



SAPIENZA
UNIVERSITÀ DI ROMA

Ricerca di vulnerabilità con fuzz testing

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Corso di Laurea in Ingegneria Informatica e Automatica

Claudia Giuseppini

Matricola 2069387

Relatore

Prof. Daniele Cono D'Elia

Correlatore

Ing. Matteo Marini

Anno Accademico 2024/2025

Tesi non ancora discussa

Ricerca di vulnerabilità con fuzz testing

Sapienza Università di Roma

© 2025 Claudia Giuseppini. Tutti i diritti riservati

Questa tesi è stata composta con \LaTeX e la classe Sapthesis.

Email dell'autore: giuseppini.2069387@studenti.uniroma1.it

Sommario

Indice

1	Introduzione	1
1.1	Obiettivi	2
2	Background	3
2.1	Bug e Vulnerabilità	3
2.2	Sanitizers	4
2.2.1	MSan	4
2.2.2	QMSan	6
2.2.3	Valgrind	6
2.3	Fuzzing	7
2.3.1	AFL++	10
2.3.2	Fuzzer Test Suite	11
2.3.3	OSS-Fuzz	11
3	Metodologia	13
3.1	Configurazione dell'Ambiente	13
3.2	Descrizione delle Attività	14
3.2.1	Costruzione dell'immagine docker	14
3.2.2	Creazione e avvio del Docker	15
3.2.3	Avvio di una sessione di Fuzzing con AFL++	15
3.2.4	Riproduzione manuale di un crash	16
3.2.5	Analisi con Valgrind	16
3.3	Deduplication e Categorizzazione	16
4	Risultati	19
4.1	Panoramica generale	19
4.1.1	OpenSC	19
4.1.2	libucl	19
4.1.3	GPAC	20
4.2	Case Study: GPAC	20
5	Conclusione	21
	Bibliografia	23

Capitolo 1

Introduzione

In seguito alle rivoluzioni industriali dello scorso secolo la tecnologia è diventata sempre più parte della nostra società. Sempre più sistemi, dapprima analogici, divengono digitali; molte professioni che hanno rappresentato un pilastro per la società si trovano oggi a essere automatizzate o addirittura obsolete, sostituite da macchine e algoritmi sempre più sofisticati. Risulta impossibile non incontrare nella nostra quotidianità strumenti digitali di cui non possiamo fare a meno, e li consideriamo infallibili. In aggiunta, senza una reale consapevolezza dei rischi connessi, condividiamo dati personali e sensibili su Internet.

La progressiva centralizzazione delle informazioni lascia prevedere scenari in cui la quasi totalità dei dati umani sarà gestita all'interno di ampie infrastrutture di calcolo e archiviazione.

Come risultato di questo cambiamento tecnologico, la società si sta adattando e una componente chiave di questo processo, in questo caso, è lo sviluppo di una forte cultura della sicurezza delle informazioni. Infatti, esiste un inevitabile rischio di furto, frode o perdita quando i dati vengono raccolti e gestiti. Quello che prima accadeva materialmente con documenti fisici o segreti aziendali circoscritti, oggi accade con identità digitali, transazioni finanziarie e sistemi di infrastrutture critiche come i sistemi energetici e sanitari.

Nel presente scenario, la sicurezza informatica non rappresenta più un tema riservato agli esperti ma anche una componente cruciale per garantire la stabilità della società. A causa di questi fattori, il campo della sicurezza informatica si sta espandendo a un ritmo esponenziale, adottando nuove tecniche e metodologie, e affrontando minacce impensabili solo pochi anni fa, come la protezione dell'intelligenza artificiale, del cloud computing e dei sistemi Internet of Things (IoT).

L'obiettivo principale della sicurezza del software è permettere l'utilizzo previsto delle funzionalità applicative, impedendo al contempo qualsiasi uso non intenzionale o non autorizzato [18]. Un impiego improprio, infatti, può comportare conseguenze dannose, come l'uso involontario o abusivo delle risorse di calcolo al di fuori dei limiti consentiti e definiti dal progetto.

Lo scopo di questo tirocinio è contribuire, anche se in piccola parte, a questo sforzo collettivo: sfruttare le tecniche di sicurezza per identificare le vulnerabilità in alcuni programmi open source utilizzati quotidianamente da milioni di persone. Una volta identificate, le vulnerabilità sono state tempestivamente comunicate ai

responsabili di progetto, i quali hanno cominciato il processo di correzione. In questo modo, non solo è possibile ridurre il rischio per gli utenti finali, ma anche valorizzare l'importanza della collaborazione all'interno della comunità informatica, dove la trasparenza svolge un ruolo cruciale nel rendere il cyberspazio un luogo più sicuro per tutti.

1.1 Obiettivi

L'obiettivo principale di questo tirocinio è l'individuazione di errori software che non vengono rilevati dai tradizionali strumenti di analisi, con particolare attenzione alle vulnerabilità di tipo Use-of-Uninitialized Memory (UUM), come verrà approfondito nei capitoli successivi. Tali errori rappresentano una categoria critica di vulnerabilità, poiché non sempre determinano comportamenti evidenti come un crash immediato, ma possono compromettere in modo silente lo stato interno del programma e, in scenari più gravi, essere sfruttati da un attaccante per ottenere accesso a informazioni sensibili o accedere ad un sistema senza i necessari privilegi.

Il lavoro svolto ha dunque un duplice scopo: da un lato, incrementare la capacità di rilevamento di bug rispetto agli strumenti tradizionalmente utilizzati, dall'altro fornire un contributo concreto alla sicurezza delle piattaforme analizzate. Una volta identificate le vulnerabilità, infatti, esse vengono documentate e segnalate ai rispettivi gestori o sviluppatori, così da permettere l'avvio di un processo di patching e di mitigazione del rischio.

Capitolo 2

Background

Questo capitolo mira a fornire tutte le informazioni necessarie per comprendere il lavoro svolto e le tecnologie utilizzate. Dapprima vedremo cosa significa bug o errore e perché è rilevante risolverli. Vedremo poi quali sono i principali strumenti utilizzati per rilevare bug, come Sanitizers e Fuzzing e i benefici della loro implementazione simultanea.

2.1 Bug e Vulnerabilità

Un bug è un difetto in un software che porta ad un risultato inaspettato. Errori nei software sono dovuti ad errori umani nel codice, compilatore o sistema runtime [13]. Questi portano il programma in uno stato non voluto e ne causano possibilmente il blocco. Solo nel caso in cui il bug potrebbe permettere ad un attaccante la compromissione di Confidenzialità, Integrità o Disponibilità si parla allora di vulnerabilità [18].

Una vulnerabilità è quindi una debolezza del sistema che permette ad un attaccante di sfruttare un bug. Richiede 3 componenti principali:

1. un sistema suscettibile al difetto;
2. un avversario con accesso al difetto;
3. un avversario che abbia le capacità di sfruttare il difetto [18].

La sicurezza di un sistema software è disciplinata da un insieme di principi fondamentali. Quando questi principi sono rispettati, il software può essere considerato sicuro. Tra le principali dimensioni della sicurezza informatica si distinguono la Memory Safety e la Type Safety.

La Memory Safety garantisce che ogni puntatore faccia riferimento esclusivamente a celle di memoria valide e correttamente allocate [12]. La Type Safety, invece, assicura che gli oggetti vengano utilizzati in conformità al loro tipo, rispettando la gerarchia dei tipi e le conversioni (cast) corrette durante l'esecuzione del programma [17]. La presenza di bug nel codice può compromettere la memory o type safety, esponendo il programma a vulnerabilità sfruttabili per modificare il suo stato di runtime. In tali casi, un attaccante potrebbe eseguire operazioni malevole. Quando un programma non segue più controlli definiti e flussi di dati corretti, si può parlare

di una weird machine [8]. Il rispetto della memory e type safety consente di prevenire difetti a basso livello, limitando l'esposizione del software a vulnerabilità logiche specifiche. Va inoltre sottolineato che i requisiti di sicurezza non sono direttamente testabili. Il testing consente di individuare la presenza di errori, ma non la loro completa assenza. L'obiettivo del testing, in questo contesto, è verificare se durante l'esecuzione vengano rispettate le proprietà di memory e type safety [22].

Nonostante ciò, il Testing resta una componente essenziale nel ciclo di vita di un software, ad ogni aggiornamento o ulteriore implementazione di funzionalità deve seguire un rigido regime di testing. Il testing si divide principalmente in due categorie: Statico e Dinamico. Il testing Statico si riferisce a tutte quelle tecniche capaci di rilevare errori senza eseguire il codice. Tra questi troviamo symbolic execution. Il testing dinamico invece esegue il programma e rileva errori osservando il comportamento durante l'esecuzione. Nel lavoro di questa tesi sono stati utilizzati strumenti di testing dinamico, sanitizers e fuzzers. L'idea alla base è combinare le molteplici esecuzioni del fuzzing che permette di esplorare più percorsi possibili ai controlli aggiuntivi imposti dai sanitizer che validare queste esecuzioni.

2.2 Sanitizers

Esistono diversi tipi di tecniche di testing: le più comuni si limitano a rilevare bug tramite asserzioni, segmentation faults o eccezioni che causano la terminazione del programma. Tuttavia, queste tecniche non sempre riescono a individuare errori più sottili, che non portano immediatamente a un crash ma possono comunque compromettere la correttezza della computazione [18].

Per affrontare questi casi, sono stati sviluppati strumenti noti come Sanitizers, i quali inseriscono, già in fase di compilazione, un codice di strumentazione che controlla e valida il comportamento del programma durante l'esecuzione. Ogni sanitizer è specializzato nell'individuare una specifica classe di bug (o poche classi affini), come ad esempio buffer overflows, utilizzo di valori non inizializzati o differenziazione di indirizzi non validi. In pochi anni, questi strumenti sono diventati essenziali nell'identificare vulnerabilità [21]. Il primo strumento di questo tipo fu l'AddressSanitizer (ASan), introdotto da Google nel 2012, progettato per rilevare errori di memoria come use-after-free e accessi fuori dai limiti [20], [11].

2.2.1 MSan

In questo progetto di tesi ci siamo concentrati sull'individuazione dei bug noti come Use of Uninitialized Memory (UUM), ossia l'utilizzo di memoria non inizializzata durante l'esecuzione di un programma [23]. Il sanitizer più diffuso per questa categoria di errori è il MemorySanitizer (MSan), sviluppato da Google nel 2015 [15].

Questo è un detector di memory read per C/C++ che aiuta gli sviluppatori a trovare e aggiustare errori di tipo UUM, considerati solitamente difficili da trovare in quanto non occorrono in ogni esecuzione e possono essere triggerati da una qualunque operazione del programma. I linguaggi C/C++ in particolare lasciano allo sviluppatore il compito di allocare, utilizzare e liberare correttamente ogni cella di memoria acceduta dal programma - sono definite per questo motivo memory-unsafe [17]. Questo tipo di vulnerabilità non possono essere sfruttate solo per alterare il

```

#3 0x405e51 in std::_Vector_base<int, std::allocator<int> >::_M_allocate(unsig
ned long) /opt/sde/packages/gcc-9.3.0/include/c++/9.3.0/bits/stl_vector.h:343
#4 0x405d00 in void std::vector<int, std::allocator<int> >::_M_range_initiali
ze<int const*>(int const*, int const*, std::forward_iterator_tag) /opt/sde/packag
es/gcc-9.3.0/include/c++/9.3.0/bits/stl_vector.h:1579
#5 0x404a00 in std::vector<int, std::allocator<int> >::vector(std::initializ
er_list<int>, std::allocator<int> const&) /opt/sde/packages/gcc-9.3.0/include/c++/
9.3.0/bits/stl_vector.h:626
#6 0x404491 in main /home/jdoe/demo/asan/cppbook_companion/miscellany/buggy/a
pp.cpp:7
#7 0x7f7d7599b1a2 in __libc_start_main (/lib64/libc.so.6+0x271a2)

SUMMARY: AddressSanitizer: heap-use-after-free /home/jdoe/demo/asan/cppbook_compa
nion/miscellany/buggy/app.cpp:12 in main
Shadow bytes around the buggy address:
 0x0c047fff7fb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x0c047fff7fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x0c047fff7fd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x0c047fff7fe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x0c047fff7ff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x0c047fff8000: fa fa fd fd fa fa fa fa fa fa fa fa fa fa fa
 0x0c047fff8010: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
 0x0c047fff8020: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
 0x0c047fff8030: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa

```

Figura 2.1. Esempio di utilizzo di ASan

flusso di esecuzione del programma ma anche per rilevare informazioni sullo stato interno del programma e contenuti dello stack/heap [25].

Un bug di tipo UUM si origina solitamente quando viene utilizzata una variabile o un buffer prima della sua inizializzazione. Questo risulta essere pericoloso poiché le variabili locali non inizializzate contengono valori casuali (spesso derivati da utilizzi precedenti della memoria) e l'utilizzo di queste variabili porta ad un comportamento indefinito del programma [?].

MSan adotta un approccio compiler-based, in quanto inserisce nel programma, in fase di compilazione, istruzioni aggiuntive per tracciare lo stato di inizializzazione di ogni cella di memoria. Il suo funzionamento si fonda sul concetto di shadow memory, una memoria parallela che mantiene le informazioni sullo stato della memoria principale [15]. Ogni operazione di scrittura aggiorna la shadow memory, mentre ogni operazione di lettura è accompagnata da un controllo che permette di intercettare l'uso di valori non inizializzati [13].

Un aspetto complesso di questo approccio è che non tutti i pattern rilevati come potenzialmente pericolosi corrispondono a veri bug. Alcune istruzioni apparentemente in conflitto con i principi di memory safety sono in realtà introdotte dal compilatore per motivi di ottimizzazione [?]. Per ridurre i falsi positivi, MSan concentra quindi le verifiche su eventi semanticamente rilevanti per l'esecuzione, come:

- la valutazione di una branch condition;
- la dereferenziazione di un puntatore;
- il passaggio di dati a una system call o a una chiamata di libreria sensibile.

Per garantire la correttezza del tracciamento, i sanitizer per UUM devono modellare il flusso dei dati attraverso le diverse istruzioni, operazione nota come shadow propagation. Questo riduce i falsi positivi (ad esempio nei casi di copia di memoria non inizializzata), ma introduce un overhead significativo a runtime. Inoltre, uno dei principali limiti di MSan consiste nella necessità di ricompilare tutte le librerie con la strumentazione: in assenza di tale ricompilazione, il numero di falsi positivi cresce sensibilmente [16].

2.2.2 QMSan

Questo lavoro di tesi ha utilizzato lo strumento QMSan, un sanitizer che riprende i concetti fondamentali di MSan e supera alcune limitazioni strutturali. QMSan utilizza QEMU User Emulation, a differenza di MSan, quindi non richiede la ricompilazione completa delle librerie né il codice sorgente. Questo lo rende anche applicabile immediatamente a software binari di terze parti o a librerie che non hanno il sorgente [16].

Mentre MSan monitora lo stato di inizializzazione solo per gli oggetti che osservano l'allocazione o la deallocazione, QMSan sfrutta una tecnica più semplice ed efficace: considera la memoria come non inizializzata di default; lo stato viene aggiornato solo dopo le operazioni di scrittura.

Questo consente di mantenere un tracciamento coerente a livello binario, riducendo le ambiguità tipiche dell'approccio compiler-based. Pertanto, è possibile mantenere un tracciamento coerente a livello binario, riducendo le ambiguità tipiche dell'approccio basato sul compilatore.

Il suo design multi-layer opportunistico è un'innovazione significativa di QMSan. Inizialmente, il sistema utilizza un detector leggero che cerca anomalie nelle operazioni di load and store. Il programma viene rieseguito utilizzando un detector accurato basato su shadow propagation completa quando viene scoperto un bug sospetto. Se l'anomalia si dimostra un falso positivo, viene registrata e ignorata nelle esecuzioni successive per evitare di ripetere controlli ridondanti.

Questa architettura consente a QMSan di mantenere un buon compromesso tra accuratezza e performance, con un overhead significativamente inferiore rispetto a Memcheck e altri strumenti binari. La sua compatibilità universale lo rende utilizzabile con qualsiasi binario, indipendentemente dalla ricompilazione o dal supporto del compilatore. In questo senso, QMSan è una soluzione più pratica e flessibile rispetto a MSan, specialmente quando si tratta di fuzzing di software di terze parti o proprietario.

2.2.3 Valgrind

Valgrind è un framework di analisi dinamica popolare per il debugging di applicazioni scritte in linguaggi a basso livello come C++ e C. La capacità di rilevare errori di memoria come accessi fuori dai limiti, utilizzo di memoria non inizializzata e perdite di memoria sono tra le sue caratteristiche principali [26], [27]. Tuttavia, la capacità di creare uno stack trace preciso e affidabile in caso di crash è tra le caratteristiche più importanti del debugging avanzato. Quando un programma si arresta in modo anomalo, identificare esattamente dove si è verificato l'errore è fondamentale per comprendere la causa del problema e creare una correzione efficace.

Valgrind monitora l'esecuzione del programma in un ambiente virtualizzato e traccia ogni operazione di memoria. Questa modalità consente allo strumento di mantenere informazioni accurate sul contesto di ogni istruzione eseguita, inclusi i registri della CPU e i puntatori di stack. Ciò evita le ambiguità che si verificano spesso durante un crash normale del sistema. Ciò significa che Valgrind può fornire un backtrace accurato e coerente anche in presenza di buffer overflow o corruzioni di

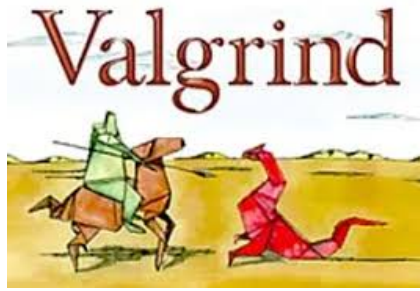


Figura 2.2. Logo di Valgrind

memoria, mostrando i parametri passati e gli indirizzi di ritorno della sequenza di chiamate di funzione che ha causato il crash.

In contesti di testing automatizzato e di fuzzing, dove i crash possono verificarsi in modo imprevisto e su percorsi di codice poco esplorati, Valgrind è uno strumento molto utile. Questo perché ha la capacità di creare uno stack affidabile. In particolare, per quanto riguarda i valori non definiti [4]. Lo stack trace creato da Valgrind non solo consente di identificare rapidamente il punto di origine dell'errore, ma fornisce anche informazioni utili per ottimizzare le strategie di fuzzing o per creare nuovi sanitizer nei workflow sperimentali. In particolare, Valgrind garantisce una tracciabilità completa, rendendo possibile la riproduzione del bug e la verifica della sua natura in modo affidabile. Ciò lo distingue dai crash comuni, in cui lo stack può essere corrotto o parzialmente leggibile.

2.3 Fuzzing

Il fuzzing è una delle tecniche di testing automatico più diffuse grazie alla capacità di rilevare problemi critici. Questa tecnica consiste nel sollecitare sistematicamente un programma con una serie di input casuali o inaspettati per verificare che non avvengano crash, eccezioni o terminazioni anomale [28], [19]. Questo strumento è particolarmente efficace, quando comparato a tool come angr (utilizzato per la symbolic execution), AFL (fuzzing tool che vedremo successivamente) è riuscito a trovare più bug in un periodo di 24 ore [24].

Gli input nel fuzzing sono generati automaticamente e spesso includono elementi di randomicità; particolare attenzione è posta ai casi limite (edge cases). Un fuzzer è lo strumento che implementa il processo di fuzzing: esegue un ciclo automatizzato guidato da feedback e mantiene una coda (corpus) di test case. Il funzionamento tipico di un fuzzer può essere descritto come segue:

1. si inizializza la coda con un insieme di input detto corpus;
2. si generano nuovi input mutando quelli presenti nella coda o generandone di nuovi;
3. si esegue il programma con gli input prodotti;

4. si osservano informazioni di esecuzione (es. crash, code coverage, asserzioni) e si raccolgono feedbacks mantengono nella coda gli input ritenuti interessanti per successive mutazioni e analisi [18].

Core fuzzing algorithm:

```

corpus ← initSeedCorpus()
queue ← ∅
observations ← ∅
while ¬isDone(observations,queue) do
  candidate ← choose(queue, observations)
  mutated ← mutate(candidate, observations)
  observation ← eval(mutated)
  if isInteresting(observation, observations) then
    queue ← queue ∪ mutated
    observations ← observations ∪ observation
  end if
end while

```

parameterized by functions:

- **initSeedCorpus**: Initialize a new seed corpus.
- **isDone**: Determine if the fuzzing should stop or not based on progress toward a goal, or a timeout.
- **choose**: Choose at least one candidate seed from the queue for mutation.
- **mutate**: From at least one seed and any observations made about the program so far, produce a new candidate seed.
- **eval**: Evaluate a seed on the program to produce an observation.
- **isInteresting**: Determine if the observations produced from an evaluation on a mutated seed indicate that the input should be preserved or not.

Figura 2.3. Fuzzing come schema algoritmico [14]

In una prima fase, il fuzzer organizza la coda con i seeds: un insieme di input validi o semi-validi. Questi permettono al fuzzer di esplorare percorsi realistici, superando i controlli basilari di validazione e penetrando più a fondo nella logica del programma. Anche un singolo seed efficace può garantire una copertura significativa. In assenza di seeds, il fuzzer tenderà a generare molti input che causeranno la terminazione immediata del programma, fino a quando non sarà in grado di apprendere come superare autonomamente i controlli iniziali. Spesso vengono utilizzati test cases provenienti da fonti esterne, forniti come input al programma per osservare il suo comportamento.

Il mutatore rappresenta una delle componenti principali del fuzzer: esso genera nuovi input a partire da uno o più elementi presenti nella coda. In questo modo è possibile ottenere input non solo nuovi, ma anche diversificati, aumentando l'efficacia del corpus che potrà essere sfruttato in sessioni future. L'obiettivo di un buon fuzzer è produrre input sufficientemente validi da essere accettati, ma al tempo stesso abbastanza anomali da poter innescare condizioni limite.

In letteratura si distinguono due principali approcci: mutation-based e generation-based.

I mutation-based fuzzers richiedono un set iniziale di seeds; a partire da questi applicano un insieme di trasformazioni come bit flip, inserimento o rimozione di byte, operazioni aritmetiche e combinazioni tra input differenti. Tali strumenti integrano strategie sia deterministiche che non deterministiche per selezionare e schedulare le mutazioni a ogni tentativo di generazione.

I generation-based fuzzers, invece, non necessitano di seeds iniziali poiché sono in grado di creare input ex novo basandosi su meccanismi di randomicità. Tuttavia, questo approccio richiede generalmente più tempo per generare input di qualità e per ottenere risultati comparabili a quelli basati su mutazioni.

Un ulteriore aspetto cruciale riguarda le performance: l'obiettivo è ottimizzare il numero di esecuzioni ripetute e monitorare come l'input raggiunga il codice da testare. Spesso, per facilitare questa attività, gli sviluppatori implementano un harness, ossia un frammento di codice progettato per leggere i dati forniti dal fuzzer e invocare le porzioni di codice da testare.

Infine, un meccanismo di feedback accurato è essenziale per garantire l'efficacia di un fuzzer. Per ciascun test case eseguito, il fuzzer raccoglie un profilo di esecuzione e lo confronta con le esecuzioni precedenti, in modo da identificare input che conducano a percorsi nuovi o interessanti del programma. I fuzzers possono essere accompagnati da diverse tecniche di analisi finalizzate ad aumentarne l'efficienza e l'efficacia. Si distinguono principalmente tre approcci:

- **Black-box fuzzing**: in questo caso il fuzzer non ha alcuna conoscenza della struttura interna del programma. È particolarmente efficace nell'individuare i cosiddetti bug di superficie, ossia errori che non richiedono un'esplorazione profonda del codice per essere rilevati. Questo approccio si caratterizza per semplicità, velocità e scalabilità, ma non garantisce necessariamente la copertura di percorsi complessi del programma;
- **White-box fuzzing**: al contrario, il fuzzer dispone di informazioni dettagliate sulla struttura interna del programma e genera input accuratamente progettati per esplorare il maggior numero possibile di percorsi. Questo metodo consente di individuare vulnerabilità collocate in profondità, ma comporta costi elevati in termini di tempo e risorse computazionali a causa delle analisi strutturali richieste. Inoltre, non sempre riesce a rilevare errori che un approccio più esplorativo avrebbe potuto scoprire;
- **Grey-box fuzzing**: in questo modello il fuzzer dispone di alcune informazioni parziali sul programma, senza tuttavia conoscerne completamente la struttura interna. Tale compromesso consente di guidare l'esplorazione verso percorsi rilevanti, mantenendo al tempo stesso un buon livello di indipendenza e flessibilità nell'esecuzione [7].

I grey-box fuzzers sono anche noti come coverage-guided fuzzers, in quanto l'esplorazione del programma in fase di test viene regolata da meccanismi di coverage feedback. Durante l'esecuzione, infatti, viene mantenuta una coverage map: una rappresentazione che registra le porzioni di codice raggiunte dagli input. Ogni volta che viene eseguita una nuova sezione del programma, la mappa viene aggiornata di conseguenza [10].

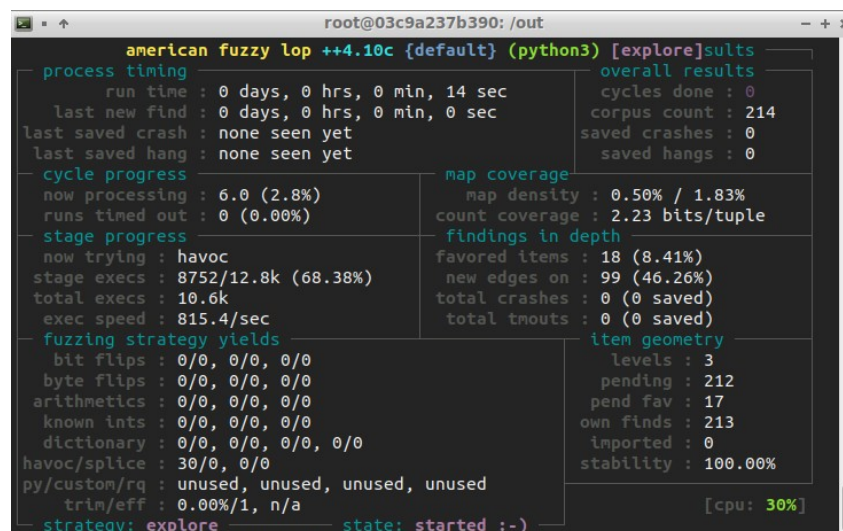
Questo meccanismo permette di identificare rapidamente quali input conducono a percorsi di esecuzione nuovi o inesplorati. Gli input ritenuti “interessanti” vengono conservati nella coda del fuzzer per ulteriori mutazioni, in modo da ampliare progressivamente la copertura del programma e aumentare le probabilità di individuare bug profondi o edge cases.

2.3.1 AFL++

Il fuzzer più popolare, e quello che è stato utilizzato nello svolgimento di questo tirocinio è American Fuzzy Loop ++ (AFL++), uno sviluppo dell’omonimo AFL [9], [6]. Questo è un fuzzer di tipo coverage-guided.

La metrica principale per la performance di un fuzzer è data dai crashes, ma altre caratteristiche da prendere in considerazione sono:

- throughput (exec/sec): test cases completati nell’unità di tempo;
- queue size (“total paths”): test cases distinti generati;
- coverage: numero di edges coperti dal programma.



```

root@03c9a237b390: /out
american fuzzy lop ++4.10c {default} (python3) [explore]sults
process timing
  run time : 0 days, 0 hrs, 0 min, 14 sec
  last new find : 0 days, 0 hrs, 0 min, 0 sec
  last saved crash : none seen yet
  last saved hang : none seen yet
cycle progress
  now processing : 6.0 (2.8%)
  runs timed out : 0 (0.00%)
stage progress
  now trying : havoc
  stage execs : 8752/12.8k (68.38%)
  total execs : 10.6k
  exec speed : 815.4/sec
fuzzing strategy yields
  bit flips : 0/0, 0/0, 0/0
  byte flips : 0/0, 0/0, 0/0
  arithmetics : 0/0, 0/0, 0/0
  known ints : 0/0, 0/0, 0/0
  dictionary : 0/0, 0/0, 0/0, 0/0
  havoc/splice : 30/0, 0/0
  py/custom/rq : unused, unused, unused, unused
  trim/eff : 0.00%/1, n/a
strategy: explore
state: started :-)
overall results
  cycles done : 0
  corpus count : 214
  saved crashes : 0
  saved hangs : 0
map coverage
  map density : 0.50% / 1.83%
  count coverage : 2.23 bits/tuple
findings in depth
  favored items : 18 (8.41%)
  new edges on : 99 (46.26%)
  total crashes : 0 (0 saved)
  total tnouts : 0 (0 saved)
item geometry
  levels : 3
  pending : 212
  pend fav : 17
  own finds : 213
  imported : 0
  stability : 100.00%
[cpu: 30%]

```

Figura 2.4. schermata di AFL++

I primi due sono calcolati da AFL++, il terzo viene derivato processando la coda. Throughput è particolarmente importante, maggiori sono le esecuzioni maggiori sono le probabilità di trovare un bug. Talvolta gli input rigettati possono condizionare questa metrica in quanto sono considerate esecuzioni particolarmente veloci. Bisogna poi tenere conto della lunghezza della coda, infatti, una lunghezza troppo corta può significare che l’esplorazione è troppo superficiale mentre una coda troppo lunga non darà abbastanza tempo al fuzzer per mutare tutti gli input interessanti. Un’altra metrica è la code coverage che ci descrive quanto il programma è stato esplorato e quali parti. Maggiore questa è, più parti del programma abbiamo esplorato.

AFL ci fornisce come output una serie di file con le statistiche relative alla campagna e i relativi log ma anche una serie di cartelle:

- `crashes/`: vengono conservati i test cases che producono crashes per permettere di riprodurli successivamente ed analizzarli;
- `hangs/`: input che si bloccano o restano indefinitivamente bloccati in un loop;
- `queue/`: conserva gli inpt che si sono dimostrati capaci di esplorare grandi o nuove parti de programma.

Per quanto riguarda la coda mantiene dinamicamente un insieme di input di test-cases non ridonanti che permettono di esplorare tutti gli edges (coperti fino a quel momento) della mappa.

Most papers failed to perform multiple runs, and those that did failed to account for varying performance by using a statistical test. This is a problem because our experiments showed that run-to-run performance can vary substantially. • Many papers measured fuzzer performance not by counting distinct bugs, but instead by counting “unique crashes” using heuristics such as AFL’s coverage measure and stack hashes. This is a problem because experiments we carried out showed that the heuristics can dramatically over-count the number of bugs, and indeed may suppress bugs by wrongly grouping crashing inputs. This means that apparent improvements may be modest or illusory. Many papers made some consideration of root causes, but often as a “case study” rather than a performance assessment. • Many papers used short timeouts, without justification. Our experiments showed that longer timeouts may be needed to paint a complete picture of an algorithm’s performance. • Many papers did not carefully consider the impact of seed choices on algorithmic improvements. Our experiments showed that performance can vary substantially [14]

2.3.2 Fuzzer Test Suite

I Fuzzer Test Suite (FTS) sono un insieme di benchmark sviluppate Google, create per testare sistematicamente l’efficacia e le prestazioni dei fuzzer. Questa infatti raccoglie programmi reali che in passato hanno avuto serie vulnerabilità; queste vulnerabilità sono state poi documentate, diventando così un riferimento affidabile per misurare la capacità dei fuzzer di individuare bug. Per questo motivo FTS è ampiamente utilizzato nell’ambito della ricerca accademica e nello sviluppo di nuovi fuzzers e sanitizers, mette infatti a disposizione un ambiente di testing comunque e riproducibile.

Nell’ambito di questo tirocinio, l’FTS è stato impiegato per verificare la validità del mio setup sperimentale.

2.3.3 OSS-Fuzz

Oss-Fuzz è un servizio di fuzzing continuo fornito da Google per progetti di tipo open-source. Il suo lavoro è quello di ottimizzare le esecuzioni di molteplici fuzzer su build strumentate e coordinare la raccolta, deduplicazione e segnalazione dei crashes. Esegue quindi testing su larga scala ed invia report riproducibili e tracciabili ai manutentori dei progetti. Questo approccio permette di scoprire vulnerabilità reali in modo proattivo, migliorando la qualità del software e fornendo un ambiente di testing continuo e standardizzato.

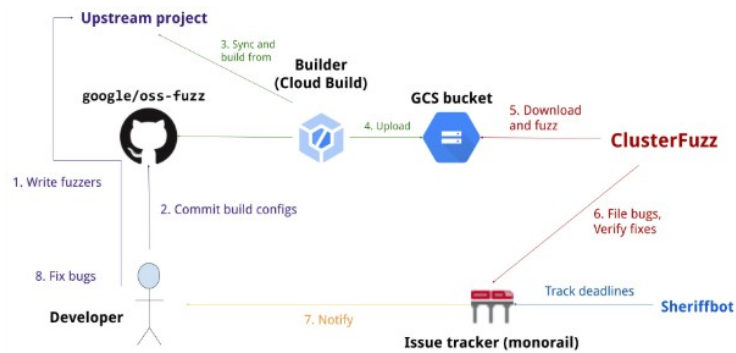


Figura 2.5. Come funziona OSS-Fuzz

Nel contesto di questo tirocinio, alcuni programmi di OSS-Fuzz sono stati testati attraverso una combinazione di fuzzing e l'utilizzo del sanitizer QMSan con lo scopo di individuare UUM bug tralasciati dal sistema automatico di reporting di OSS-Fuzz.

Capitolo 3

Metodologia

Questo capitolo descrive le pratiche utilizzate durante lo svolgimento di questo tirocinio. Nella prima parte, delinea l'ambiente di lavoro e spiego come riprodurlo; Nella seconda sezione sono descritte le campagne di fuzzing che sono state eseguite, insieme agli strumenti e ai parametri necessari; Infine, illustro il processo utilizzato per discriminare e raggruppare i crash trovati, che include i metodi di triaging e deduplicazione.

3.1 Configurazione dell'Ambiente

Tutti i test sono stati eseguiti su macchine virtuali personali con Ubuntu 22.04.4 LTS. L'ambiente sperimentale impiegato comprende, oltre agli strumenti già citati, i seguenti componenti: Docker, Valgrind, Python e GDB. La scelta di utilizzare Docker come ambiente è soprattutto legata alla riproducibilità dei test, possiamo così controllare che non ci siano variazioni tra macchine diverse. L'infrastruttura di test è stata realizzata mediante più Dockerfile distinti, ciascuno predisposto per una parte specifica del workflow di fuzzing e analisi, in particolare in tre sezioni:

- **compiler**: questo Dockerfile è pensato per la preparazione della toolchain e dell'integrazione con AFL++. Partendo da un'immagine ispirata a OSS-Fuzz, installa compilatori, debugger e librerie necessarie. Compila un driver (`aflpp_driver.c`) che integra AFL++ come libreria di fuzzing, imposta i flag di compilazione e le variabili d'ambiente specifiche per il motore di fuzzing e per il linguaggio C/C++, e crea i seed e le directory richieste per l'esecuzione automatica delle campagne. Inoltre, clona e costruisce una versione personalizzata di AFL++ (AFL-QMSan) da repository GitHub usando LLVM 10 per abilitare funzionalità avanzate.
- **qmsan**: questo Dockerfile crea un ambiente dedicato al fuzzing con QMSAN, un sanitizer che estende il runtime con taint tracking. Le operazioni eseguite includono: clonazione e build di QMSAN in diverse configurazioni (con/senza tracciamento di taint, con integrazione AFL), compilazione di un driver di fuzzing statico, predisposizione delle directory di input/output per il fuzzing e configurazione delle variabili d'ambiente e delle opzioni specifiche per QMSAN

(ad es. affinità CPU e flag dedicati). L'immagine è resa compatibile con toolchain basate su LLVM 10 per supportare modalità avanzate.

- programma sotto testing: a partire dall'immagine base `qmsan:latest` vengono installate le dipendenze generali, scaricato l'ultimo commit del programma oggetto di test e configurate le opzioni di compilazione (compile flags). Il progetto viene compilato e nel container viene aggiunta una directory seed contenente input di prova (ad es. `"hi"` o `{}`). Qui vengono anche impostate variabili d'ambiente comuni al workflow (ad es. `DOCKER_NAME=gpac`) e predisposte le directory di input/output per le campagne.

Questa divisione in file Docker distinti separa le responsabilità (configurazione di base, componente sanitizzante/taint e toolchain/driver di fuzzing), rendendo più semplice la riproducibilità, la manutenzione e l'analisi comparativa di configurazioni diverse.

3.2 Descrizione delle Attività

La prima parte del tirocinio è stata dedicata a verificare il corretto funzionamento dell'intero setup sperimentale. Ho usato i programmi della Google Fuzzer Test Suite come casi di prova per assicurarmi che immagini, toolchain e harness fossero correttamente configurati e integrati. Per fare ciò ho utilizzato dei seed "dummy" ovvero validi ma perché molto semplici. La durata di queste campagne era solitamente di 24 ore.

Nella seconda parte ho invece esplorando alcuni programmi di OSS-Fuzz ovvero, `gpac`, `libredwg`, `libc` e `opensc`. Inizialmente, ho eseguito varie campagne di fuzzing partendo da seed "dummy" ma essendo questi programmi più complicati dei precedenti non ho avuto risultati. Ho successivamente iniziato le campagne di fuzzing da un pool di seed mantenuto da oss-fuzz con dei seed già considerati interesting. Questo è stato un passo essenziale nella scoperta dei bug. La durata di queste campagne era solitamente di 36 ore.

Tabella 3.1. Recap dei binari utilizzati relativi ad ogni programma

Programma	Binario testato
<code>gpac</code>	<code>fuzz_probe_analyze</code>
<code>libredwg</code>	<code>llvmfuzz</code>
<code>libucl</code>	<code>ucl_add_string_fuzzer</code>
<code>opensc</code>	<code>fuzz_pkcs15init</code>

3.2.1 Costruzione dell'immagine docker

Per ogni componente rilevante (immagine del programma sotto test, immagine per il compiler/toolchain e immagine per QMSAN) ho creato un'immagine Docker dedicata a partire dal rispettivo Dockerfile. Questo passaggio permette di isolare le dipendenze e di rendere riproducibile l'ambiente.

Un esempio di comando per costruire l'immagine è il seguente:

```
docker build -t libucl_qmsan .
```

Il comando costruisce un'immagine nominata `re2_qmsan` a partire dal Dockerfile nella directory corrente. Ripetendo la build con i Dockerfile opportuni si ottengono le immagini per il compiler e per QMSAN.

3.2.2 Creazione e avvio del Docker

Dall'immagine del programma ho creato un container interattivo per poter esplorare l'ambiente, eseguire comandi manuali e lanciare sessioni di fuzzing in modo controllato. Esempio di comando per avviare un container interattivo:

```
docker run -it -name libucl_fuzzing libucl_qmsan bash
```

dove `-name re2_fuzzing` assegna un nome al container, `re2_qmsan` è l'immagine di partenza, `-it bash` apre una shell interattiva (`bash`) dentro il container così da poter lanciare manualmente gli strumenti.

3.2.3 Avvio di una sessione di Fuzzing con AFL++

La campagna di fuzzing è stata avviata con AFL++ (build integrata con QMSAN). Il fuzzer genera test case, li fornisce al target attraverso l'harness QMSAN e registra informazioni su coverage, crash e hangs nella directory di output. Comando usato per il fuzzing:

```
/qmsan-afl/afl-fuzz -D -U -i seeds/ -o /out/1.1 -m none -t 100+ -  
python3 /sanitizer/qmsan/qmsan ./ucl_add_string_fuzzer @@
```

dove

- `/qmsan-afl/afl-fuzz`: percorso del binario AFL++ integrato con QMSAN (responsabile della generazione dei test case e della gestione della coda);
- `-i seeds/`: cartella contenente i seed iniziali (corpus minimo da cui partire);
- `-o /out/1.1`: directory di output dove AFL salva coda, statistiche, crash, hangs;
- `-m none`: rimuove il limite di memoria imposto al processo target (utile quando il sanitizer aumenta l'uso di memoria);
- `-t 100+`: timeout per l'esecuzione del target impostato a 100 ms (il `+` indica che AFL applica un piccolo margine dinamico);
- `-`: separa le opzioni di AFL dal comando target;
- `python3 /sanitizer/qmsan/qmsan`: l'harness QMSAN che avvia il target sotto il sanitizer/taint-tracker;
- `./ucl_add_string_fuzzer`: eseguibile di destinazione (il target da fuzzare);
- `@@`: segnaposto che AFL sostituisce con il percorso del file di input generato.

3.2.4 Riproduzione manuale di un crash

Quando AFL segnala un crash viene inserito all'interno della cartella `/out/default/crashes` insieme agli altri crashes trovati. Nel caso si abbia la necessità di eseguire il crash per confermare il tipo e visualizzare la stack trace si utilizza il seguente comando:

```
/sanitizer/qmsan/qmsan ./ucl_add_string_fuzzer  
"libucl_1209/id_000000"
```

In questo modo si utilizza `qmsan` con il binario per eseguire nuovamente il crash con `id_000000`.

3.2.5 Analisi con Valgrind

Per indagare problemi di memoria, ho eseguito i crashes precedentemente raccolti sotto Valgrind in modo da ottenere informazioni su leak, accessi invalidi e stack trace. Comando di analisi con Valgrind:

```
valgrind ./ucl_add_string_fuzzer "libucl_1209/id_000000"
```

Valgrind fornisce un report dettagliato sugli errori di memoria che può essere salvato su file o incluso nel log di triage.

Poiché le campagne di fuzzing producono spesso molti crash, ho automatizzato la riproduzione in batch e la raccolta degli output in file di log aggregati. Questo facilita la triage: si possono eseguire tutti i crash presenti in `/out/1.1/crashes`, salvare `stdout/stderr` e produrre report separati per Valgrind.

3.3 Deduplication e Categorizzazione

Ogni crash è causato da un bug, ma crash differenti possono avere origine dallo stesso errore. Da questa osservazione deriva la necessità di distinguere i crash che dipendono da bug differenti. Il processo di analisi dei crash finalizzato a individuare quelli effettivamente unici prende il nome di bug triaging. Poiché il numero di crash può facilmente raggiungere le migliaia, è fondamentale disporre di un processo automatico in grado di raggrupparli in categorie, così da ottenere un sottoinsieme rappresentativo in cui ogni elemento corrisponde a un bug distinto. Questo processo è noto come bug deduplication e, nella pratica, viene spesso implementato utilizzando come discriminanti l'indirizzo del crash e l'identità dei top-k indirizzi presenti nello stack.

In AFL++, i crash vengono distinti principalmente in base alla coverage. Una tecnica diffusa consiste nell'estrarre il backtrace del programma al momento del crash e nel considerare i top-k frame dello stack. La scelta di `k` influisce sul risultato:

- un valore troppo grande porta a un over-count, ossia ogni crash viene considerato unico;
- un valore troppo piccolo porta a un under-count, ossia crash differenti vengono erroneamente raggruppati;

- in pratica, valori come $k = 3$ o $k = 5$ rappresentano un buon compromesso.

Esistono strumenti già pronti che supportano questo tipo di analisi. La deduplicazione consente di ridurre drasticamente l'insieme iniziale dei crash prodotti dal fuzzer a un numero più gestibile di cluster di crash unici. Tuttavia, questo rappresenta generalmente il limite dell'automazione: per associare i crash a bug reali è ancora necessario un triaging manuale, ossia un'analisi del codice per comprenderne la causa radice. Tale compito può risultare complesso, poiché il bug può trovarsi in una porzione di codice lontana dal punto in cui si manifesta il crash.

Nel mio lavoro ho quindi inizialmente utilizzato uno script Python per eseguire il declustering con $k = 5$, fornendo come input i log raccolti in precedenza. Questo script nel caso di errori provenienti da molteplici contesti, prende in considerazione solo il primo errore rilevato e suppone che i successivi siano errori a cascata derivante dal primo. Questo è stato fatto non perchè è ciò che accade in ogni situazione ma è quello che viene considerato come vero dagli sviluppatori per semplicità. Successivamente ho analizzato nel dettaglio i crash unici rimanenti, studiandone la tipologia e gli ultimi indirizzi nello stack. Ho successivamente diviso i vari gruppi secondo l'indizzo di crash.

Capitolo 4

Risultati

Una volta completate le varie campagne di fuzzing, ognuna della durata di 36 ore, ed eseguito il processo di deduplication, ho controllato se i crash trovati fossero già noti agli sviluppatori. A tal fine ho consultato la sezione “Issues” dei repository GitHub dei progetti analizzati [5], [3], [2], [1]. Per la ricerca ho usato parole chiave come l’indirizzo di crash e le prime voci della stack trace, verificando che i problemi riscontrati non fossero già segnalati.

4.1 Panoramica generale

Di seguito vengono riportati i crash riscontrati durante le campagne di fuzzing.

4.1.1 OpenSC

I crash principali per OpenSC sono i seguenti:

```
Use of uninitialised value of size 8
==112==    at 0x4E0C63A: _itoa_word (_itoa.c:180)
==112==    by 0x4E28574: __vfprintf_internal (vfprintf-internal.c:1687)
==112==    by 0x4E360F8: __vsprintf_internal (iovsprintf.c:95)
```

```
==88== Invalid read of size 1
==88==    at 0x14AD7C: asn1_encode_entry (in /out/fuzz_pkcs15init)
==88==    by 0x147671: asn1_encode (in /out/fuzz_pkcs15init)
==88==    by 0x14AC19: asn1_encode_entry (in /out/fuzz_pkcs15init)
```

```
Conditional jump or move depends on uninitialised value(s)
==118==    at 0x483FEDC: strcmp (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck)
==118==    by 0x13B6A5: find_macro (in /out/fuzz_pkcs15init)
==118==    by 0x13B48E: build_argv (in /out/fuzz_pkcs15init)
```

4.1.2 libucl

I crash principali per libucl sono:

```
==18== Invalid read of size 8
==18==    at 0x119F80: ucl_hash_destroy (in /out/ucl_add_string_fuzzer)
==18==    by 0x110B6F: ucl_object_free_internal (in /out/ucl_add_string_fuzzer)
==18==    by 0x11123D: ucl_parser_free (in /out/ucl_add_string_fuzzer)

==21== Invalid read of size 4
==21==    at 0x11A3E4: kh_put_ucl_hash_node (in /out/ucl_add_string_fuzzer)
==21==    by 0x11A045: ucl_hash_insert (in /out/ucl_add_string_fuzzer)
==21==    by 0x10AF3D: ucl_parser_process_object_element (in /out/ucl_add_string_fuzzer)

==24== Invalid read of size 1
==24==    at 0x112B10: ucl_load_handler (in /out/ucl_add_string_fuzzer)
==24==    by 0x10DAD9: ucl_state_machine (in /out/ucl_add_string_fuzzer)
==24==    by 0x10C0D7: ucl_parser_add_chunk_full (in /out/ucl_add_string_fuzzer)
```

4.1.3 GPAC

Il crash principale riscontrato in GPAC è il seguente:

```
==109== Conditional jump or move depends on uninitialised value(s)
==109==    at 0x58305C: hev_parse_vps_extension (in /out/fuzz_probe_analyze)
==109==    by 0x56D1A7: gf_hevc_read_vps_bs_internal (in /out/fuzz_probe_analyze)
==109==    by 0x570CFD: gf_hevc_parse_nalu_bs (in /out/fuzz_probe_analyze)
```

4.2 Case Study: GPAC

Capitolo 5

Conclusione

Bibliografia

- [1] Gpac issues. Available from: <https://github.com/gpac/gpac/issues>.
- [2] Libredwg issues. Available from: <https://github.com/LibreDWG/libredwg/issues>.
- [3] libucl issues. Available from: <https://github.com/vstakhov/libucl/issues>.
- [4] Memcheck: a memory error detector. <https://valgrind.org/docs/memcheck2005.pdf>.
- [5] Opensc issues. Available from: <https://github.com/OpenSC/OpenSC/issues>.
- [6] AFL++. Fuzzing library “libafl”. Available from: <https://aflplus.plus/libafl-book/libafl.html>.
- [7] BOHME, M., CADAR, C., AND ROYCHOUDHURY, A. Fuzzing: Challenges and reflections.
- [8] BRATUS, S., LOCASTO, M. E., PATTERSON, M. L., SASSMAN, L., AND SHUBINA, A. Exploit programming: From buffer overflows to "weird machines" and theory of computation. In *Usenix ;login:* (2011).
- [9] FIORALDI, A., MAIER, D., EISSFELDT, H., AND HEUSE, M. Afl++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 2020)* (2020). Available from: <https://www.usenix.org/conference/woot20/presentation/fioraldi>.
- [10] GITLAB. Coverage-guided fuzz testing. Available from: https://docs.gitlab.com/ee/user/application_security/coverage_fuzzing/.
- [11] GOOGLE. Google sanitizers github repository. Available from: <https://github.com/google/sanitizers>.
- [12] HICKS, M. What is memory safety? (2014). Accessed: July 21, 2014. Available from: <http://www.pl-enthusiast.net/2014/07/21/memory-safety/>.
- [13] HOPPER, G. Software bug etymology (1947).
- [14] KLEES, G., RUEF, A., COOPER, B., WEI, S., AND HICKS, M. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018).

- [15] LLVM. Memorysanitizer documentation. Available from: <https://clang.llvm.org/docs/MemorySanitizer.html>.
- [16] MARINI, M., D'ELIA, D. C., PAYER, M., AND QUERZONI, L. Qmsan: Efficiently detecting uninitialized memory errors during fuzzing.
- [17] MITCHELL, J. C. *Concepts in Programming Languages* (2002).
- [18] PAYER, M. *Software Security: Principles, Policies, and Protection*.
- [19] PAYER, M. The fuzzing hype-train: How random testing triggers thousands of crashes. *IEEE Security & Privacy*, **17** (2019), 78.
- [20] SEREBRYANY, K., BRUENING, D., POTAPENKO, A., AND VYUKOV, D. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference (ATC)* (2012).
- [21] SONG, D., LETTNER, J., RAJASEKARAN, P., NA, Y., VOLCKAERT, S., LARSEN, P., AND FRANZ, M. Sok: Sanitizing for security.
- [22] SOUAG, A., SALINESI, C., MAZO, R., AND COMYN-WATTIAU, I. A security ontology for security requirements elicitation. In *ESSoS* (2015).
- [23] STEPANOV, E. AND SEREBRYANY, K. Memorysanitizer: fast detector of c uninitialized memory use in c++.
- [24] STEPHENS, N., GROSEN, J., SALLS, C., DUTCHER, A., WANG, R., CORBETTA, J., SHOSHITAISHVILI, Y., KRUEGEL, C., AND VIGNA, G. Driller: Augmenting fuzzing through selective symbolic execution. In *Network and Distributed System Security Symposium (NDSS)* (2016).
- [25] SZEKERES, L., PAYER, M., WEI, T., AND SONG, D. Sok: Eternal war in memory. In *IEEE Symposium on Security and Privacy*, pp. 48–62 (2013).
- [26] VALGRIND. Memcheck: a memory error detector (manual). Available from: <https://valgrind.org/docs/manual/mc-manual.html>.
- [27] VALGRIND. Valgrind website. Available from: <https://valgrind.org/>.
- [28] ZELLER, A., GOPINATH, R., BOHME, M., FRASER, G., AND HOLLER, C. The fuzzing book (2019). [Online; accessed 25-May-2020]. Available from: <https://www.fuzzingbook.org/>.