, non derivati direttamente da seed esistenti. Mentre AFL si basa prevalentemente su mutazioni incrementali di input validi, altri framework di fuzzing possono adottare generatori basati su modelli, grammatiche o specifiche del formato. Questo approccio risulta particolarmente utile per formati complessi, ma richiede una conoscenza approfondita della struttura degli input.
Nel contesto di AFL, il processo di fuzzing è quindi principalmente mutation-based: l'efficacia del fuzzer dipende dalla qualità del corpus iniziale e dalla capacità dei mu-

tators di produrre variazioni significative che permettano di superare nuovi controlli logici e semantici del programma bersaglio.

2.2.3 Stages di AFL: deterministic, havoc e splicing

AFL organizza il processo di fuzzing in diverse fasi (*stages*), ciascuna caratterizzata da strategie di mutazione differenti. Questa suddivisione consente di bilanciare l'esplorazione sistematica e l'esplorazione casuale dello spazio degli input.

Nella fase **deterministica**, AFL applica mutazioni semplici e riproducibili, come il flipping e l'incremento o decremento di valori numerici (ovvero le tecninche Flip e Arith citati nella sezione 2.2.2). Questa fase mira ad esplorare in modo ordinato le variazioni più immediate degli input, ed è particolarmente efficace nelle fasi iniziali della campagna.

La fase **havoc** introduce, invece, mutazioni fortemente casuali e combinate. In questa modalità, AFL seleziona casualmente diverse operazioni di mutazione e le applica in sequenza sullo stesso input. Havoc consente di generare input altamente distorti, aumentando la probabilità di attivare condizioni di errore non raggiungibili tramite mutazioni deterministiche.

Figura 2.1: Screenshot di input generati nel havoc stage.

Infine, la fase di **splicing** viene attivata quando il fuzzer fatica a trovare nuovi percorsi di esecuzione. In questa fase, AFL combina porzioni di due input distinti presenti nella queue, che hanno coperto percorsi differenti, creando un nuovo input ibrido. Lo splicing consente di esplorare combinazioni di stati interni del programma che non emergerebbero da una singola mutazione incrementale.

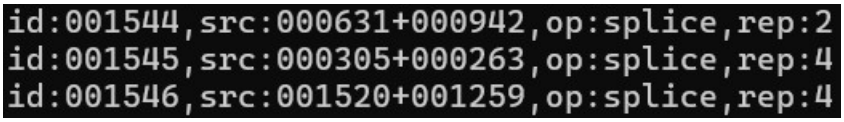


Figura 2.2: Screenshot di input generati nello splice stage.

2.2.4 Ottimizzazioni

L'efficacia del fuzzing dipende in larga misura dal numero di test che possono essere eseguiti nell'unità di tempo. Per questo motivo, AFL adotta diverse tecniche di ottimizzazione dell'esecuzione volte a ridurre l'overhead associato al lancio ripetuto del programma target. Una delle ottimizzazioni principali è l'utilizzo del `fork server`, che evita la creazione completa di un nuovo processo per ogni input. Il programma target viene inizializzato una sola volta e successivamente duplicato tramite `fork`, riducendo significativamente il costo di avvio. In altri contesti di fuzzing avanzato, possono essere utilizzate tecniche di `snapshotting`, che permettono di ripristinare rapidamente lo stato iniziale del programma prima dell'elaborazione di ogni input. Sebbene AFL non utilizzi direttamente snapshotting a livello applicativo, il principio sottostante è analogo: minimizzare il lavoro ripetuto per massimizzare il throughput dei test. Queste ottimizzazioni rendono AFL particolarmente adatto a campagne di fuzzing su larga scala, in cui l'esecuzione efficiente di milioni o miliardi di test rappresenta un fattore determinante per l'individuazione di vulnerabilità.

2.3 Struttura PNG

Portable Network Graphics (PNG)[6] è un formato di file utilizzato in informatica per memorizzare immagini. È stato introdotto nel 1995 ed è caratterizzato da compressione lossless, ovvero una compressione senza perdita di informazioni durante le fasi di compressione e decompressione. Il PNG nasce come alternativa libera al formato GIF, con l'obiettivo di fornire un metodo di arichiviazione più moderno che andasse a superare alcune limitazioni tecniche del suo predecessore. Il formato è concepito come un contenitore strutturato, organizzato in una sequenza di blocchi chiamati *chunk*, ciascuno dei quali ha un ruolo specifico nella rappresentazione

dell'immagine. Questa architettura rende PNG relativamente facile da analizzare tramite librerie come libpng. I primi 8 byte di un file PNG contengono sempre i seguenti valori:

```
Esadecimale:  89  50  4E  47  0D  0A  1A  0A
ASCII:         .   P   N   G   .   .   .   .
```

```
8950 4e47 0d0a 1a0a 0000 000d 4948 4452  .PNG........IHDR
```

Figura 2.3: Signature del file PNG evidenziata in rosso.

Questa firma indica che il resto del file contiene una singola immagine PNG, costituita da una serie di blocchi che iniziano con un blocco IHDR[7] e terminano con un blocco IEND.
Ogni blocco è composto da quattro parti:

- **Length**: è un numero intero non negativo di 4 byte, indica esattamente quanti byte sono contenuti nei *chunk data* successivi.
- **Chunk Type**: è un codice di 4 caratteri ASCII (di 4 byte) che definisce la natura o il contenuto dei dati del segmento. Alcuni dei più importanti sono:
  - *IHDR*: segmento di intestazione dell'immagine (contiene altezza, larghezza, tipo di colore, ...).
  - *IDAT*: contiene i dati dell'immagine. Quest'ultima può essere suddivisa in più chunk IDAT consecutivi.
  - *PLTE*: contiene la palette dei colori (obbligatoria per le immagini indicizzate).
  - *IEND*: segna la fine dell'immagine.
- **Chunk Data**: è il contenuto vero e proprio del chunk. La natura dei dati dipende dal chunk type.
- **CRC**: il Cyclic Redundancy Check è un valore di controllo di 4 byte calcolato sul chunk Type e sul chunk Data, per verificare l'integrità del blocco.

2.4 Libpng

La progenitrice di tutte le librerie PNG è **libpng**, libreria di riferimento gratuita utilizzata da molte delle applicazioni che supportano il formato PNG. Sviluppata negli anni '90, è scritta prevalentemente in linguaggio C e si appoggia alla libreria zlib per le operazioni di compressione e decompressione di dati. Uno degli obiettivi principali di libpng è garantire la portabilità e la robustezza, per questo motivo viene fornita come libreria indipendente e facilmente integrabile in applicazioni molto diverse tra loro. Questo ampio utilizzo fa sì che eventuali vulnerabilità presenti nel codice abbiano un impatto potenzialmente significativo, motivo per cui libpng è oggetto di attività di fuzzing e analisi della sicurezza.
La fase di parsing dei file PNG è gestita dalle funzioni interne della libreria, e si basa principalmente su due strutture dati fondamentali[8]:

- **pngstruct**: struttura utilizzata per gestire lo stato interno del processo di lettura o scrittura (qui dentro sono contenuti i parametri necessari al flusso di parsing).
- **pnginfo**: struttura che contiene i metadati del file PNG; memorizza le informazioni descrittive dell'immagine.

Il flusso tipico di utilizzo della libreria prevede l'inizializzazione del lettore, la verifica della signature PNG e la successiva lettura sequenziale dei chunk tramite funzioni dedicate (ad esempio *pngread*). Ogni chunk viene validato attraverso il controllo della sua lunghezza, del tipo e del CRC, garantendo che la struttura del file sia coerente prima di procedere alle fasi successive.

Capitolo 3

Ambiente sperimentale

La valutazione sperimentale delle tecniche di fuzzing richiede un ambiente di esecuzione controllato e riproducibile, in grado di garantire la coerenza dei risultati ottenuti. In questo capitolo viene descritto il contesto sperimentale all'interno del quale è stata condotta la campagna di fuzzing, con particolare attenzione alle scelte tecnologiche adottate e alle configurazioni utilizzate.

3.1 Debian

Gli esperimenti sono stati condotti su un sistema operativo Linux, scelta motivata dalla diffusione di tale piattaforma nell'ambito dello sviluppo e testing del software. In particolare, è stata utilizzata la distribuzione Debian[9], selezionata per la sua stabilità ed attenzione alla sicurezza. Una distribuzione Linux consiste in un insieme coerente di software costruito attorno al kernel Linux, comprendente il sistema di base, le librerie e gli strumenti necessari al funzionamento del sistema operativo. L'utilizzo di Debian ha consentito di disporre di un ambiente affidabile e riproducibile, adeguato alla compilazione dei programmi target, all'esecuzione del fuzzer e alla raccolta sistematica dei risultati sperimentali.

3.2 Hardware

Gli esperimenti di fuzzing descritti in questa tesi sono stati eseguiti all'interno di una macchina virtuale ospitata su piattaforma *Proxmox*, al fine di garantire isolamento dell'ambiente di esecuzione e controllo delle risorse disponibili. La macchina virtuale è stata configurata con le seguenti caratteristiche hardware:

- **CPU host**: 32 core virtuali su processore Intel Xeon Gold 6140 (frequenza di base 2.3 GHz)
- **Memoria RAM**: 128 GB

La configurazione della CPU in modalità *host* consente alla macchina virtuale di sfruttare direttamente le funzionalità del processore fisico, riducendo l'overhead di virtualizzazione e fornendo prestazioni adeguate per carichi di lavoro computazionalmente intensivi come il fuzzing feedback-based. Questa configurazione hardware è stata scelta per supportare l'elevato numero di esecuzioni richiesto dalla campagna di fuzzing e per minimizzare l'impatto di eventuali colli di bottiglia legati alle risorse di sistema sui risultati sperimentali.

3.3 MAGMA

Sebbene il fuzzing si sia dimostrato una tecnica estremamente efficace, il confronto oggettivo tra fuzzer diversi risulta complesso. Le metriche comunemente adottate, come il numero di crash rilevati, sono imprecise, poichè molte esecuzioni anomale possono essere riconducibili allo stesso difetto di programmazione, portando a una sovrastima dei risultati ottenuti. Nella valutazione di un fuzzer, la natura delle vulnerabilità presenti nel software target riveste un ruolo determinante. È possibile distinguere due principali categorie di bug utilizzate nei benchmark di fuzzing:

- **Bug sintetici**: vulnerabilità iniettate automaticamente nel codice tramite script, come nel caso del benchmark LAVA-M. Questo approccio consente di generare un elevato numero di programmi target; tuttavia, le condizioni di attivazione dei bug risultano spesso artificiali e poco rappresentative della complessità del software reale.

- **Bug reali**: difetti di programmazione effettivamente presenti in versioni di produzione del software. MAGMA adotta una tecnica di *forward-porting* [1] per reintrodurre tali vulnerabilità nelle versioni più recenti dei programmi target, preservandone il contesto logico originale. Questi bug risultano profondamente integrati nella logica applicativa e costituiscono una sfida più realistica per i fuzzer.

Benchmark	Workloads		Bugs		Bug Density	Ground truth
	#	Real/Synthetic	#	Real/Synthetic		
BugBench [35]	17	R	19	R	1.12	◗
CGC [11]	131	S	590	S	4.50	◗
Google FTS [20]	24	R	47	R	1.96	◗
Google FuzzBench [19]	21	R	–	–	–	–
LAVA-M [14]	4	R	2265	S	566.25	✓
UniFuzz [33]	20	R	?	R	?	✗
Open-source software	–	R	?	R	?	✗
Magma	7	R	118	R	16.86	✓

Figura 3.1: Confronto tra benchmark di fuzzing e caratteristiche delle vulnerabilità.

Come riportato nello studio originale relativo al benchmark MAGMA [10], sono stati analizzati sette fuzzer ampiamente utilizzati, tra cui AFL, AFL++ e Honggfuzzer, per un totale superiore a 200.000 ore di CPU. I risultati mostrano come molti fuzzer, pur raggiungendo elevati livelli di copertura del codice, incontrino difficoltà nell'attivare vulnerabilità complesse. I programmi target inclusi in MAGMA non sono stati selezionati casualmente, ma tramite un'analisi statistica basata sulla Principal Component Analysis (PCA), al fine di garantire un insieme eterogeneo di software, differenziati per funzionalità quali parsing, compressione e crittografia.

MAGMA è un benchmark di tipo *ground truth* basato su programmi reali contenenti vulnerabilità reali. In particolare, sono state selezionate vulnerabilità documentate tramite identificatori Common Vulnerabilities and Exposures (CVE)[2] e reintegrate nelle versioni più recenti e stabili del software. Questo approccio consente di valutare i fuzzer su codice moderno mantenendo la presenza di difetti noti.

Per misurare in modo accurato l'efficacia dei fuzzer, MAGMA introduce nel codice dei sensori, detti *oracoli*, che permettono di distinguere tre livelli di successo:

[1] Forward-porting: processo che consiste nell'identificare vulnerabilità reali che sono state corrette in passato e nel reintrodurle manualmente all'interno di versioni software moderne e stabili.

[2] CVE: dizionario standardizzato di vulnerabilità di sicurezza informatica pubblicamente note, che fornisce un identificatore univoco per ogni falla.

- **Reached**: il fuzzer ha eseguito la porzione di codice contenente la vulnerabilità;
- **Triggered**: l'input ha soddisfatto le condizioni logiche del bug, anche in assenza di un crash;
- **Detected**: il fuzzer ha effettivamente rilevato la vulnerabilità tramite un crash.

Target	Drivers	Version	File type	Bugs	Magic values	Recursive parsing	Compression	Checksums	Global state
libpng	`read_fuzzer`, `readpng`	1.6.38	PNG	7	✓	✗	✓	✓	✗
libtiff	`read_rgba_fuzzer`, `tiffcp`	4.1.0	TIFF	14	✓	✗	✓	✗	✗
libxml2	`read_memory_fuzzer`, `xml_reader_for_file_fuzz` `xmllint`	2.9.10	XML	18	✓	✓	✗	✗	✗
poppler	`pdf_fuzzer`, `pdfimages`, `pdftoppm`	0.88.0	PDF	22	✓	✓	✓	✓	✗
openssl	`asn1`, `asn1parse`, `bignum`, `bndiv`, `client`, `cms`, `conf`, `crl`, `ct`, `server`, `x509`	3.0.0	*Binary blobs*	21	✓	✗	✓	✓	✓
sqlite3	`sqlite3_fuzz`	3.32.0	SQL queries	20	✓	✓	✗	✗	✓
php	`exif`, `json`, `parser`, `unserialize`	8.0.0–dev	*Various*	16	✓	✓	✗	✗	✗

Figura 3.2: Caratteristiche dei programmi target inclusi nel benchmark MAGMA e delle rispettive funzionalità di parsing.

3.4 Docker

Per l'esecuzione della campagna di fuzzing è stata adottata la piattaforma di containerizzazione Docker[11], al fine di garantire un ambiente di esecuzione isolato e riproducibile. Tali caratteristiche risultano particolarmente rilevanti nel contesto del fuzzing, dove l'esecuzione prolungata dei fuzzer e la necessità di confrontare i risultati richiedono condizioni sperimentali stabili e coerenti. Docker consente di incapsulare all'interno di un contenitore l'intero stack software necessario alla sperimentazione, includendo il fuzzer, i programmi target e le relative dipendenze. Questo approccio rende l'ambiente di esecuzione indipendente dal sistema host e riduce il rischio di interferenze dovute a differenze di configurazione o a modifiche del sistema operativo. Nel contesto di questa tesi, Docker è stato utilizzato per eseguire il benchmark MAGMA e il fuzzer AFL in conformità alla configurazione prevista dagli autori del

benchmark. L'uso dei container ha permesso di replicare fedelmente le condizioni sperimentali e di facilitare la gestione delle esecuzioni di fuzzing. L'adozione di Docker ha inoltre semplificato la gestione delle dipendenze software, in quanto i programmi target presentano requisiti eterogenei in termini di librerie e strumenti di compilazione. Infine, l'impiego di Docker ha favorito l'automazione delle campagne sperimentali e l'esecuzione di test su lunga durata, rendendo il processo complessivo più robusto e facilmente gestibile.

3.5 Set-up di AFL e del target

La configurazione sperimentale ha previsto la selezione esplicita del programma target *libpng*, scelto in quanto rappresentativo di un software reale ampiamente utilizzato e caratterizzato da una logica di parsing complessa. MAGMA fornisce script dedicati che consentono di costruire automaticamente un'immagine Docker contenente il fuzzer e il programma target opportunamente strumentato. Durante questa fase, il codice del target viene compilato utilizzando la strumentazione di AFL, in modo da abilitare il tracciamento della copertura di codice e il feedback necessario a guidare il processo di fuzzing. Una volta completata la fase di build, la campagna di fuzzing viene avviata tramite uno script che esegue AFL per un intervallo di tempo predefinito. I risultati prodotti durante la campagna, inclusi gli input generati, i crash rilevati e le informazioni di copertura, vengono salvati in una directory condivisa per consentire l'analisi successiva. Per l'avvio delle campagne è stata adottata la modalità manuale di MAGMA, basata su script dedicati, in quanto più adatta a esperimenti controllati e mirati su un numero limitato di target. Questo approccio ha permesso di mantenere un controllo diretto sui parametri sperimentali e di semplificare l'interpretazione dei risultati.

```
cd tools/captain

# Build the docker image for AFL and a Magma target (e.g., libpng)
FUZZER=afl TARGET=libpng ./build.sh

# To start a single 24-hour fuzzing campaign, use the start.sh script
mkdir -p ./workdir
FUZZER=afl TARGET=libpng PROGRAM=libpng_read_fuzzer SHARED=./workdir POLL=5 \
  TIMEOUT=24h ./start.sh
```

Figura 3.3: Istruzioni per l'avvio manuale della campagna di fuzzing.

3.6 Corpus

Nel contesto di questa campagna di fuzzing, il corpus iniziale fornito da MAGMA è costituito da un insieme minimo di input, composto da quattro file PNG intenzionalmente malformati:

- not_kitty_alpha.png
- not_kitty_gamma.png
- not_kitty_icc.png
- not_kitty.png

Il fuzzing non viene eseguito direttamente sulla libreria libpng, bensì su un programma di test (harness)[3] che utilizza le API di libpng per il parsing e l'elaborazione di immagini PNG. Gli input del corpus iniziale sono progettati per superare le fasi iniziali di parsing di tale programma di test, includendo magic number validi e una struttura sintattica minima conforme al formato PNG, pur contenendo alterazioni strutturali mirate che possono indurre comportamenti anomali durante le fasi successive di elaborazione all'interno della libreria. I **magic number** sono sequenze di byte poste all'inizio di un file e utilizzate dai programmi per identificare il formato dell'input. Nel caso del PNG, essi costituiscono una signature fissa (Figura 2.3) che viene verificata prima di procedere con ulteriori elaborazioni. Nel contesto del fuzzing feedback-based, la presenza di controlli sui magic number rappresenta

[3]Test harness: nel contesto del fuzzing, si indica un programma di test minimale progettato per ricevere input generati automaticamente dal fuzzer che invoca in modo controllato le funzionalità della libreria o del componente software oggetto di analisi.

uno dei problemi classici che limitano l'efficacia della generazione di input completamente casuali. Gli input che non presentano valori corretti in tali campi vengono scartati immediatamente nelle prime fasi di esecuzione, senza produrre alcun feedback di copertura significativo. In assenza di feedback, il fuzzer non è in grado di apprendere quali mutazioni siano promettenti, rimanendo bloccato in percorsi di esecuzione superficiali del programma target. L'adozione di un corpus iniziale ridotto ma semanticamente valido rappresenta pertanto una scelta metodologica essenziale per consentire al fuzzer di oltrepassare questi controlli preliminari. AFL utilizza il corpus iniziale come punto di partenza per generare nuovi input tramite mutazioni guidate dalla copertura del codice, applicando modifiche incrementali a file che già soddisfano i vincoli sintattici del formato. Durante l'esecuzione della campagna, il corpus si arricchisce dinamicamente: gli input che consentono di superare ulteriori controlli di validazione, raggiungere nuovi percorsi di esecuzione o incrementare la copertura del codice vengono automaticamente selezionati e aggiunti al corpus. Questo processo incrementale permette al fuzzer di ampliare gradualmente la varietà e la complessità degli input generati, portando l'esplorazione verso porzioni di codice più profonde e complesse, dove è più probabile imbattersi in comportamenti anomali o vulnerabilità.

Capitolo 4

Fuzzing

Il presente capitolo descrive e analizza i risultati della campagna di fuzzing. L'obiettivo è quello di valutare il comportamento del fuzzer in termini di test, individuazione di crash e capacità di attivare vulnerabilità note. L'analisi prende in considerazione sia metriche quantitative, quali il numero di test eseguiti e i crash ottenuti, sia aspetti legati alla natura della vulnerabilità scoperta, attraverso una RCA e una descrizione dettagliata di quest'ultima. Infine vengono discusse possibili strategie di mitigazione della vulnerabilità identificata.

4.1 Numero di test

Nel contesto del fuzzing, un test corrisponde all'esecuzione del programma target su un input generato o mutato dal fuzzer. Il numero complessivo di test eseguiti rappresenta pertanto una misura diretta dell'intensità della campagna di fuzzing e della capacità del fuzzer di esplorare lo spazio degli input. Come discusso nel capitolo precedente, il benchmark MAGMA fornisce un corpus iniziale estremamente ridotto, composto da quattro immagini PNG intenzionalmente corrotte. A partire da questo insieme minimo, AFL genera progressivamente nuovi casi di test applicando mutazioni automatiche agli input esistenti.

Per questa tesi la campagna è durata **24 ore** ed il corpus si è arricchito fino a raggiungere un totale di 1557 input distinti, ciascuno associato a un percorso di esecuzione ritenuto interessante dal fuzzer.

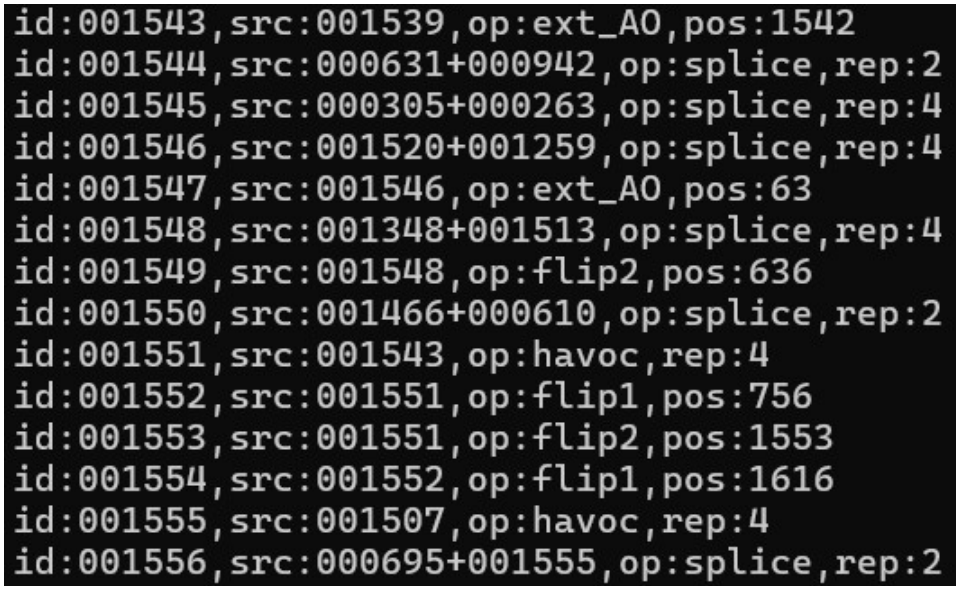


Figura 4.1: Screenshot degli ultimi input salvati nella cartella queue.

Durante l'intera campagna sono stati eseguiti complessivamente circa **480 milioni di test**, con una velocità media di circa **4100 esecuzioni al secondo**. Il fuzzer ha completato **537 cicli** sull'intero corpus, applicando mutazioni di profondità crescente, fino a raggiungere una profondità massima pari a **34**. Questo comportamento indica un'esplorazione progressiva e non superficiale dello spazio degli input.

Nello specifico, il numero di cicli indica quante volte il fuzzer ha passato in rassegna l'intero set di input interessanti (corpus), tentando di mutare ogni file esistente per scoprire nuovi percorsi di esecuzione. La profondità massima, invece, rappresenta il grado di "evoluzione" dei file generati: un valore pari a 34 indica che gli input più complessi sono il risultato di una catena di 34 mutazioni consecutive andate a buon fine, segno che il fuzzer è riuscito a penetrare logiche di controllo profonde e stratificate del software target. Inoltre l'elevata stabilità dell'esecuzione (99,38%) conferma la regolarità della campagna e l'assenza di anomalie sistematiche. Le informazioni relative alla campagna di fuzzing vengono stampate su un file chiamato *fuzzer_stats*.

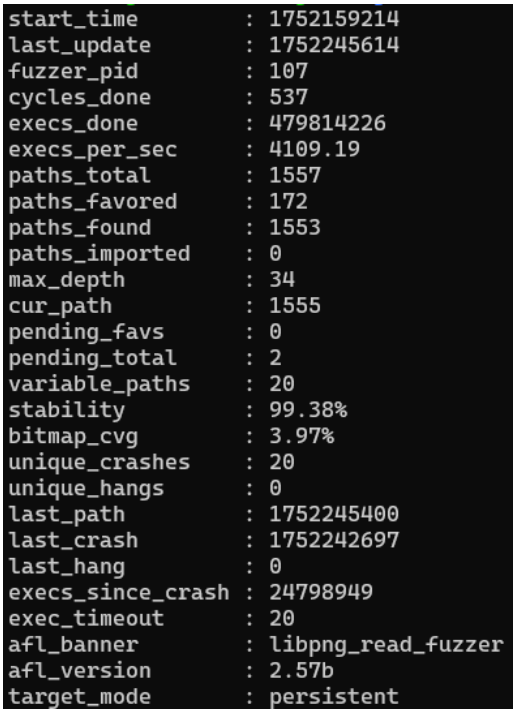


Figura 4.2: Screenshot del file fuzzer_stats, contenente le informazioni relative alla campagna.

4.2 Crash

Nel corso della campagna di fuzzing, AFL ha individuato un totale di **20 crash unici** a fronte di circa 480 milioni di esecuzioni complessive e 537 cicli completati sul corpus iniziale. Un crash si verifica quando l'esecuzione del programma target termina in modo anomalo. Il numero di crash unici rappresenta una metrica più significativa rispetto al numero totale di crash osservati. Questo perchè input diversi possono causare lo stesso errore, portando a una sovrastima del numero di vulnerabilità. AFL identifica i crash unici sulla base del segnale di terminazione e dal punto di esecuzione in cui l'errore si manifesta. L'assenza di hang unici indica che il fuzzer non ha individuato condizioni di blocco prolungato dell'esecuzione, ma esclusivamente terminazioni anomale riconducibli a veri e propri errori di sicurezza o robustezza del programma.

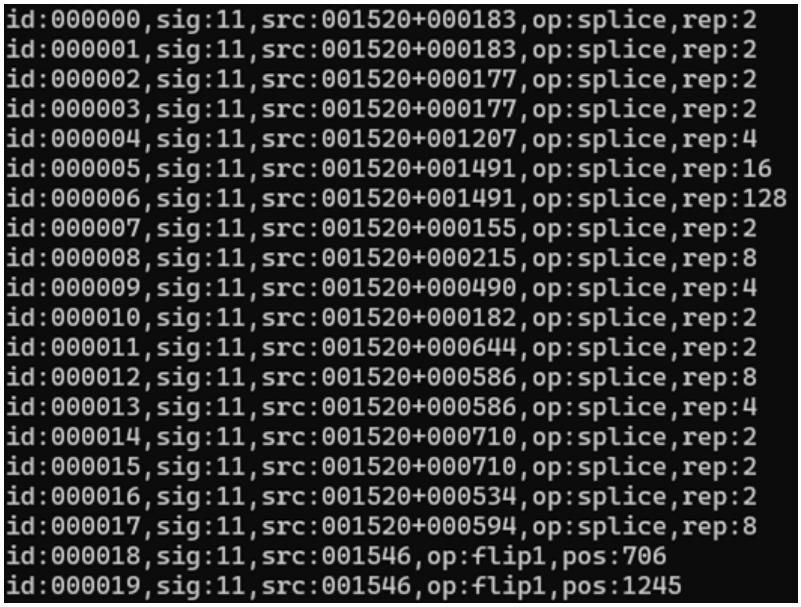


Figura 4.3: Screenshot dei crashes trovati con relative informazioni.

Dallo screenshot appena riportato, possiamo notare che ogni riga è divisa in più campi:

- **Id**: rappresenta l'indice progressivo univoco assegnato dal fuzzer a ogni nuovo crash identificato come unico.
- **Sig**: indica il numero del segnale di sistema che ha causato l'interruzione del programma. Nel caso specifico, il valore 11 corrisponde a SIGSEGV (Segmentation Fault), segnalando che l'input ha indotto il software a tentare un accesso non valido in una zona di memoria.
- **Src**: identifica gli input "genitori" (estratti dalla queue) che sono stati utilizzati come base per le mutazioni. Se compaiono due numeri separati dal segno +, significa che il file è il risultato di una combinazione di due seed diversi.
- **Op**: specifica la tecnica di mutazione finale che ha scatenato il crash.
- **Rep**: specifica quante volte quella particolare operazione è stata ripetuta.
- **Pos**: indica l'offset esatto all'interno del file dove è stata applicata la mutazione. Questo dato è cruciale per l'analisi manuale del file e per identificare quale campo della struttura dati è corrotto.

4.3 Root Cause Analysis

L'analisi approfondita è stata condotta solo su un singolo crash rappresentativo. Il crash è stato analizzato utilizzando strumenti come GNU Debugger (GDB)[1][12], l'ispezione dei metadati del file PNG generato dal fuzzer e l'analisi del codice sorgente coinvolto. L'obiettivo della RCA è stato quello di identificare la causa del crash e di verificare la sua riconducibilità a una vulnerabilità reale presente nella libreria libpng, come previsto dal benchmark MAGMA. Per individuare l'origine del crash, il programma target è stato eseguito sotto il controllo del debugger GDB, analizzando il backtrace e lo stato delle variabili al momento della terminazione anomala. Come mostrato in Figura 4.6, l'esecuzione del programma termina con un segnale di tipo SIGSEGV (Segmentation Fault), indicativo di un accesso non valido alla memoria.

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000427fc7 in png_do_expand_palette (png_ptr=<optimized out>,
    row_info=<optimized out>, row=<optimized out>, palette=0x0,
    trans_alpha=0x0, num_trans=0) at pngrtran.c:4367
4367                    *dp-- = palette[*sp].blue;
(gdb) bt
#0  0x0000000000427fc7 in png_do_expand_palette (
    png_ptr=<optimized out>, row_info=<optimized out>,
    row=<optimized out>, palette=0x0, trans_alpha=0x0, num_trans=0)
    at pngrtran.c:4367
#1  MAGMA_png_do_read_transformations (png_ptr=<optimized out>,
    row_info=<optimized out>) at pngrtran.c:4791
#2  0x000000000041689c in MAGMA_png_read_row (png_ptr=0x773280,
    row=0x775820 "", dsp_row=0x0) at pngread.c:572
#3  0x0000000000402c21 in LLVMFuzzerTestOneInput (data=<optimized out>,
    size=<optimized out>) at contrib/oss-fuzz/libpng_read_fuzzer.cc:201
#4  0x0000000000444954 in ExecuteFilesOnyByOne (argc=2,
    argv=0x7fffffffe3f8) at afl_driver.cpp:167
#5  main (argc=2, argv=0x7fffffffe3f8) at afl_driver.cpp:183
(gdb)
```

Figura 4.4: Screenshot di GDB con backtrace e stato delle variabili.

L'analisi del frame #0 consente di identificare con precisione il punto in cui avviene l'errore. Risulta evidente che il puntatore *palette* assume valore nullo *0x0* al momento dell'accesso. Tale condizione indica che il codice tenta di dereferenziare una struttura dati non inizializzata o assente, in violazione delle assunzioni logiche

[1]GDB: è un debugger che consente di ispezionare il funzionamento interno di un programma durante la sua esecuzione. È inoltre in grado di analizzare retrospettivamente lo stato di un'applicazione nel momento esatto in cui si è verificato un errore critico.

del programma. I frame successivi del backtrace permettono di ricostruire l'intera catena di chiamate che porta all'esecuzione della funzione vulnerabile. Tutto ha origine dal *main*, che invoca `ExecuteFilesOneByOne` per processare gli input generati dal fuzzer. Tale funzione richiama quindi `LLVMFuzzerTestOneInput`, interfaccia standard utilizzata per l'integrazione dei fuzzer con il programma target. Il flusso di esecuzione prosegue all'interno delle funzioni strumentate da MAGMA, in particolare `MAGMA_png_read_row` e `MAGMA_png_do_read_transformations`, responsabili dell'elaborazione delle righe dell'immagine e dell'applicazione delle trasformazioni previste dal formato PNG. All'interno di questo contesto viene infine invocata la funzione `png_do_expand_palette`, incaricata di espandere i pixel indicizzati in valori RGB consultando la palette dei colori, ovvero il chunk PLTE. È proprio in questa fase che l'assenza di una palette valida, non correttamente verificata dalle funzioni chiamanti, causa un'accesso a memoria non valido e la conseguente dereferenziazione di un puntatore nullo, causando il crash del programma.

È stato quindi analizzato anche il file PNG che ha causato il crash. L'ispezione dell'input tramite dump esadecimale mette in evidenza la presenza di un chunk *PLTE* formalmente corretto dal punto di vista strutturale, ma caratterizzato da una lunghezza pari a zero.

```
00000000: 8950 4e47 0d0a 1a0a 0000 000d 4948 4452  .PNG........IHDR
00000010: 0000 0020 0000 0020 0403 0000 006c 5467  ... ... .....lTg
00000020: c700 0000 0050 4c54 4500 68a9 9c00 0000  .....PLTE.h.....
00000030: 0274 524e 5300 ab37 97b7 5800 0000 0162  .tRNS..7..X....b
00000040: 4b47 4400 8805 1d48 0000 0009 6954 5874  KGD....H....iTXt
00000050: 2009 0048 0000 0048 0046 c96b 3e00 0000   ..H...H.F.k>...
00000060: 0970 4859 7300 0000 4800 0000 4800 46c9  .pHYs...H...H.F.
00000070: 6b3e 0000 0067 4944 4154 28cf bdd1 cb0d  k>...gIDAT(.....
00000080: 8020 1045 5162 0956 a0d7 6001 d882 3430  . .EQb.V..`...40
00000090: 0914 6002 fd97 e06f 3383 2b17 7a97 27bc  ..`....o3.+.z.'.
000000a0: 8480 6384 7e80 8e3b ab2e 0637 3a30 3091  ..c.~..;...7:00.
000000b0: 2a96 cf00 0000 1974 4558 7453 6f66 7477  *......tEXtSoftw
000000c0: 6172 6500 4164 6f62 6520 496d 6167 0080  are.Adobe Imag..
000000d0: 5161 6479 71c9 653c 0000 0020 4945 4e44  Qadyq.e<... IEND
000000e0: ae42 6082                                .B`.
```

Figura 4.5: Analisi esadecimale del file PNG che ha generato il crash.
In giallo la lunghezza del chunk, in blu la signature PLTE.

4.4 Mitigation

Il crash analizzato evidenzia una classe di vulnerabilità riconducibile a una validazione semantica insufficiente dell'input. In particolare, il problema non risiede esclusivamente nella dereferenziazione di un puntatore nullo, ma nel fatto che lo stato interno del programma possa raggiungere fasi critiche di elaborazione senza che siano state verificate tutte le precondizioni logiche necessarie. Una strategia di mitigazione efficace richiede l'applicazione sistematica di rigorosi *sanity check*[2], anche in presenza di formati di input standardizzati e ampiamente documentati. Il superamento dei controlli sintattici preliminari non deve infatti essere considerato sufficiente a garantire la coerenza semantica dei dati. In particolare, le funzioni che operano trasformazioni sui dati dovrebbero adottare un approccio *fail fast*[3], interrompendo l'elaborazione in modo controllato qualora le precondizioni logiche non siano soddisfatte. Questo approccio consente di impedire che input malformati o inconsistenti raggiungano istruzioni critiche di accesso alla memoria. In un'ottica di *secure coding*[4], il caso analizzato ribadisce l'importanza di mantenere una corrispondenza rigorosa tra i metadati dell'input e lo stato interno delle strutture dati allocate, verificando sistematicamente la validità dei puntatori e dei vincoli logici prima di qualsiasi dereferenziazione.

[2]Sanity check: In ambito informatico, un sanity check (o controllo di plausibilità) è un test rapido e localizzato volto a verificare la validità di base di un dato o del risultato di un calcolo.

[3]Fail fast: è una filosofia di innovazione che incoraggia a sperimentare rapidamente, identificare i problemi presto e imparare dagli errori velocemente, invece di investire a lungo termine in idee non valide

[4]Secure coding: è un insieme di pratiche, tecniche e linee guida applicate durante lo sviluppo del software per prevenire l'introduzione di vulnerabilità, falle di sicurezza o bug sfruttabili.

Capitolo 5

Discussione

Questo capitolo analizza i risultati emersi dalla campagna di fuzzing, con l'obiettivo di interpretare le evidenze sperimentali in relazione all'efficacia della tecnica adottata e alle sue implicazioni pratiche per la sicurezza del software. In particolare, verranno esaminati i punti di forza del fuzzing coverage-guided nel trattamento di input altamente strutturati, come i file PNG. L'analisi si sposterà poi sulla vulnerabilità specifica rilevata, contestualizzandola attraverso il riferimento alla relativa CVE e approfondendo la strategia di risoluzione adottata.

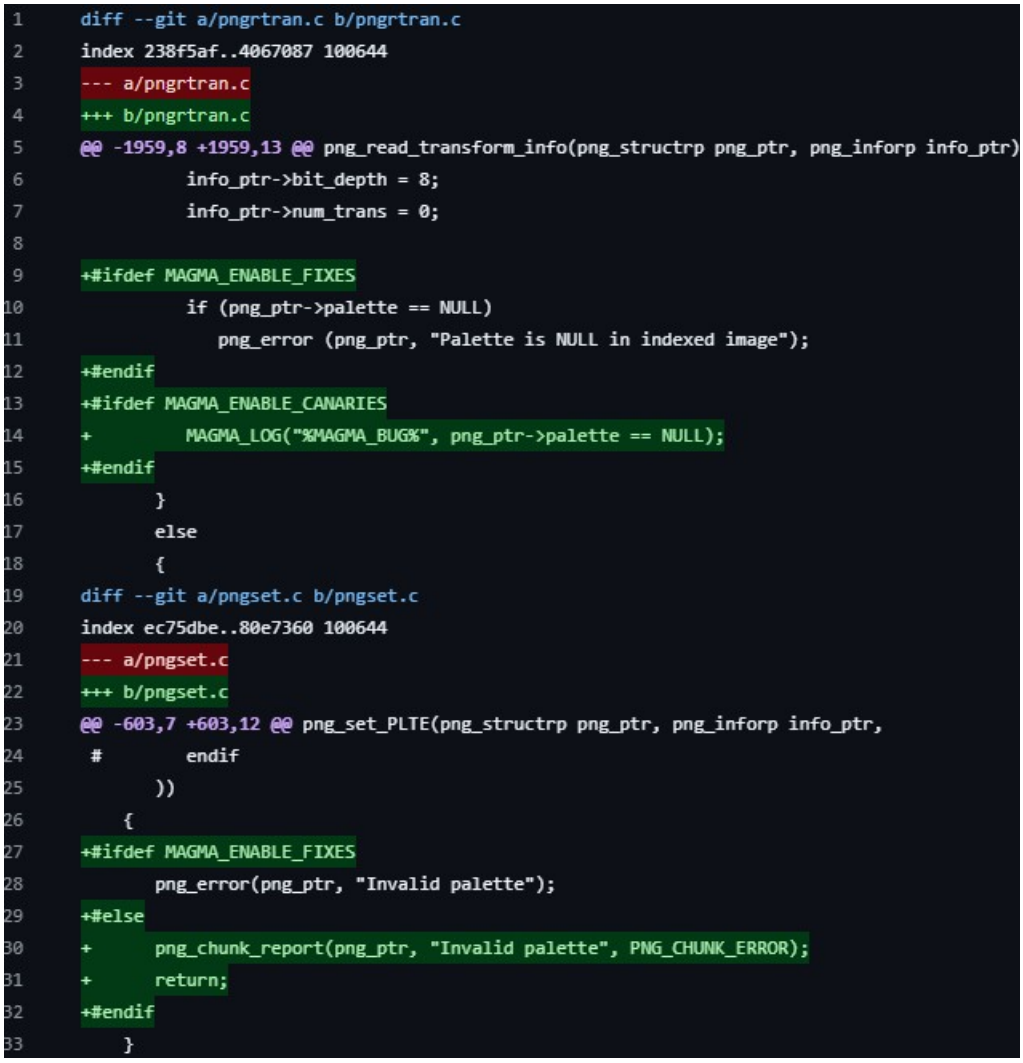
5.1 Punti di forza del fuzzing

Il caso di studio conferma diversi punti di forza del fuzzing, in particolare nel contesto di librerie che implementano parser di formati strutturati. AFL è stato in grado di generare input sintatticamente plausibili ma semanticamente inconsistenti, superando i controlli preliminari e raggiungendo porzioni di codice profonde legate alla logica interna di decodifica. Questo aspetto è particolarmente rilevante, poiché molti difetti di sicurezza non emergono con input completamente casuali, ma richiedono combinazioni specifiche di metadati e contenuti che rispettino parzialmente il formato atteso. Un ulteriore vantaggio del fuzzing coverage-guided consiste nella capacità di guidare l'evoluzione degli input tramite feedback di esecuzione. L'espansione progressiva del corpus osservata durante la campagna sperimentale evidenzia come il fuzzer non si limiti a esplorare variazioni superficiali, ma costruisca nel tem-

po input sempre più efficaci, massimizzando la probabilità di attivare vulnerabilità annidate in percorsi di esecuzione profondi, altrimenti difficili da stimolare con test convenzionali.

5.2 Correzione del crash

Il crash analizzato è riconducibile alla vulnerabilità documentata nel benchmark MAGMA e associata alla `CVE-2013-6954`[13], che riguarda la gestione non corretta della palette nei file PNG in presenza di metadati semanticamente inconsistenti. In particolare, la vulnerabilità si manifesta quando il programma tenta di accedere alla struttura dati della palette durante le fasi di trasformazione dell'immagine, senza aver verificato che tale struttura sia stata correttamente allocata e inizializzata in fase di parsing. La correzione del bug, così come implementata dagli autori del benchmark MAGMA, consiste nell'introduzione di controlli aggiuntivi sulla coerenza del chunk `PLTE` e sullo stato interno delle strutture dati associate. In presenza di una palette assente o non valida, l'elaborazione dell'input viene interrotta in modo controllato, evitando che il flusso di esecuzione raggiunga istruzioni critiche che comporterebbero l'accesso a memoria non valida. Questo intervento è coerente con il principio di progettazione *fail fast*: qualora le precondizioni logiche richieste per una determinata trasformazione non siano soddisfatte, il programma segnala immediatamente l'errore e termina l'elaborazione dell'input, prevenendo comportamenti indefiniti o crash. Nel caso specifico, il controllo esplicito sulla presenza della palette impedisce la dereferenziazione di un puntatore nullo all'interno della funzione `png_do_expand_palette`, eliminando la causa diretta del segmentation fault osservato durante l'analisi.



```
diff --git a/pngrtran.c b/pngrtran.c
index 238f5af..4067087 100644
--- a/pngrtran.c
+++ b/pngrtran.c
@@ -1959,8 +1959,13 @@ png_read_transform_info(png_structrp png_ptr, png_inforp info_ptr)
          info_ptr->bit_depth = 8;
          info_ptr->num_trans = 0;

+#ifdef MAGMA_ENABLE_FIXES
          if (png_ptr->palette == NULL)
             png_error (png_ptr, "Palette is NULL in indexed image");
+#endif
+#ifdef MAGMA_ENABLE_CANARIES
+         MAGMA_LOG("%MAGMA_BUG%", png_ptr->palette == NULL);
+#endif
       }
       else
       {
diff --git a/pngset.c b/pngset.c
index ec75dbe..80e7360 100644
--- a/pngset.c
+++ b/pngset.c
@@ -603,7 +603,12 @@ png_set_PLTE(png_structrp png_ptr, png_inforp info_ptr,
 #        endif
       ))
    {
+#ifdef MAGMA_ENABLE_FIXES
       png_error(png_ptr, "Invalid palette");
+#else
+      png_chunk_report(png_ptr, "Invalid palette", PNG_CHUNK_ERROR);
+      return;
+#endif
    }
```

Figura 5.1: Patch di sicurezza per la `CVE-2013-6954` nel repository MAGMA.

La Figura 5.1 mostra un estratto del codice corretto fornito dal benchmark MAGMA, nel quale è possibile osservare l'introduzione di verifiche esplicite sullo stato della palette prima del suo utilizzo. Il confronto tra il comportamento del codice vulnerabile e quello corretto conferma la validità della RCA condotta: il crash non è dovuto a un'anomalia del fuzzer o dell'ambiente di esecuzione, ma a una mancanza di validazione semantica dell'input all'interno della libreria. Nel complesso, questa correzione evidenzia come il fuzzing, supportato da benchmark basati su bug reali come MAGMA, non si limiti a individuare crash superficiali, ma consenta di ricondurre tali eventi a vulnerabilità concrete e documentate, facilitando l'analisi delle cause e la definizione di soluzioni robuste. Il collegamento diretto tra input

malformato, crash, CVE e patch dimostra l'efficacia del fuzzing come strumento di supporto allo sviluppo di software più sicuro e resiliente.

Capitolo 6

Conclusioni

In questa tesi è stata condotta un'analisi sperimentale del fuzzing applicato a software reale, utilizzando il fuzzer AFL all'interno del framework MAGMA. L'obiettivo principale del lavoro era valutare l'efficacia del fuzzing nel generare input malformati ma sintatticamente validi, capaci di attivare vulnerabilità reali all'interno di un programma ampiamente utilizzato come *libpng*. I risultati ottenuti mostrano come AFL sia in grado di esplorare in modo progressivo ed efficace lo spazio degli input, partendo da un corpus iniziale estremamente ridotto e ampliandolo fino a generare un insieme di oltre un migliaio di file PNG differenti. L'evoluzione del corpus e l'elevato numero di test eseguiti evidenziano la capacità del fuzzer di adattare le proprie mutazioni sulla base del feedback di copertura, raggiungendo porzioni di codice non inizialmente previste dagli sviluppatori. La campagna di fuzzing ha portato all'individuazione di diversi crash, tra i quali è stato selezionato un caso rappresentativo per un'analisi approfondita. La Root Cause Analysis ha permesso di ricondurre il crash a una vulnerabilità reale documentata, associata alla `CVE-2013-6954`, confermando la validità del benchmark MAGMA come strumento di valutazione basato su bug reali. L'analisi ha evidenziato come input semanticamente inconsistenti, pur rispettando la struttura sintattica del formato PNG, possano violare assunzioni implicite del codice e causare accessi non validi alla memoria. Questo risultato sottolinea uno degli aspetti più rilevanti del fuzzing: la capacità di mettere in luce debolezze logiche e carenze di validazione che difficilmente emergerebbero tramite test manuali o casi di test tradizionali. In particolare, il caso analizzato mostra come la mancanza di

controlli semantici su strutture dati critiche possa condurre a vulnerabilità anche in librerie mature e ampiamente utilizzate. Il lavoro svolto presenta tuttavia alcuni limiti. L'analisi è stata condotta su un singolo fuzzer e un unico programma target, l'approfondimento tramite RCA è stato applicato a un solo crash rappresentativo. Estensioni future potrebbero includere il confronto tra diversi fuzzer, l'analisi automatizzata di un numero maggiore di crash e l'applicazione della metodologia a ulteriori formati di file o librerie. In conclusione, l'esperienza maturata dimostra come il fuzzing rappresenti una tecnica efficace e pratica per l'analisi della sicurezza del software, anche in contesti reali. L'integrazione di strumenti come AFL e MAGMA consente non solo di individuare vulnerabilità concrete, ma anche di comprendere più a fondo le cause degli errori e di promuovere pratiche di sviluppo più robuste e orientate alla sicurezza.

Bibliografia

[1] Ahmad Hazimeh, Adrian Herrera e Mathias Payer. *MAGMA: A Ground-Truth Fuzzing Benchmark*. 2020. URL: https://hexhive.epfl.ch/magma/.

[2] Greg Roelofs and libpng contributing authors. *Official libpng Homepage*. 2025. URL: https://www.libpng.org/pub/png/libpng.html.

[3] Wikipedia. *Fuzzing*. 2025. URL: https://it.wikipedia.org/wiki/Fuzzing.

[4] AFL Developers. *Motivation for Fuzz Testing*. 2019. URL: https://afl-1.readthedocs.io/en/latest/motivation.html.

[5] Andrea Fioraldi e Dominik Maier. *The LibAFL Fuzzing Library*. 2026. URL: https://aflplus.plus/libafl-book/libafl.html.

[6] W3C. *PNG Structure Specification*. 2009. URL: https://www.libpng.org/pub/png/spec/1.2/PNG-Structure.html.

[7] W3C. *PNG Specification – IDAT Chunk*. 2025. URL: https://www.w3.org/TR/png/#11IDAT.

[8] Libpng Development Team. *libpng Manual*. 2025. URL: https://libpng.org/pub/png/libpng-manual.txt.

[9] Wikipedia. *Distribuzione Linux*. 2025. URL: https://it.wikipedia.org/wiki/Distribuzione_Linux.

[10] Ahmad Hazimeh, Adrian Herrera e Mathias Payer. *Magma: Documentation*. 2020. URL: https://hexhive.epfl.ch/publications/files/21SIGMETRICS.pdf.

[11] Docker. *Docker*. 2026. URL: https://www.docker.com/.

[12] IBM. *GDB*. 2026. URL: https://www.ibm.com/docs/it/sdk-java-technology/8?topic=techniques-debugging-gdb.

[13] The MITRE Corporation. *Common Vulnerabilities and Exposures*. 2026. URL: https://www.cve.org/CVERecord?id=CVE-2013-6954.