Efficient Seed Generation for Expert-based Directed Fuzzing

A Thesis

Presented in Partial Fulfillment of the Requirements for the

Degree of Master of Science

with a

Major in Computer Science

in the

College of Graduate Studies

University of Idaho

by

Koffi Anderson Koffi

Major Professor: **Dr. Constantinos Kolias**

Committee Members: **Dr. Aleksandar Vakanski**, **Dr. Min Xian**

Department Administrator: **Dr. Terence Soule**

May 2023

Abstract

Fuzzing is a process for discovering inputs in a program that may trigger unexpected behavior. In the past few years, fuzzing has gained traction for the discovery of bugs and security vulnerabilities. However, the exploration of the input space of programs can often be prohibitively expensive. To improve this exploration, several modern fuzzing techniques rely on human expertise to provide plausible initial test cases. However, the process of handcrafting test cases for fuzzing is often strenuous for humans and requires a deeper understanding of the Program-Under-Test (PUT). Also, the use of known inputs to programs often does not trigger vulnerable program behavior or reach potentially vulnerable code locations. To address those issues, we propose a seed generation framework with human-in-the-loop directed attributes. Our proposed framework uses symbolic execution to generate seeds that exercise paths to target program locations. Moreover, our framework enables the visualization of the explored execution paths in binaries for the test inputs. The experimental results of our approach show its effectiveness in improving AFL's performance in discovering software bugs.

Acknowledgments

I want to extend my gratitude to my advisor Dr. Constantinos Kolias for his continuous support and guidance. Also, I am grateful to the committee members, Dr. Alex Vakanski and Dr. Min Xian, for their insightful feedback. I also appreciate the feedback and guidance from researchers at the University of Idaho, namely Dr. Jia Song and researchers from the Idaho National Laboratory (INL) Robert Ivans and Michael Cutshaw. I am also grateful to the Falcon HPC team at INL for their help in conducting the experiments. I am grateful to the Department of Energy (DoE) for partially sponsoring this research under the program DoE-LDRD INL "Target Aware Fuzzing" project.

Dedication

I am thankful to my wife, Sara Koffi, for her unconditional love and continuous support.

Table of Contents

Abstract ii

Acknowledgments iii

Dedication iv

Table of Contents v

List of Figures ix

List of Tables xi

List of Code Listings xii

List of Acronyms xiii

Chapter 1: Introduction 1

Chapter 2: Technical Background & Definitions . . . 3
2.1 Historical Context . . . 3
2.2 Taxonomy . . . 3
2.2.1 Program Analysis . . . 4
2.2.2 Application Domain . . . 4
2.2.3 Input Generation . . . 5
2.2.4 Exploration Strategy . . . 5
2.3 Modern Fuzzers . . . 6
2.4 Overview of AFL . . . 6
2.4.1 Genetic Algorithms . . . 6
2.4.2 Code Coverage . . . 7
2.4.3 Instrumentation . . . 8
2.4.4 Test Cases Generation . . . 8
2.4.5 Fuzzing Strategies . . . 9
2.4.6 Limitations . . . 9
2.5 Challenges . . . 10
2.6 Directed Fuzzing Methods . . . 10
2.7 Human-Machine Collaboration Approaches . . . 10
2.8 Binary Analysis . . . 11
2.8.1 Angr: A Binary Analysis Tool . . . 11
2.9 Symbolic Execution . . . 13
2.9.1 Overview . . . 13
2.9.2 Path Explosion . . . 14

Chapter 3: Problem Statement . . . 15

Chapter 4: Proposed Framework . . . 17
4.1 Architecture . . . 17
4.1.1 Seed Generator . . . 17
4.1.2 Visualization . . . 18
4.2 Algorithms . . . 18

4.2.1 Basic Block Address Extraction 18

4.2.2 Depth-First-Search 18

4.2.3 Directed Symbolic Execution 19

4.3 Implementation 20

4.3.1 Visualization 20

4.3.2 Seed Generation 20

Chapter 5: Experimental Setup & Dataset 21

5.1 Evaluation Dataset 21

5.2 Experimental Setup 22

5.3 Implementation Considerations 23

Chapter 6: Experimental Evaluation 24

6.1 Seed Generation 24

6.1.1 Crash Triage 24

6.1.2 Performance Metrics 24

6.1.3 First Crash Time 25

6.1.4 Unique Crashes 25

6.1.5 Fuzzing Execution Cycles 26

6.1.6 Speed Up 26

6.1.7 Symbolic Execution Overhead 27

6.2 Visualization 31

6.2.1 Paths of Generated Testcases 31

6.2.2 Paths of AFL Testcase Queue 31

Chapter 7: Discussion 35

7.1 Path Visualization 35

7.2 Direction Based on Symbolic Execution 36

7.3 Efficiency 36

7.3.1 Scalability 36

7.3.2 Usability 36

Chapter 8: Related Work **37**

8.1 Human-Machine Collaboration in Fuzzing 37

8.2 Visualization for Human-in-the-Loop Fuzzing 39

8.3 Directed and Human-in-the-Loop Fuzzing 43

8.4 Symbolic Execution in Human-in-the-Loop Fuzzing 45

Chapter 9: Conclusion & Future Work **51**

References **53**

List of Figures

2.1 Fuzzing taxonomies 4
2.2 AFL Genetic Algorithm 8
2.3 Human-machine Collaboration in Fuzzing 11
2.4 angr architecture [1] 12
2.5 Typical binary analysis workflow with angr. 13

4.1 Architecture of the proposed framework. 18

6.1 First crash time 25
6.2 Unique crashes found 26
6.3 Number of executions of the binary 27
6.4 Speedup over AFL for finding the first crash 28
6.5 Seed generation processing times for branch depth and width 29
6.6 Seed generation processing times for condition complexity and vulnerable functions 30
6.7 CFG of function_2 32
6.8 Path traversed by the generated seed seed_1 33
6.9 Path traversed by the generated seed seed_2 34

8.1 Taxonomy of Human-machine collaboration in fuzzing 39
8.2 FuzzSplore visual panel 40
8.3 FMViz sample image representation of testcase 41
8.4 VisFuzz path coverage visualization and fuzzing statistics 42
8.5 VisFuzz Call Graph visualization 42
8.6 VisFuzz Control Flow Graph visualization 42
8.7 JMPscare user interface in Binary Ninja 45

8.8 HaCRS user interface diagram 47

List of Tables

2.1 Popular fuzzers 6

5.1 C programs used in the experiments. 22

8.1 Summary of related works in Human-in-the-loop fuzzing 48

List of Code Listings

5.1 Function **vulnerable** 21
5.2 Example of a program in the evaluation dataset 22

List of Acronyms

AFL American Fuzzy Lop

PUT Program-Under-Test

GA Genetic Algorithm

CG Call Graph

CFG Control Flow Graph

DDG Data Dependency Graph

INL Idaho National Laboratory

DGF Directed Gray-box Fuzzing

HMC Human-machine Collaboration

SE Symbolic Execution

Chapter 1

Introduction

Fuzzing is an automated software testing technique that aims to discover bugs, i.e., unexpected program behaviors, by exploring the alternative inputs. These inputs are typically provided as a vector by the analyst. The total alternative inputs that can be provided to a target program is often referred to as its *input*. Due to the large size of modern programs and their complexity, the exploration of potential alternatives, i.e., *search space* can often be prohibitively time-consuming. However, with an appropriate choice of initial seeds, the bug discovery process can be significantly improved. Nevertheless, human experts typically need a deeper understanding of the Program-Under-Test (PUT) to provide an initial set of high-quality seeds. Thus, in practice, most users revert to known valid inputs to the program rather than investing time in good-seed discovery. Unfortunately, by providing valid inputs to a program, a bug-free behavior is expected to be triggered at least at the early stages of the fuzzing process. Typically, it is only after extensive seed mutations that unexpected behavior can emerge. Thus, it is necessary for the fuzzer to engage in strenuous cycles of seed mutation to unearth vulnerable program behavior. To this day, finding good initial seeds for fuzzing is challenging and remains an active research topic. To address those issues, extensive research has been dedicated to generating seeds for fuzzers automatically. Such approaches often rely on modern techniques such as Deep Learning (DL) and Machine Learning (ML) to create good initial seeds [2]. However, such approaches tend to be fully automated, blind and do not provide insights into their inner workings. Often, a human analyst has experience regarding previously seen patterns that may lead to bugs. What is more, human experts tend to have intuition regarding candidates for the specific location of bugs. Thus starting from or focusing on these areas can dramatically boost

the efficiency of a fuzzer.

Although some research has already been done to enable guided fuzzing (also seen as *directed fuzzing*), most modern fuzzers do not allow the user to guide the process to specific program locations. We argue that enabling the governance of fuzzers towards specific program locations dictated by a human expert via means of specific, well-crafted initial seeds can lead to dramatic boosts in fuzzing efficiency.

We propose a seed generation framework for Directed Fuzzing with Human-in-the-loop functions in the fuzzing process. Our proposed framework uses symbolic execution to generate seeds that guide fuzzing to target program locations. In addition, our proposed framework can track the paths in the program traversed by a particular seed. In this way, feedback can be provided to the human expert.

We implemented our solution using the *angr* binary analysis tool and constructed a dataset of C programs to evaluate our methodology. Our experimental results showed a significant performance over AFL using generic input seeds. Our approach can generate seeds that drastically improve the performance of AFL with up to 1000x speed up for programs having more complex branching conditions and depths. In addition, our proposed solution is non-intrusive and completely decoupled from a particular fuzzer. As a result, most existing fuzzer tools can benefit from our seed generation to improve their performance.

Chapter 2

Technical Background & Definitions

In this section, we present some technical background of fuzzing, binary analysis, and symbolic execution.

2.1 Historical Context

The introduction of the term *fuzzer* was done by Professor Barton Miller in his 1988 CS736 class, at the University of Wisconsin. The purpose of his method was to cause certain unix programs to crash by feeding them with random inputs [3]. About one-third of the programs tested at the time would crash during fuzzing. This approach at the time was proven quite effective at discovering software vulnerabilities. However, this rudimentary approach cannot yield the same success in modern software programs which are order of magnitude more complex. Since then, researchers have introduced several novel techniques [2, 4, 5, 6] to develop more sophisticated fuzzers that improve the overall fuzzing process.

2.2 Taxonomy

Since drastically diverse fuzzing approaches have been developed, a comprehensive fuzzing taxonomy can be useful in classifying alternative types of fuzzers and their characteristics. Fuzzing can be organized into alternative families (Figure 2.1) based on the level of program analysis, the application domain, the input generation, and the fuzzing exploration strategy.

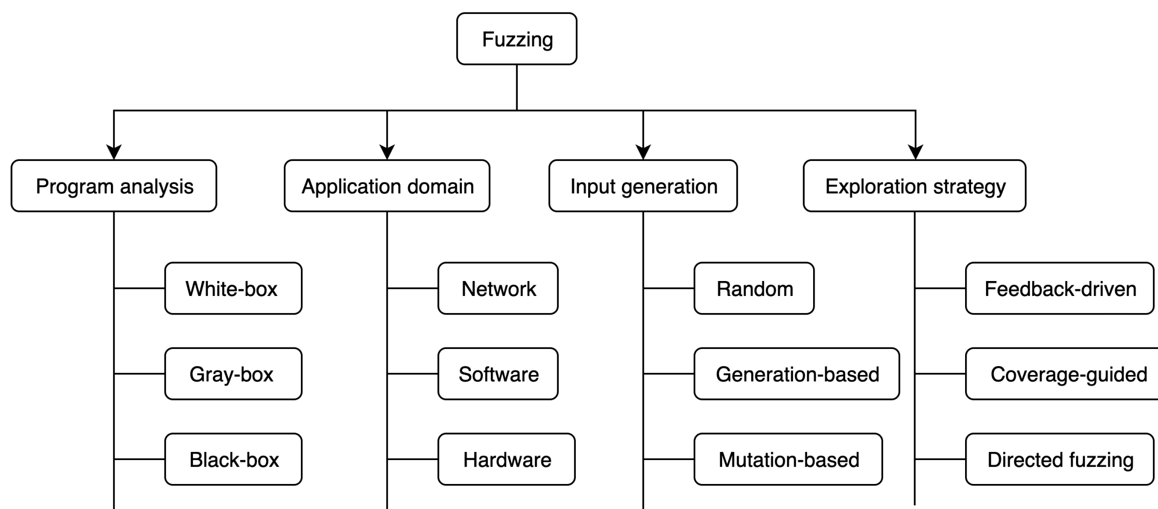


Figure 2.1: Fuzzing taxonomies

2.2.1 Program Analysis

One criterion for classifying fuzzers is based on the degree of program analysis and availability of source code or program's execution information. Fuzzers can leverage the knowledge of a program structure to guide the fuzzing process. White-box fuzzers have access to the program's source code [7, 8, 9], while gray-box fuzzers do not have access to the source code but rely on program analysis techniques to gain specific information about the target program [5, 6, 10]. Black-box fuzzers do not have access to the program's source code and rely solely on external inputs to test the program [11, 12].

2.2.2 Application Domain

Fuzzers can be classified based on their application domain such as software fuzzing (file format fuzzing, API fuzzing, or web application fuzzing), hardware fuzzing (IoT devices, microcontrollers, FPGAs, etc), and network fuzzing (network protocols, networking applications and tools, etc).

2.2.3 Input Generation

Based on the type of input generation used, a fuzzer can be classified as random, generation-based, mutation-based, and evolutionary. Random fuzzers is less used since they generate random input test cases to be fed to a program without performing any analysis. Generation-based fuzzers, at the other hand, uses the information on the structure of the input of the program to generate more plausible inputs. More specifically, generation-based fuzzers take information about the expected input format or protocol through specifications and generate inputs accordingly. They require knowledge of the program and cannot select mutations intelligently. Mutation-based fuzzers attempt to mutate or modify the provided input to feed to the program following several mutation strategies and input scheduling algorithms. In this case, no knowledge of program input is required, but a set of valid initial inputs are usually required. Evolutionary fuzzers build on mutation-based fuzzers by selecting some inputs over others for mutation based on some heuristic.

2.2.4 Exploration Strategy

Fuzzers can also be classified based on the fuzzing exploration strategy used during the fuzzing process. Feedback-driven fuzzing aims to provide feedback to the fuzzer about the program's behavior to guide the mutation selection process. Coverage-guided fuzzing seeks to cover as much program coverage as possible to expose bugs. Directed fuzzing is a more targeted approach that aims to guide the fuzzer towards potential bug locations in a program.

Overall, understanding the taxonomy of fuzzing can help researchers and practitioners choose the appropriate fuzzer for a given testing scenario and ultimately improve the quality and security of software systems. By selecting the right fuzzer and input generation technique, and applying coverage-guided or directed fuzzing approaches, it is possible to increase the likelihood of uncovering vulnerabilities and bugs that might otherwise go undetected. Modern fuzzers such as hongfuzz, American Fuzzy Loop (AFL), and libFuzzer can leverage several of those techniques.

Name of fuzzer	Input Generation	Exploration strategy	Program analysis
AFL	Genetic algorithm	Coverage-guided	Grey-box
AFL++	Genetic algorithm	Coverage-guided	Grey-box
AFLFast	Markov Chains	Coverage-guided	Grey-box
Driller	Concolic Execution	Coverage-guided	Grey-box

Table 2.1: Popular fuzzers

2.3 Modern Fuzzers

Modern fuzzers make use of several sophisticated techniques to enhance the fuzzing process (See Table 2.1). Approaches such as Machine Learning, Deep Learning, Evolutionary computation, and Symbolic and Concolic Execution have been extensively incorporated in the past as part of different stages of the fuzzing process.

Currently, AFL, which is one of the most popular fuzzer, uses a genetic programming for mutating the input seeds. AFL has proven its efficacy in finding several critical vulnerabilities. Therefore, it is often considered a benchmark to compare the performance of newly developed fuzzers.

2.4 Overview of AFL

The American Fuzzy Lop (AFL) is a mutation-based and coverage-based fuzzer [5]. It is an instrumentation-guided, i.e., instruments or adds compile time code for measuring code coverage to the target programs. AFL is also a genetic fuzzer, i.e., uses genetic programming for mutating the input seeds. Thus, AFL uses compile-time instrumentation and relies on Genetic Algorithm (GA) to automatically discover test cases that trigger new internal states in a target binary.

2.4.1 Genetic Algorithms

A Genetic Algorithm (GA) is a search-based algorithm rooted in natural selection and is often used to solve optimization problems [13]. GA perform usually better than random search or brute-force algorithms.

In a GA, the population is a subset of the probable solutions to the search problem. The

elements in the population are ranked based on a fitness function. Thus, a fitness function is a function that returns a score representing the suitability of a given solution, typically referred to as chromosome, i.e., a set of genes. Hence, a fitness score determines the probability of a chromosome (or solution) being chosen for further modifications and testing (in this context referred to as reproduction).

The reproduction are perform with a set of Genetic operators which are a set of operators that can alter the genetic composition of the next generation. The most utilized genetic operators are selection, mutation, and crossover. A selection chooses which candidates in the population can become parents in the next generation. The mutation operator applies random modifications to some candidates in the population. A crossover produces next generation from some combination of parents in the population. The primary goal of a genetic operator is to guide the search algorithm toward suitable solutions.

AFL maintains a queue of testcases as the population of its genetic algorithms. The queue of testcases are the testcases provided by the user and the testcases discovered during fuzzing. AFL uses a fitness function to provide fitness scores that measure the ability of testcase to increase the code coverage. In AFL terminology *interesting* test cases are those that increase the code coverage; while *favorite* testcases are the fastest and smallest *interesting* testcases. AFL uses several genetic operators (bit/byte flips, swaps, deletion, insertion, etc.) to perform mutations and cross-overs of input seeds. The selection of the next generation in AFL is made by popping a test case from the queue. Finally, the reproduction in AFL executed by applying various operations on a test case retrieved from the queue. The tool ensures that all the necessary mutations and crossover (bit flip, swaps, arithmetics, etc.) are done by skipping some which might not lead to good execution paths. Furthermore, the AFL fuzzer replaces the old population (the testcases in the queue) with a new population according to their fitness scores.

2.4.2 Code Coverage

Code coverage a metric often employed to estimate how much of the target's binary is covered during fuzzing. In code coverage fuzzing, a Basic Block (BB) is the unit of measurement of the program coverage. AFL uses the edge/branch measurement method to represent basic

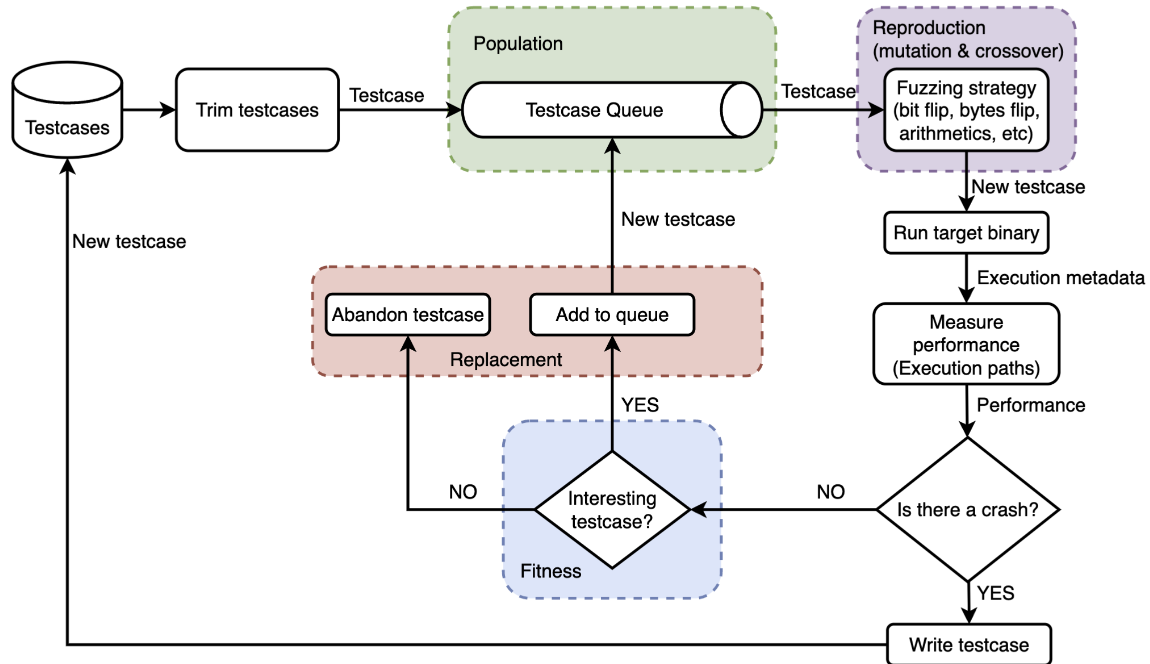


Figure 2.2: AFL Genetic Algorithm

blocks. AFL *shared_mem* array is a 64 kB shared memory region passed to the instrumented binary by the caller. Every byte set in the output map can be considered a hit for a particular $(branch_src, branch_dst)$ tuple in the instrumented code.

2.4.3 Instrumentation

During the instrumentation phase, AFL inject a compile-time random integer value to the variable *cur_location* for each basic block. AFL computes a hash representing a basic block transition from *prev_location* to *curr_location* and stores the corresponding byte value in a bitmap *shared_mem* array. Thus, enables AFL to track the code coverage through code instrumentation and coverage bitmap.

2.4.4 Test Cases Generation

AFL can receive some initial inputs as set of files or as command-line inputs. The content of the inputs are later on mutated by a genetic algorithm. In each iteration, AFL takes the seed, mutates it to generate a test case and adds the test case to the current execution corpus

directory (sequence of branching points explored, e.g., A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D). In all subsequent fuzzing iterations, AFL takes the previous mutated seed, further mutates it, and adds the new test case to the current corpus if it triggers new states (or coverage).

Since the generated files that contain the test cases can become extremely large, AFL trims the test cases files to their minimal size that can still trigger the corresponding internal states.

2.4.5 Fuzzing Strategies

AFL starts its fuzzing strategy following a deterministic approach and proceeds to non-deterministic operations. The deterministic strategies consist of sequential bit flips with varying lengths and stepovers, sequential addition and subtraction of small integers, and sequential insertion of known interesting integers (0, 1, INT_MAX, etc.). In most cases, the deterministic strategies will tend to generate compact test cases with slight differences between the non-crashing and crashing inputs. After the deterministic processes are completed for a particular input file, AFL will continue with a never-ending loop of non-deterministic operations that consist of a sequence of stacked bit flips, insertions, deletions, arithmetics, and splicing of different test cases.

As a last resort, AFL will perform test case splicing. This strategy consists of taking two input files from the queue that differ in at least two locations and splicing them randomly in the middle an operation which resembles crossover. The resulting inputs are run through a stacked tweaks algorithm.

2.4.6 Limitations

Generally, AFL does not attempt to reason about the relationship between specific mutations and program states. Thus, AFL fuzzing steps are inherently blind and are only guided by the evolutionary process of the input queue. Another major limitation of AFL is that its mutation engine is syntax-blind and only optimized for compact data formats such as binary files. Generally, it considered challenging for most general-purpose fuzzers to deal with rich syntax input formats.

2.5 Challenges

Several challenges can impede the performance of fuzzers. Many works have attempted to address those challenges, but several problems still remain open. First of all, fuzzing can be a challenging process due to the large search space of inputs in most real-world programs. Notice that a program with N checks on an integer value in a C program would require a search space of 2^{32*N}, which can be very large, which makes the task of bypassing complex condition statements necessary for fuzzers. In addition, it is often challenging for fuzzer to trigger certain complex bug types. For example, some vulnerabilities, such as race conditions, time bombs, etc., can only be triggered in some specific and rare circumstances. Moreover, most fuzzers cannot fuzz some targets which rely on external dependencies such as events and states. For example, hardware and firmware, as well as stateful software, are often difficult to fuzz. Additionally, fuzzing can benefit from human expertise, but it is often challenging to enable efficient human-machine collaboration in fuzzing. Finally, most fuzzers are intended to be used only by human experts. To this day, improving the usability of fuzzing tools remains elusive.

2.6 Directed Fuzzing Methods

The seminal work by [6] introduced the concept of directed greybox fuzzing which aims to control the fuzzing process to focus on specific parts of a program. Unlike previous approaches based on symbolic execution and constraint solving, directed greybox fuzzing uses compile-time instrumentation and a runtime seed selection algorithm to optimize the distance between inputs and program code targets.

2.7 Human-Machine Collaboration Approaches

Since its inception, fuzzing has enabled human auditors to discover many vulnerabilities. To work effectively, fuzzers can benefit from human knowledge and expertise. In a typical setting, the human provides knowledge about the target program while the fuzzer outputs insights into its fuzzing process to the human (see Figure 2.3). Thus, the human and the fuzzer can

collaborate to enable an efficient fuzzing cycle. However, the level of human involvement in the fuzzing process can be a determinant of the overall performance of the fuzzer. Human-machine collaboration in fuzzing aims involve human expertise in the fuzzing process [14].

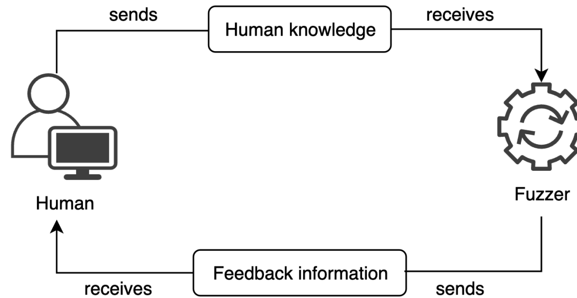


Figure 2.3: Human-machine Collaboration in Fuzzing

2.8 Binary Analysis

Binary analysis is a technique for analyzing programs in their binary format, their properties, and structures [15]. The binary analysis primarily deals with machine code and binary data. Binary analysis can be grouped into two main categories: static analysis and dynamic analysis. In static binary analysis, the analysis engine does not execute the binary during analysis which allows it to scale linearly with the program size. The static analysis engine aims to reason about the behavior of the binary without execution which can often lead to false positives. Thus, static analysis can often be infeasible due to incomplete information about the binary (e.g., dynamically loaded code). In dynamic binary analysis, the analysis engine executes the binary. Therefore, dynamic analysis can scale with the execution length while avoiding false positives in discovery bugs. In general, dynamic analysis is more suitable for programs where appropriate inputs are available. Both type of analyses require the availability of a binary obtained through compilation.

2.8.1 Angr: A Binary Analysis Tool

The Binary analysis tool angr [1] is a popular analysis tool for implementing high-level analysis on binaries. Analysis such as CFG, CDG, Call Graph, Pointer Analysis, etc., can be easily

implemented in angr for most binary executable formats. In addition, analysis tools can use angr to perform Symbolic execution, automatic exploit generation, and automated binary hardening. Angr is comprised of several components (see Figure 2.4).

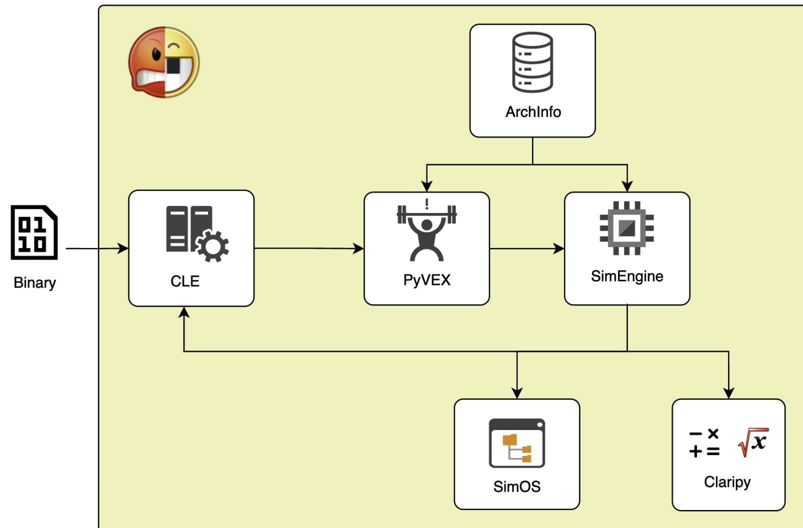


Figure 2.4: angr architecture [1]

One of the most important modules of angr is its binary loader, namely CLE (CLE Loads Everything). The CLE is responsible for loading the binary by extracting code and data from formats such as ELF and PE (Portable Executable). Next, the Archinfo module contains architectural information about the target binary. It is essentially a database of architectural information about the binary such as the compilation platform, the architecture (AMD, ARM, etc.), and the register size (32-bit or 64-bit). Another import module is the SimEngine which is the simulation executor. This module provides the necessary tools for symbolically executing a binary and collecting constraints that can be solved by a solver engine. Another component is the lifter, called PyVEX, which provides machine-code translation to the VEX intermediate representation. The module SimOS provides OS-level emulation and an emulated file system during the analysis. Finally, angr's module Claripy is a constraint solver built on top of the popular Z3 Theorem solver [16]. Figure 2.5 shows a typical binary analysis workflow using angr.

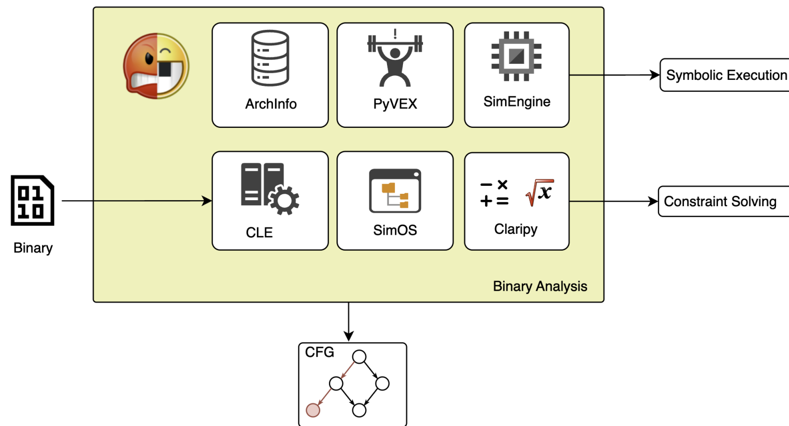


Figure 2.5: Typical binary analysis workflow with angr.

2.9 Symbolic Execution

2.9.1 Overview

Symbolic execution is a software analysis technique that uses symbolic values to analyze programs instead of using concrete values [17]. During the execution, all variables in the PUT are converted to the symbolic representation counterpart, and the operation performed on them are translated to mathematical equations or constraints. Thus, in symbolic execution, a symbolic value or input is a mathematical symbol representing a concrete variable in a program. Instead of concretely running a program, a symbolic execution engine substitutes the concrete values of variables with symbolic values during executions. During execution, the symbolic execution engine maintains a set of symbolic states and an execution context, i.e., a mapping from concrete to symbolic values. For each execution path, a symbolic execution engine will construct a path constraint, which is a logical formula representing the condition an input must satisfy to execute that path. A symbolic execution engine will often rely on a constraint solver to check properties on the path constraints. One of the most well-known solver engines is the Z3 Theorem solver, which is a Satisfiability Modulo Theories (SMT) solver. The Z3 solver can be used to check whether a path constraint is satisfiable or not, i.e., there exists a solution or inputs to exercise the path during execution. Traditionally, symbolic execution has been used extensively to enable directed fuzzing.

2.9.2 Path Explosion

During symbolic execution, a symbolic execution engine checks the path satisfiability and makes a copy (fork) of the symbolic states of satisfiable path constraints for each conditional branching instruction. The forking of states enables a symbolic execution engine to explore different paths without executing the same instructions several times. However, this can be problematic because it doubles the number of states for each conditional branching instruction. Thus, the number of states can increase exponentially based on the number of conditional branching instructions present in a program. In addition, unbounded loops in a program can lead to unbounded symbolic states during symbolic execution. This issue, referred to as the *path explosion* problem, is a common problem for most symbolic execution engines. Recent works have attempted to address the path explosion problem with various techniques, such as directed symbolic execution [18, 19, 20], selective symbolic execution [21, 22], using fuzzing in symbolic execution, and concolic execution [8, 23].

Chapter 3

Problem Statement

Fuzzing can be viewed as a human-machine collaboration process. The human involvement in the fuzzing process can be limited to having the human act as a mere user. In all cases, the human need to provide good initial test cases can undermine their usability because of the difficulties in handcrafting valuable test cases. Thus, a tool for discovering good initial seeds for fuzzers in a human-in-the-loop fashion is highly desirable. In addition, it is often challenging to stir up the fuzzing process toward the direction of potentially vulnerable or interesting locations in the programs during fuzzing. It is often desirable for the human auditor of a software program to visualize the different areas in the program exercised during the analysis. But, visualization tools [24, 25] often fail to provide insight into the analysis.

Many works have attempted to incorporate human knowledge and insight into the fuzzing process [24, 25, 26, 27]. However, they often fail to enable the human expert to guide the fuzzing process toward paths of interest.

We aim to address those issues with a Directed fuzzing approach using symbolic execution and visualization in human-in-the-loop fuzzing. Symbolic execution is highly susceptible to the path explosion problem, which we address with directed symbolic execution on the shortest paths to target basic blocks. Thus, we use user-provided targets; we address the issue of lack of guidance in the fuzzing process and the problem of handcrafting initial seeds. Moreover, we provide a visualization tool for the test cases, which enables practical insight into the fuzzing process and can inform human experts regarding the performance of each input seed.

The main contributions of our work are as follows:

A seed generation tool. Prior works [20, 28, 29] have demonstrated the use of directed symbolic execution to generate inputs for testing specific parts of programs. Our proposed approach differs from these works by leveraging human-in-the-loop for seed generation. The experimental results show that our approach can increase the speed of finding the first crash up to 10,000x compared to AFL.

A visualization tool. We observe that visualization can help human experts get a more in-depth understanding of a PUT. Related works [24, 26, 27] have demonstrated the effectiveness of visualization in providing insight to human experts to improve the fuzzing process. We propose a visualization of the Call Graph (CG) and Control Flow Graph (CFG) for fuzzing test cases, which can give insightful information to human experts.

Chapter 4

Proposed Framework

In our work, we proposed a HMC framework for an effective seed generation using symbolic execution for DGF. Our proposed framework uses a human-in-the-loop approach to enable directed fuzzing on existing fuzzers by generating seeds that can effectively reach specified target code locations in a binary during fuzzing.

4.1 Architecture

Our framework consists of an input seed generator and visualization components that aim to help the human expert directly guide AFL's fuzzing process toward target code locations. Both tools complement each other by allowing the human-expert to not only generate good initial seeds which target specific code locations, but also to visualize whether those seeds can reach those locations during fuzzing.

4.1.1 Seed Generator

The seed generator takes as input a list of target functions and the program's binary to fuzz and generate initial seeds for the fuzzer. The generated seeds can be included in the seeds corpus to fuzz the program.

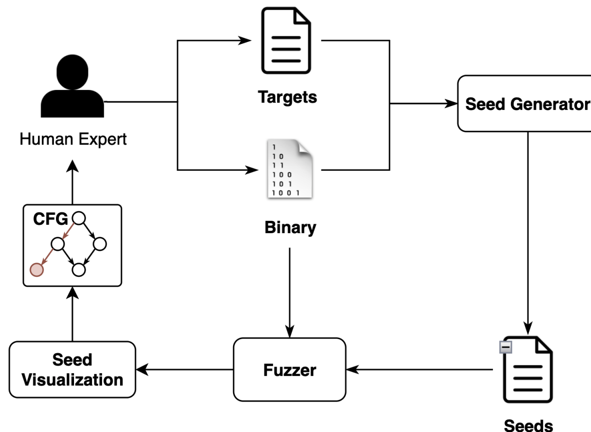


Figure 4.1: Architecture of the proposed framework.

4.1.2 Visualization

In addition, the human expert can visualize the paths taken by certain seeds in the program's CFG or can choose to monitor the progress of the fuzzing by visualizing the paths taken by all seeds in the fuzzer's seeds queue. This can help the auditor to discard some seeds which take potentially non-interesting paths in favor of those which take paths that contain suspicious functions.

4.2 Algorithms

4.2.1 Basic Block Address Extraction

We extract the addresses of the basic blocks from the user-provided target function calls. Our tool is able to retrieve the address of the function calls based on their name. We present our algorithm to retrieve all the addresses of target function calls in Algorithm 1.

4.2.2 Depth-First-Search

The main goal of our approach is to reach deeply hidden bugs. To do so, we prioritized paths to target basic blocks of interest based on the user-provided target functions. We use depth-first-search to force the symbolic execution to prioritize depth over breadth during the search for the basic blocks. Finally, we pruned the symbolic execution states by prioritizing basic block

Algorithm 1 Finding target function call addresses

Require: *targets* ▷ A set of function call targets
Require: *CFG* ▷ The Control Flow Graph of the binary program
Ensure: *addresses* ▷ A set of basic block addresses containing the function calls
  for function in $CFG.functions$ **do**
    for block in $function.blocks$ **do**
      if last instruction in the block is call instruction and call in *targets* **then**
        $addresses \leftarrow block.address$
      end if
    end for
  end for

nodes along the shortest paths to target basic blocks. The shortest paths are computed using Dijstra's shortest path algorithm. Our approach aims to address the issue of path explosion by only exploring in a depth-first-search approach the shortest paths to target basic blocks (see Algorithm 2).

Algorithm 2 Depth-First-Search and Shortest Path of the CFG exploration (path pruning)

Require: *simgr* ▷ The symbolic execution simulation manager
Require: *CFG* ▷ The Control Flow Graph of the binary program
Require: *addresses* ▷ A set of basic block addresses containing the function calls
Ensure: *paths* ▷ The shortest paths
  $paths \leftarrow set(all_shortest_paths(CFG, target = t)$ for t in $addresses)$
  for source, dest in $dfs_edges(CFG)$ **do**
    for node in addresses **do**
      if $(source \neq node$ or $dest \neq node)$ or not node in *paths* **then**
        $simgr.move(from_stash = "active", to_stash = "pruned")$
      end if
    end for
  end for

4.2.3 Directed Symbolic Execution

We apply symbolic execution with directed fuzzing to reach specific basic blocks of code based on a user-provided target function on the shortest paths in a depth-first-search manner. When a target is reached, a seed is generated by solving the constraint on the current path.

Algorithm 3 Directed symbolic execution on the shortest paths to targets

Require: *simgr* ▷ The symbolic execution simulation manager
Require: *CFG* ▷ The Control Flow Graph of the binary program
Require: *addresses* ▷ A set of basic block addresses containing the function calls
Ensure: *seeds* ▷ The input test cases

```
paths ← set(all_shortest_paths(CFG, target = t) for t in addresses)
while length of simgr.active >= 0 do
    simgr.step()
    path_pruning(simgr, CFG, addresses)        ▷ Path pruning based on algorithm 2
    if any (address in simgr.active for address in addresses) then
        seeds ← solve(simgr.active)            ▷ Find inputs that satisfy the path constraints
        break
    end if
end while
```

4.3 Implementation

We implement our framework as a set of two tools: a seed generation and a visualization tool. We use the Python binary analysis tool angr to perform symbolic execution and CFG recovery from the binary. We construct a custom exploration strategy that leverages angr simulation manager and uses static code analysis to identify target locations. Our seed generator is completely decoupled from the fuzzer and can be used with any existing fuzzers. Also, we provide a visualization of the path taken by the generated seeds on the CFG of the binary using Angr, which we export as a graph DOT file and as an image file.

4.3.1 Visualization

Our seed visualization tool recovers the CFG of the binary. Next, our tool symbolically executes the binary with the provided seeds while tracing the exercised paths. The Python package *networkx* is used to obtain the graph DOT file and an image of the paths exercised in the CFG.

4.3.2 Seed Generation

Our seed generation tool uses the recovered CFG of the binary to guide the symbolic execution toward target basic blocks. We implement the path pruning using the Python *netwokx* package using a depth-first-search for the target basic blocks and prune the basic block nodes that are not included in the shortest paths to the targets.

Chapter 5

Experimental Setup & Dataset

5.1 Evaluation Dataset

We built a dataset of 12 small C programs with various levels of complexity (see Table 5.1). Our focus is mainly on the ability to reach the deepest branching locations in the programs while solving complex branching conditions. We restricted the nested branching to 6 to allow conventional AFL to discover bugs within a reasonable time frame of 24h. In addition, we created several other C programs with varying branch conditions and widths, as well as dangerous C function calls (gets, scanf, strcpy, strcat, etc.).

Listing 5.1: Function vulnerable

```
void vulnerable() {
  char buffer[24];
  printf("Enter your name: ");
  scanf("%s", buffer);
  printf("Hello, %s!\n", buffer);
}

```

All the programs called a vulnerable function named **vulnerable** (see example), which contains a buffer overflow vulnerability. Moreover, we added vulnerable function calls in the test programs for testing vulnerable function calls.

Programs	Metrics	Number of programs (n)
depth_n	branch depth	3
width_n	branch width	3
condition_n	complexity in branch conditions	3
function_n	dangerous function calls	3

Table 5.1: C programs used in the experiments.

Listing 5.2: Example of a program in the evaluation dataset

```
1  int main() {
2    printf("Enter the code: ");
3    char a = getc(stdin);
4    char b = getc(stdin);
5    if (a == 123 && b == 34) {
6      printf("Access Granted.\n");
7      vulnerable();
8    } else {
9      printf("Access Denied!\n");
10   }
11   return 0;
12 }
13
```

5.2 Experimental Setup

We perform our fuzzing experiments on a set of nodes on Falcon HPC clusters. We run a set of 2800 jobs as seven sets of 400 Slurm jobs array in computing clusters with each job running for 24h for custom C programs.

5.3 Implementation Considerations

A key challenge in assessing the performance of our approach is the level of the complexity of the constructed programs. Programs with trivial branching conditions are easily traversed by both standard AFL and our proposed approach. However, they are not interesting as, in both, the discovery is rapid. On the other hand, a very complex branching condition might be too difficult for AFL to find the right seed; thus, no meaningful comparison can be made in a reasonable time frame. We constructed a set of programs with increasingly complex branching conditions to allow AFL to find some crashes in a 24h fuzzing session and be complex enough to show the improvement gain of our approach.

Additionally, it is challenging to record the first time a crash occurred during AFL fuzzing session. By default, AFL does not store the first time a crash happens during fuzzing in its output directory. So, we modified AFL to include it as a field **first_crash** in the output directory statistics. This metric **first_crash** records the timestamp of the first unique crash, which helps with our assessment.

Another challenge is the ability to stop AFL after some crashes have occurred. For example, the number of cycles is influenced by the running time of the fuzzing sessions, and it might be unfair to continue after the goal of reaching a set number of crashes is reached. AFL does not allow the users to stop the fuzzing session after some unique crashes have occurred. We modified AFL to enable the use of an environment variable, namely **AFL_STOP_AFTER_N_-UNIQUE_CRASHES** to stop the AFL fuzzing session after ‘n‘ unique crashes have occurred.

Chapter 6

Experimental Evaluation

6.1 Seed Generation

6.1.1 Crash Triage

We filter the fuzzing results to only include the fuzzing session with at least one crashing input. We store results of all the fuzzing metrics from AFL, such as **cycles_done**, **unique_crashes**, etc., as well as a custom metric, **first_crash_time** to measure the first crash time.

6.1.2 Performance Metrics

AFL uses a set of metrics to measure the performance of the fuzzing process. The metric **cycles_done** counts the fuzzer number of passes on the queue's interesting testcases, where a pass includes fuzzing the binary with a testcase. On the other hand, **execs_done** measures the number of times the program was run since the fuzzing process started. The **unique_crashes** measures the unique crashes discovered during the fuzzing session. We use those metrics to measure the performance impact of our seed generation on AFL. In addition, we modified AFL to add a metric for measuring the first time a crash was discovered, which we named **first_-crash_time**. These metrics are stored in the AFL output directory. We use those metrics to evaluate the performance of our approach on the branch depth, width, condition complexity, and the number of vulnerable functions in the programs.

6.1.3 First Crash Time

A desired goal of fuzzing is to trigger a crash as fast as possible. Thus, a good metric is the first time a crash is discovered during a fuzzing session. We report the first time AFL finds a crash with a generic seed and when AFL finds a crash with our generated seed. The experimental results show a significant improvement over AFL for each program tested (see Figure 6.1).

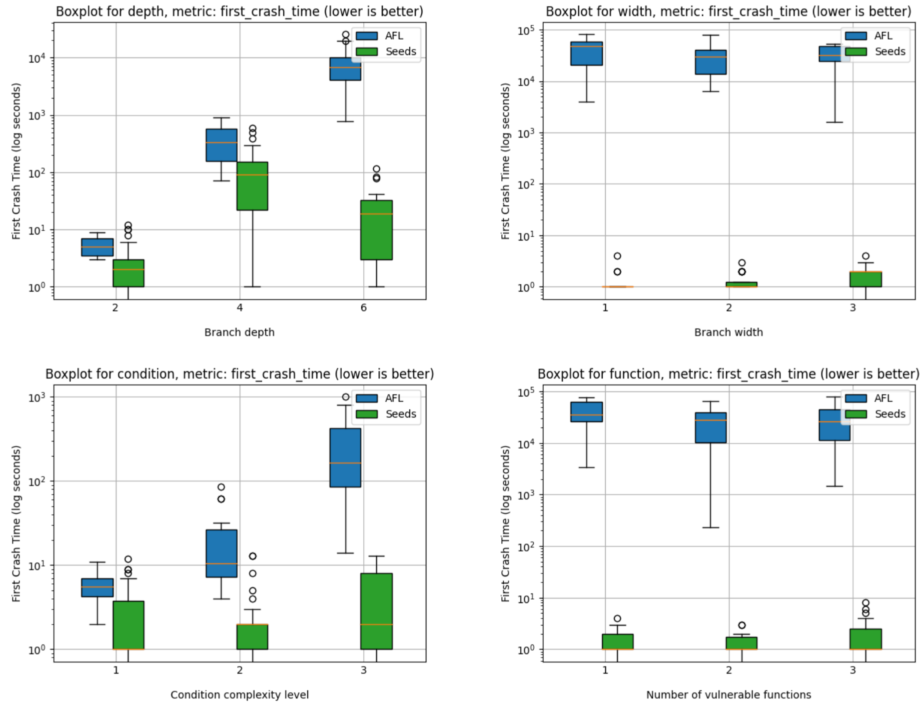


Figure 6.1: First crash time

6.1.4 Unique Crashes

Another good measurement of the performance of a fuzzer is based on the number of unique crashes found during fuzzing. In the case of AFL, a unique crash consist of the seed used to trigger the crash and where in the program the crash occurred. We report the unique crashes found in the programs with multiple vulnerable function calls. Our results show that our approach can find all the unique crashes in significantly less time than AFL.

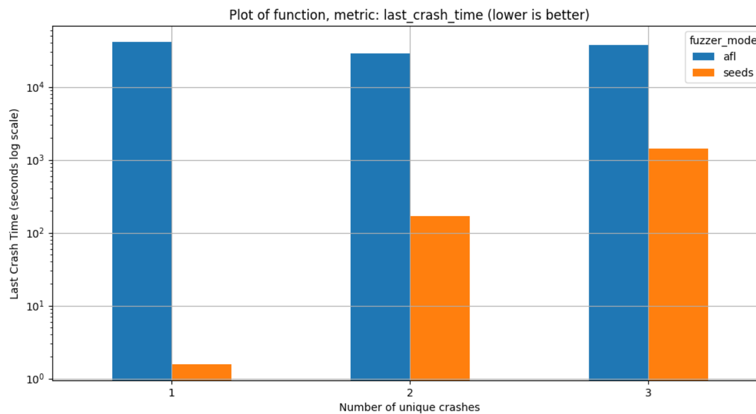


Figure 6.2: Unique crashes found

6.1.5 Fuzzing Execution Cycles

The number of execution cycles in a fuzzing process can be a good indication of the effectiveness of the initial seeds. Poor seeds usually exhibit the behavior of causing the fuzzer to make multiple passes through the program without finding a new execution path. Ideally, good initial seeds should enable the fuzzer to have fewer execution cycles. We report the number of execution cycles accomplished by AFL and our approach. Our results show that we drastically reduced the number of execution cycles needed to find a first crash (see Figure 6.3).

6.1.6 Speed Up

Overall, we measure the speed-up achieved for all our metrics of interest. We report the speed-up performed for the first time a crash occurred, the number of execution cycles, etc. Our results show that we achieve a speed-up in finding the first crash for the branch depth, width, condition complexity, and vulnerable functions (see Figure 6.4). We use equation 6.1 to measure the speed up over AFL in our metrics.

$$speedup = \frac{AFL_{first_crash}}{Our_{first_crash}} \tag{6.1}$$

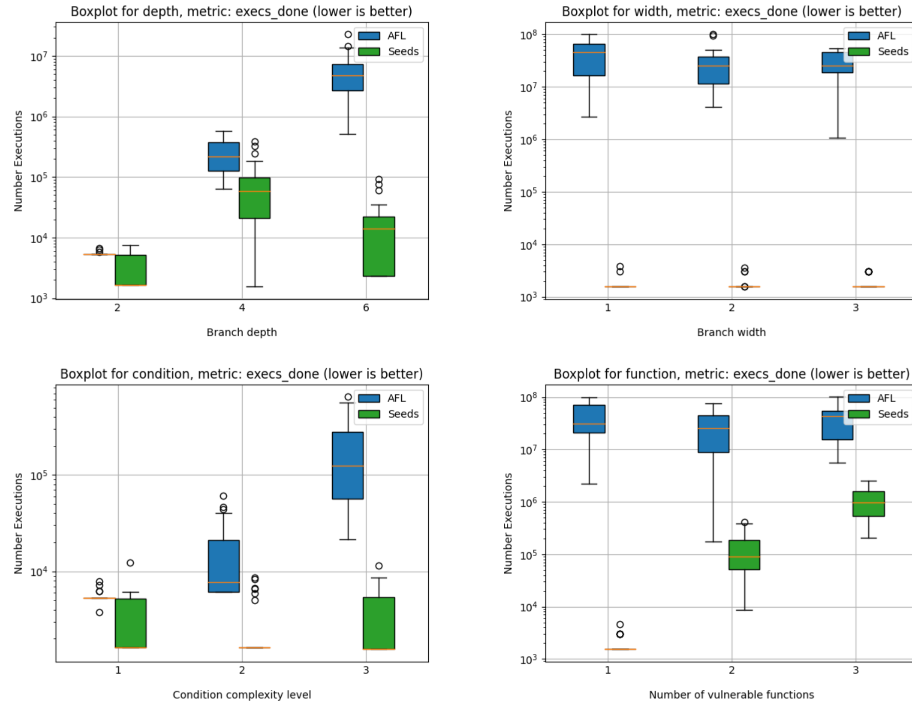


Figure 6.3: Number of executions of the binary

6.1.7 Symbolic Execution Overhead

Our seed generation uses symbolic execution to solve the path constraints to reach desired code location to generate seeds. We measure the time it takes to create the seeds to reach all the target locations. Symbolic execution is known to suffer from path explosion and can add an extra time overhead during the seed generation.

We measure the preprocessing time for our seed generation to evaluate the overhead of our approach. The preprocessing time includes the Control Flow Graph (CFG) generation, the filtering of function calls, the extraction of path constraints, the symbolic execution with path pruning, and the seed extraction from the constraints' solutions. Figure 6.5 and Figure 6.6 show the time spent during preprocessing.

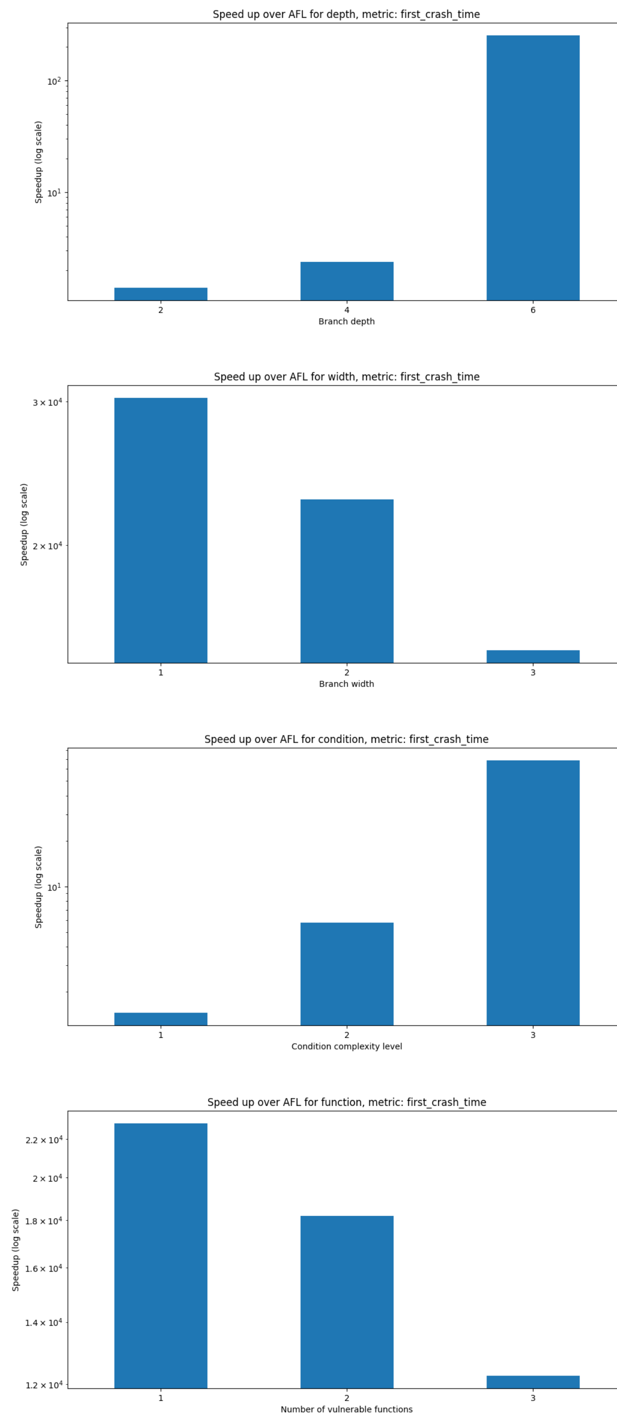


Figure 6.4: Speedup over AFL for finding the first crash

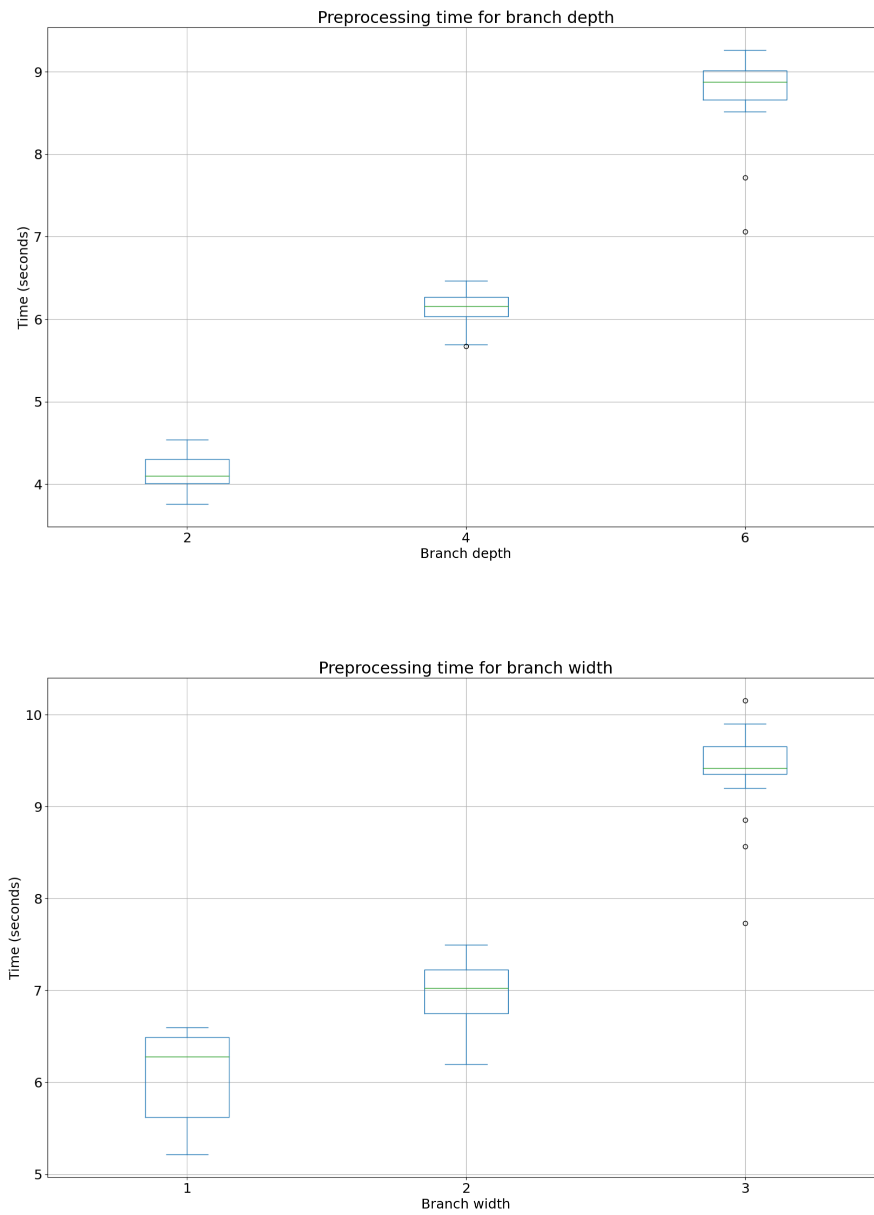


Figure 6.5: Seed generation processing times for branch depth and width

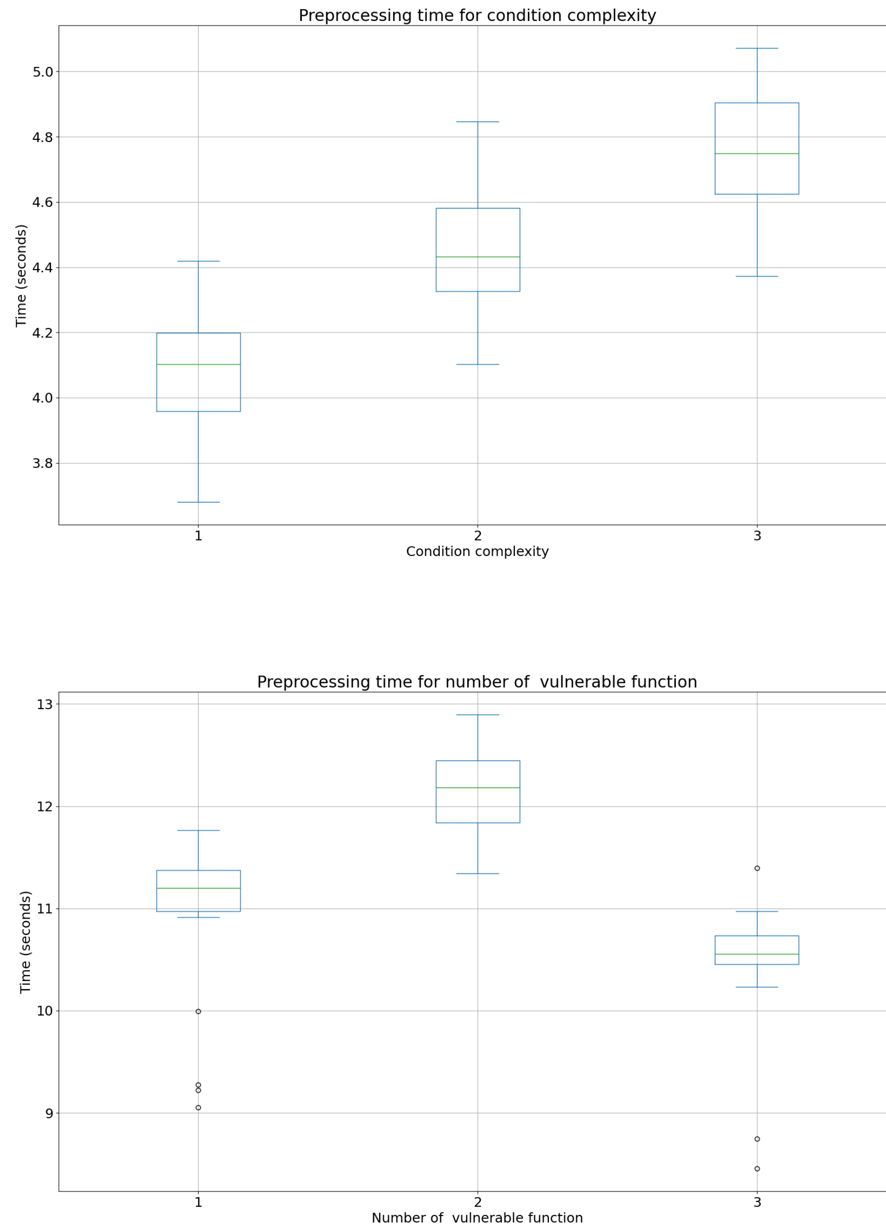


Figure 6.6: Seed generation processing times for condition complexity and vulnerable functions

6.2 Visualization

We track each seed's paths in the CFG to evaluate their effectiveness in reaching the target code locations. After finding a bug, we use our visualization tool to generate the paths traversed by each seed in our generated corpus and AFL's queue.

6.2.1 Paths of Generated Testcases

Our generated seeds can reach all the vulnerable target calls in the binary. Our tool generates an image showing the different paths to the vulnerable target calls for each seed (see Figure 6.9). We track the different paths in the CFG exercised by each seed to evaluate their effectiveness to reach the target code locations.

6.2.2 Paths of AFL Testcase Queue

We use our tool to visualize the target basic blocks reached by each seed in the AFL test case queue after mutating a generic input seed.

Figure 6.7: CFG of function_2

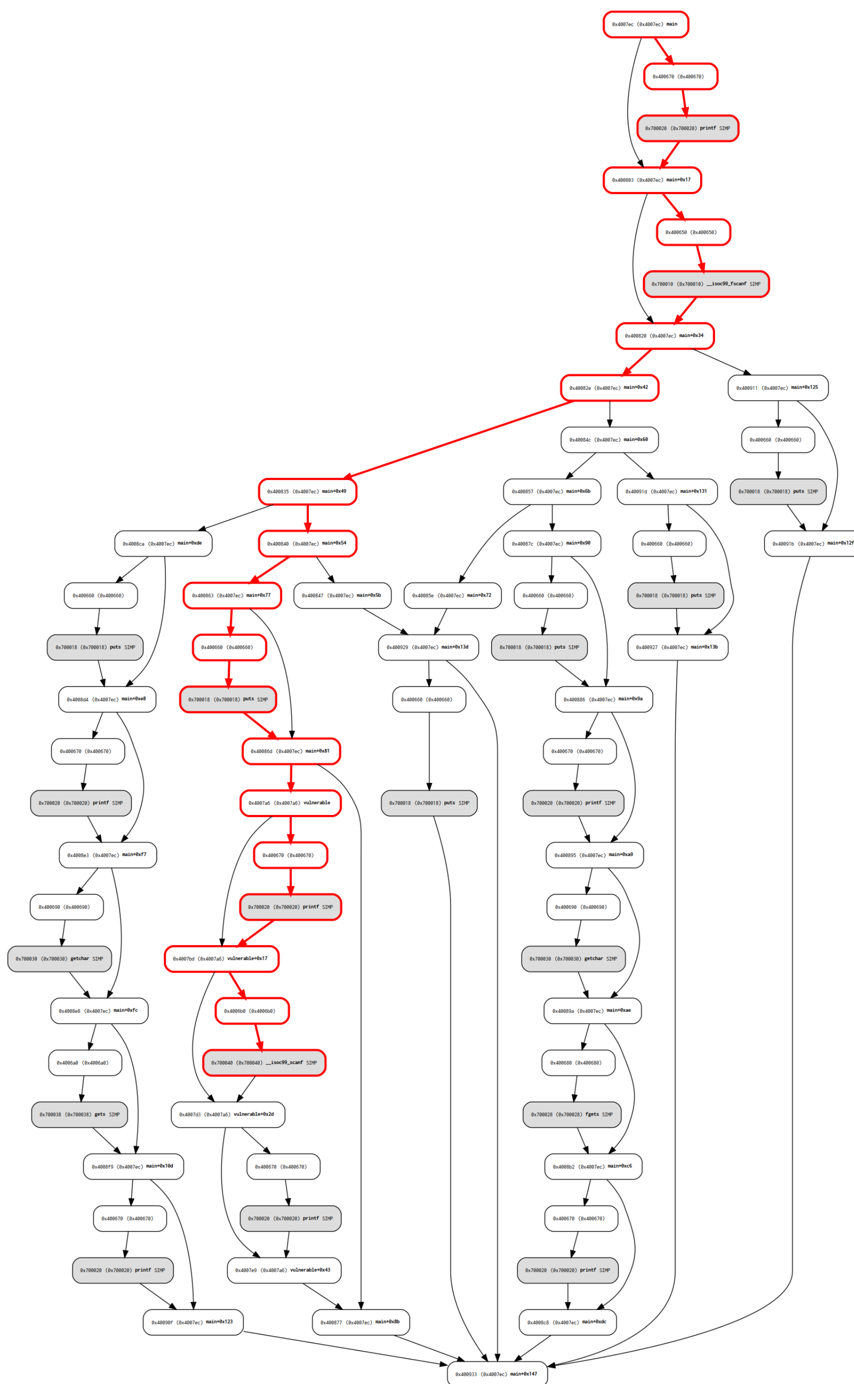


Figure 6.8: Path traversed by the generated seed seed_1

Figure 6.9: Path traversed by the generated seed seed_2

Chapter 7

Discussion

Previous works have demonstrated the effectiveness of good initial test cases in the fuzzing process [30]. In addition, a human-in-the-loop approach can significantly impact the performance of existing fuzzers [14]. Moreover, a fuzzing process can benefit from a human-in-the-loop approach in two ways. First, the fuzzer can provide insight into its internal workings to the human using visualization. Second, humans can provide knowledge about the target program to guide the fuzzing process toward interesting program locations. Our results support and augment these findings.

7.1 Path Visualization

Our approach enables human-in-the-loop fuzzing through the visualization of generated and user-provided seeds' paths in the CFG. Our tool can show the path taken by each seed provided by the user or produced by AFL mutation strategies. This gives valuable information to the human in understanding the path exercised during fuzzing. In addition, the user is able to determine when the fuzzer is stuck or unable to make meaningful progress through the visualization of its test case queue.

7.2 Direction Based on Symbolic Execution

Our results show that the seeds generated by our tool can exercise the path of interest. In addition, our results show significant improvement over AFL in finding the first crash that occurs in programs.

7.3 Efficiency

The examination of our results show an improvement over AFL in scalability and usability. Both our visualization tool and seed generation tool complement each other to achieve a human-in-the-loop efficient fuzzing process.

7.3.1 Scalability

Our results show that our framework can scale with the complexity of the conditional branching instructions. This is demonstrated with results of the set of experimental programs where the condition of the if branches of increasing complexity. In addition, as we increase the depth of the nested conditional branches, our tool is able to maintain a near linear performance. Also, increasing the width of the branches a in our test programs, our approach main significant speed up over AFL in finding the first crash. Finally, our seed generation tool incurs a very small overhead (a few seconds) and can scale with program size and discover all the of bugs present in our experimental programs.

7.3.2 Usability

The visualization of user-provided and mutated seeds can enable non-expert humans to get insight into the inner workings of a fuzzer. This can proof useful on two angles. First, human can leverage this information to understand the input structure of the program. Second, the human can understand roadblocks which the fuzzer encounters during fuzzing. Thus, our approach can contribute to improvement in the usability of existing fuzzers.

Chapter 8

Related Work

8.1 Human-Machine Collaboration in Fuzzing

While the most important tasks of fuzzers are implemented as software that acts autonomously, there are several important steps that require or can be benefited from the intervention of a human. From this viewpoint, the fuzzing process in its entirety can be regarded as a collaboration between a human expert and a fuzzer.

The level of involvement of the human expert in the fuzzing process is indicative of the *class of interaction*. The survey in [14] proposed three main classifications of human-machine collaboration in fuzzing: *Human-out-of-the-loop*, *Human-on-the-loop*, and *Human-in-the-loop*. This classification assumes the following roles for the human in the fuzzing process:

- *User*, where the human only uses the tool to retrieve the results;
- *Assistant*, where the human provides information to the fuzzer regarding the structure of the target program with the aim to possibly speed up or improve the quality of the results of the fuzzing process;
- *Supervisor*, which monitors or receives important insights about the fuzzing process, so that subsequent fuzzing rounds become more efficient;
- *Collaborator*, where the human can update the configuration or provide recommendations that can significantly impact the course of the ongoing fuzzing process.

Human-out-of-loop: In Human-out-of-loop approaches, the human acts merely as a User and does not provide any guidance or aid to the fuzzer nor the human attempts to increase the fuzzing efficiency. Works in this category focus on using techniques such as machine learning for example with the aim to automatically generate testcases [5, 9, 31]. For example, *Driller* [9] uses selective concolic execution to explore favored paths by AFL to find deeply hidden bugs.

Human-on-the-loop: In Human-on-the-loop, the human can take the role of an Assistant or Supervisor to guide the fuzzing process. In this approach, the fuzzer can incorporate human knowledge in its test case generation procedures [6, 10]. For example, Nautilus [32] uses a user-provided context-free-grammar and code coverage feedback to generate semantically and structurally correct test cases.

Human-in-the-loop: Finally, in human-in-the-loop fuzzing, the human acts as a *Collaborator*. Additionally, this approach can provide better information about the target program to humans and leverage the use of an interactive environment [25] to increase the fuzzer's efficiency. This approach is particularly useful for target programs that are overwhelmingly complex.

Many works have explored different ways to improve the efficiency and effectiveness of fuzzing techniques using *visualization*, *guidance*, and *symbolic execution*. Hereunder, we shall analyze some of the most influential of these approaches that fall under these categories. Notice that while such approaches are applied to all three classes of human-fuzzer collaboration we will insist in Human-in-the-loop approaches due to their potential and research interest.

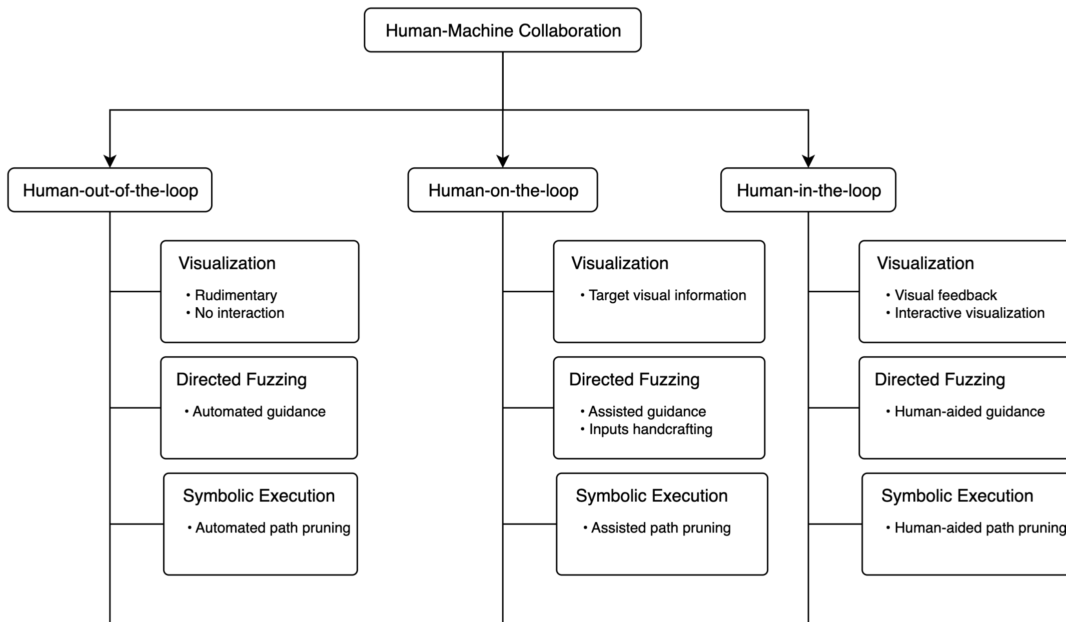


Figure 8.1: Taxonomy of Human-machine collaboration in fuzzing

8.2 Visualization for Human-in-the-Loop Fuzzing

Many works have explored ways to improve the efficiency of fuzzing techniques by leveraging visualization with a human-in-the-loop approach. In this section, we will present works that aim in employing visualization elements as means to provide insights to the human regarding various aspects of the fuzzing execution.

In [24], the authors proposed *FuzzSplore*, which provides a corpus of test cases to AFL++ to create various visual analysis graphs such as Testcases Scatterplot, Coverage Growth Plot, Interesting Testcases Plot, and Generations Graph. These graphs provide a visual summary of important aspects of the fuzzing process. For example, the Testcases Scatterplot helps to visualize the relationship between test cases by plotting them based on their coverage similarity. The Coverage Growth Plot shows the increment in code coverage over time during the fuzzing process. The Interesting Testcases Plot shows the test cases which AFL++ considers the most interesting as they are generated during the fuzzing process. The Generations Graph shows the evolution of test cases and the coverage achieved by each fuzzer over time. Overall, FuzzSplore

provides a helpful tool for understanding the behavior of fuzzers and identifying areas for improvement. However, it fails to enable human interaction with a fuzzer during its execution. Instead, the focus is to understand the behavior of the fuzzer so that the human expert can apply optimal configurations as these are discovered, to subsequent fuzzing cycles. Moreover, traditional graphing techniques can benefit mainly trained experts in the area and they do not provide significant aid to generalist analysts.

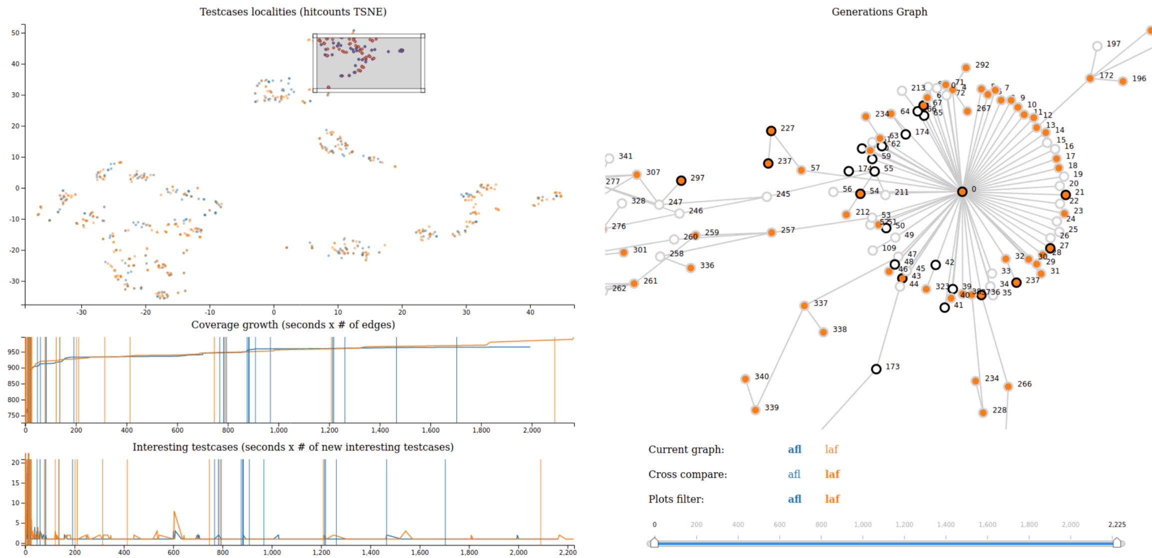


Figure 8.2: FuzzSplore visual panel

In [26], the authors proposed *FMViz*, a tool for visualizing the mutation patterns generated by AFL at the byte level. The tool uses inputs that are generated as AFL mutates the original seeds. The tool provides a visualization of byte-level mutations in fuzzers, which can help researchers and developers understand the patterns and identify areas for improvement. The authors applied their tool to libxml2 and demonstrated how it could be used to visualize and analyze the mutation patterns from the inputs generated by AFL. Although the tool enables the visualization of the inner workings of AFL mutation patterns, it fails to provide an interactive environment to allow the human expert to utilize this information to provide feedback to the fuzzer. Furthermore, using images to capture the information about the mutated bytes of the test cases can fail to capture rich semantic information of complex inputs.

Another work [27] proposed *VisFuzz*, which provides a Web interface for real-time visu-

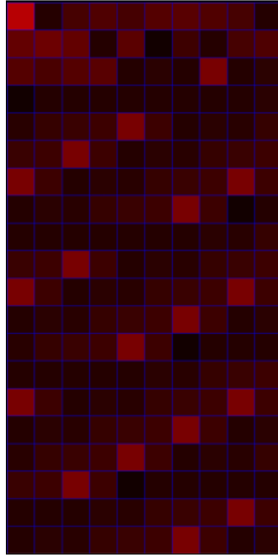


Figure 8.3: FMViz sample image representation of testcase

alization of the fuzzing process by extracting a call graph and the control flow graph from source code. The tool uses test cases and AFL bitmap to provide charts and a Call Graph and the CFG of the target program. The tool is implemented on top of a modified version of AFL as an LLVM plugin. The authors demonstrated how their tool could be used to identify and understand the behavior of the fuzzer and help users improve the overall efficiency of the fuzzing process. But, this approach relies on the availability of the source code and cannot be used for black-box fuzzing. Moreover, it expects from the human expert to effectively utilize the visualization information and manually handcraft test cases which can be a challenging undertaking especially for complex programs.

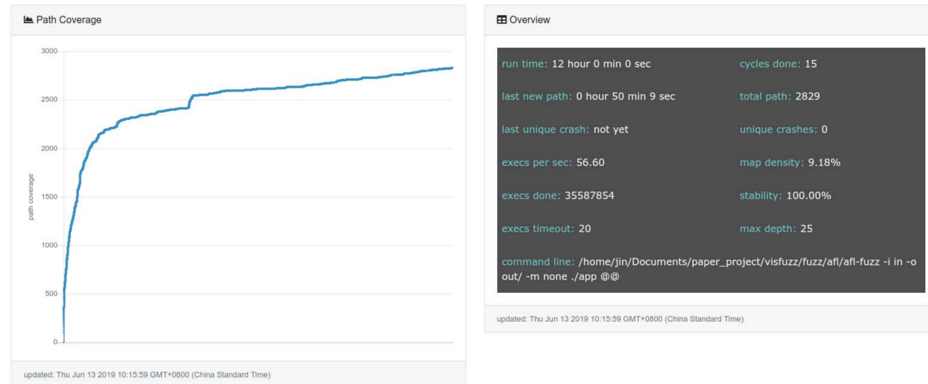


Figure 8.4: VisFuzz path coverage visualization and fuzzing statistics

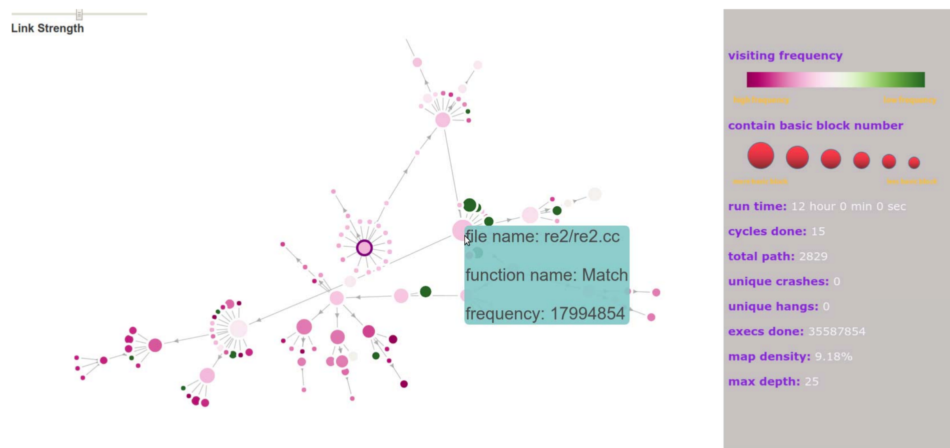


Figure 8.5: VisFuzz Call Graph visualization

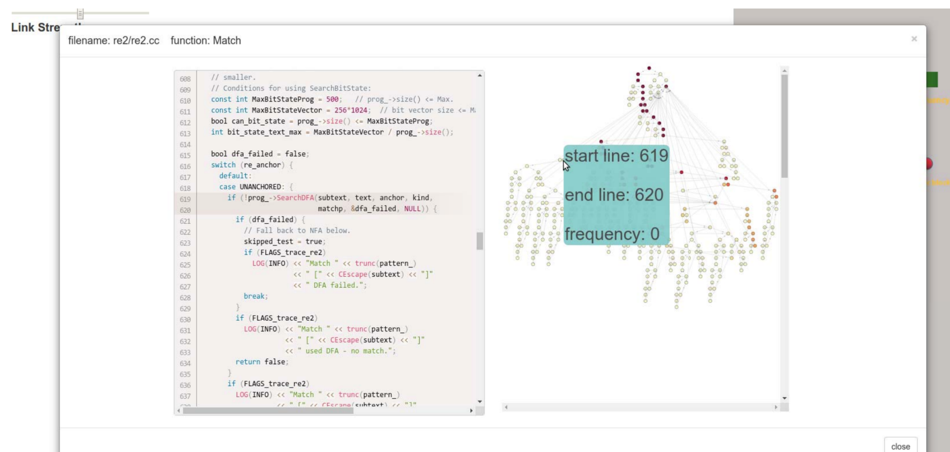


Figure 8.6: VisFuzz Control Flow Graph visualization

8.3 Directed and Human-in-the-Loop Fuzzing

Directed fuzzing is a type of fuzzing in which the goal is to reach specific program locations [6, 33]. This can be done by manually defining these locations e.g., specific functions in a human-in-the-loop fashion or in an out-of-the-loop fashion by simply defining fitness metrics. The concept of Directed Fuzzing was first introduced in the work of [6] where the authors used a simulated annealing-based power schedule to give more mutational time to seeds that are closer (based on a distance metric) to the target locations. Subsequent works have capitalized various techniques to effectively provide guidance towards a specific portion of the programs during fuzzing. One area of focus is the identification of the type of target location in the program. In [34], the authors used Git commits to identify the targets. Other target identification approaches include the use of vulnerability descriptions [21, 35, 36], bug traces [29], patch-related branches [37, 38], and deep learning [2, 39, 40]. Another approach often used in Directed fuzzing focuses on the target bugs detection. The works in [29, 41] aim to detect specific types of vulnerabilities namely, *use-after-free vulnerabilities* by capitalizing the target operation sequences. Other approaches attempt to leverage the fuzzing fitness metrics [6, 36, 41]. Finally, various techniques such as machine learning [34, 40] and symbolic execution [21] are often used to provide effective guidance to the fuzzer.

Human-in-the-loop fuzzing can leverage the use of interactive environments to allow human experts to reason about the fuzzing process and guide it toward area of interest. Consequently, the human expert can provide practical guidance to the fuzzer. Several works have relied on human-in-the-loop mechanisms to *direct* the fuzzer to desirable locations in the programs.

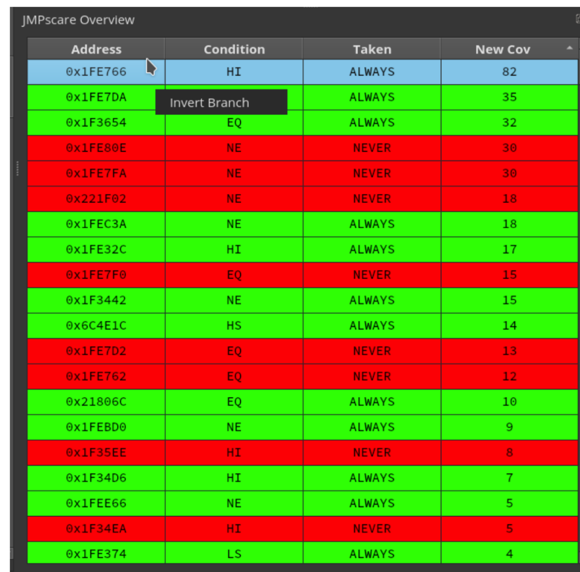
In [42], the authors proposed *Ijon*, a tool that guides AFL-based fuzzers but it depends on having access to manually annotated version of the source code. This allows the fuzzers to solve complex constraints. In turn, solving those challenging constraints makes the fuzzer generate inputs that lead to desirable locations in the program. Although The author's tool can solve some very complex constraints effectively, it requires the human to have a deeper understanding of the program and manually annotate the source code using a specific format. The entire process is error-prone and time-consuming. In addition, the requirement for annotations necessitates the source code to be available. For this reason, black-box fuzzing is not

an option. Finally, the tool has no visualization capabilities, making it difficult for a human to provide feedback *during* the fuzzing process.

Conventional coverage reports can be overwhelming. To address this issue, in [43], the authors proposed a compartment analysis approach that uses a weighted list to help the human expert prioritize which portion of code to focus their efforts on. The authors define a *compartment* as a large portion of code that has not been covered during the fuzzing process. Thus, the main goal of the proposed approach is to help the human expert discover inputs that can reach the most important compartments defined by a ranking system. The ranking of compartments is based on the profiling coverage information derived through static analysis of the interprocedural control-flow graph (ICFG). The LLVM DataFlowSanitizer (DFSan), a dynamic data analysis tool, is used to label each compartment using the input label data and the harness label data. The authors implemented their approach with Data Flow Analysis and DFSan to provide compartment analysis using the LLVM library of the clang compiler on top of the fuzzer AFL++. Moreover, they added a fork-server (cov_fuzzer) to AFL++ to provide profiling information such as Function ranking, Block Weight, Calls Weight, and Conditional as a compartment list. The authors' approach can help the human discover test cases that can cover code that typical coverage-guided fuzzers like AFL++ might not be able to reach due to the complexity of the program. The experimental results show that compartment analysis can contribute to a 94% coverage improvement over AFL++. However, this approach requires the human expert to *handcraft* the test cases based on the compartment list provided by the tool.

Most modern fuzzers require the human to understand a target program well enough to harness it, i.e., a setup to enable the fuzz target to utilize the generated or user-provided test cases. However, complex and large programs are often challenging for a human to analyze. Thus, the authors in [44] developed an introspection toolkit, namely JMPscare, to analyze the test cases' queues of fuzzers. This analysis aims to provide insight into fuzzing test case queues by highlighting unreachable basic blocks from all test cases during fuzzing. The branches that were never taken during fuzzing (so-called frontiers) are used during the analysis. In addition, the authors provide an integration plugin to the binary analysis tool *Binary Ninja* [45] to enable the forced execution through binary patching of the target to traverse the frontiers. The forced execution is achieved by altering the target's control flow with a patch or instrumented

branches to bypass the frontier checks. Moreover, JMPscare can work with most target binaries for black-box fuzzing in AFL++. Also, the toolkit is implemented as a Python and Rust library with support for fuzzers such as unicornafl [46], BaseSAFE [47], and qiling [48] harnesses. In addition, JMPscare provides a Binary Ninja plugin to enable the visualization of the covered and uncovered basic blocks with their conditions discovered during the fuzzing campaign. The proposed tool uses Potential New Coverage (PNC) analysis to effectively highlight branches to basic blocks never taken to the human expert. PNC analysis assigns scores to frontier branches, with a higher score indicating a potentially interesting branch. However, this approach is susceptible to *False positives* to forced execution using binary patching.



Address	Condition	Taken	New Cov
0x1FE766	HI	ALWAYS	82
0x1FE7DA		ALWAYS	35
0x1F3654	EQ	ALWAYS	32
0x1FE80E	NE	NEVER	30
0x1FE7FA	NE	NEVER	30
0x221F02	NE	NEVER	18
0x1FEC3A	NE	ALWAYS	18
0x1FE32C	HI	ALWAYS	17
0x1FE7F0	EQ	NEVER	15
0x1F3442	NE	ALWAYS	15
0x6C4E1C	HS	ALWAYS	14
0x1FE7D2	EQ	NEVER	13
0x1FE762	EQ	NEVER	12
0x21806C	EQ	ALWAYS	10
0x1FEBD0	NE	ALWAYS	9
0x1F35EE	HI	NEVER	8
0x1F34D6	HI	ALWAYS	7
0x1FEE66	NE	ALWAYS	5
0x1F34EA	HI	NEVER	5
0x1FE374	LS	ALWAYS	4

Figure 8.7: JMPscare user interface in Binary Ninja

8.4 Symbolic Execution in Human-in-the-Loop Fuzzing

Symbolic execution is a software testing technique that aims to explore different paths of a program by symbolically executing the target program while collecting path constraints [49]. The paths' constraints can be solved with SMT or SAT solver to generate inputs that traverse those paths. Symbolic execution is particularly useful in identifying bugs and vulnerabilities in programs without the need to have knowledge concrete input values. However, symbolic

execution suffers from path explosion [9, 49, 50], i.e., the number of symbolic states increases exponentially with the length of the branching paths. Due to the success of fuzzing in automatically generating inputs to discover bugs, several works [6, 9, 35] have considered a hybrid approach that can leverage the fuzzing and symbolic execution paradigms to discover bugs effectively.

Traditional approaches in Directed fuzzing were based on symbolic execution [4, 18]. However, these automated testing approaches often fail to perform as well as human experts for programs containing complex logic. To address this issue, the authors proposed Human-assisted Cyber Reasoning System (HaCRS) [50], a human-assisted autonomous vulnerability analysis tool. HaCRS interface uses a text-based terminal interface, coined as Human Automation Link (HAL), with the following components: *Program description*, *Interaction terminal*, *Tasklet goal and feedback*, *Example interactions*, and *CRS-Generated suggestions*. The program description outlines details regarding the execution of the program, while the Interaction terminal provides the user with an interface to interact with the automation tool. The Tasklet instructions show the instruction presented to the human assistant, while the Example interactions display the previous interactions with the program. The *CRS-Generated Suggestions* display helpful information to generate alternative test cases during the *Tasklet goal and feedback* provides feedback on the uncovered paths CFG and string outputs along those paths. The HAL text-based terminal interface allows the human assistant to provide inputs that can reach previously unreachable code by test cases in the fuzzer test case queues. Next, the human-generated test cases are mutated by the exploration components of HaCRS. The human can monitor and modify the test cases generated by all other human assistants to increase code coverage. Finally, the automation tool will mutate the test cases and prompt the human to review them. HaCRS can automatically discover the input format of programs to inform the automation tool or the human expert during test case generation. In addition, HaCRS can automatically analyze the CFG of binaries to identify and categorize the string references contained in them HaCRS can leverage its automation tool to generate alternative symbolic tokens (inputs) to give suggestions to human experts during test case generation. Although HaCRS suggestions can be helpful to human experts, it requires them to understand and utilize that information to generate suitable test cases which often be challenging.

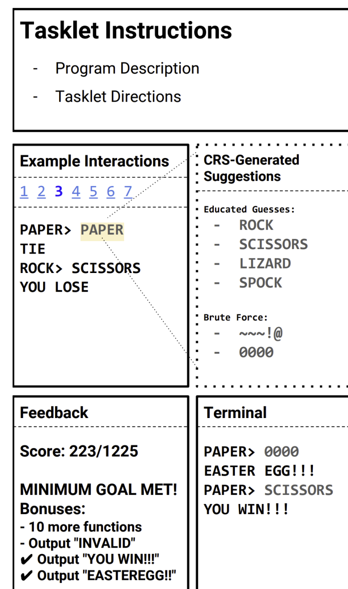


Figure 8.8: HaCRS user interface diagram

Table 8.1: Summary of related works in Human-in-the-loop fuzzing

Work	Goal	Techniques	Advantages	Limitations	Category	Human Role(s)
Ijon [42]	Guide AFL-based fuzzers using Human annotations	Source code annotations	Can solve complex constraints and generate structured inputs	Intrusive and manual code annotation	Directed Fuzzing	Supervisor, Collaborator
Homo in Machina [43]	Increase coverage by providing compartments list to help humans find inputs which can reach them.	Compartment analysis	Higher code coverage for complex programs	Requires handcrafting seeds which cover the compartment list	Directed Fuzzing, Visualization	Assistant, Supervisor

Table 8.1 continued from previous page

JMPscare [44]	Provide insight into fuzzing test case queues by highlighting uncovered basic blocks across all queue items during fuzzing.	Potential New Coverage (PNC) analysis	Can work with binaries and can cover never taken branches (frontiers) to guide the human expert towards interesting frontiers.	Can have false positives due to forced execution using binary patching.	Directed Fuzzing, Visualization	Supervisor
HaCRS [50]	Provide recommendations to human assistants during test case generation	Human-based Computation & Can discover the input format of programs to help human experts during test case generation	Requires to effectively utilize the provided suggestions to handcraft good test cases	Symbolic execution, Visualization	Supervisor, Collaborator	

Table 8.1 continued from previous page

FuzzExplore [24]	Provide insights into the fuzzing process of a particular target with the help of a visual feedback component.	Plots of various metrics for coverage and test cases generation	Provides visual information,about test case similarity and coverage evolution	Requires an understanding visual elements provided	Visualization	Supervisor, Collaborator
VisFuzz [27]	Provide visual information to help human experts understand and improve the overall efficiency of the fuzzing process	Real-time visualization of CFG Call Graph, and fuzzing statistics	Provides a web interface that can be on heterogeneous devices	Expects an effectively utilize the visualization information to handcraft test cases	Visualization	Supervisor, Collaborator

Chapter 9

Conclusion & Future Work

The use of handcrafting test cases in fuzzing poses several issues which can impede their adoption. To address those issues, we propose a seed generation and visualization framework that leverages human-in-the-loop directed fuzzing. Our approach uses symbolic execution with a path pruning technique based on the Depth-First-Search and Dijstra's shortest path algorithms. Our experimental results show a significant improvement over the AFL fuzzing process.

Several challenges still exist which are not addressed by our current approach. First, the user-provided target might not be present in the CFG of the binary. We address this issue with a symbolic execution in a depth-first approach until an exit node is found in the CFG. However, future work should address this issue with techniques that can leverage similarity metrics between the missed basic block targets and existing basic blocks in the CFG. Additionally, our seed visualization tool outputs its results in a DOT and PNG image file. Future work could make use of the graph visualization tool to enable visualization in heterogeneous devices and platforms such as computer browsers, Virtual Reality headsets, etc.

The Cyber Grand Challenge [51, 52, 53] was a vulnerability discovery and defense competition organized by DARPA (Defense Advanced Research Projects Agency). Over 100 US entities participated in the competition, which was held from 2014 to 2016. During the competition, several teams built binary analysis tools such as BAP [54] and angr, and fuzzers such as Driller and Mayhem during the competition. These tools are still used for automated software testing and analysis. The challenge datasets for the qualifying event is a set of 131 source codes and binaries written in C/C++ intended to run in custom and open source Linux OS

named DECREE (DARPA Experimental Cyber Research Evaluation Environment). Though it is possible to compile those binaries for different OS [55]. The challenge sets source code contains 50 Common Weakness Enumeration (CWE) vulnerabilities such as buffer overflow, integer overflows, and use after free. Our future work will use the challenge sets to perform a more in-depth evaluation of our framework.

Bibliography

[1] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[2] X. Zhu, S. Liu, X. Li, S. Wen, J. Zhang, J. Zhang, S. Camtepe, Camtepe Seyit, and Y. Xiang, "DeFuzz: Deep Learning Guided Directed Fuzzing." *arXiv: Cryptography and Security*, 2020.

[3] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of The ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990.

[4] V. Ganesh, T. Leek, and M. Rinard, "taint based directed whitebox fuzzing," *2009 IEEE 31st International Conference on Software Engineering*, 2009.

[5] "american fuzzy lop." [Online]. Available: https://lcamtuf.coredump.cx/afl/

[6] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed Greybox Fuzzing," *CCS*, pp. 2329–2344, Oct. 2017.

[7] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[8] I. Yun, S. Lee, Meng Xu, Meng Xu, Meng Xu, M. Xu, Y. Jang, and T. Kim, "QSYM: a practical concolic execution engine tailored for hybrid fuzzing," *USENIX Security Symposium*, pp. 745–761, Aug. 2018.

[9] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution." *Network and Distributed System Security Symposium*, Jan. 2016.

[10] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 711–725.

[11] C. Holler, K. Herzig, A. Zeller *et al.*, "Fuzzing with code fragments." in *USENIX Security Symposium*, 2012, pp. 445–458.

[12] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing." in *NDSS*, 2018.

[13] S. Katoch, S. S. Chauhan, V. Kumar, and V. Kumar, "A review on genetic algorithm: past, present, and future," *Multimedia Tools and Applications*, vol. 80, no. 5, pp. 1–36, 2020.

[14] Qian Yan, Minhuan Huang, and Huayang Cao, "A Survey of Human-machine Collaboration in Fuzzing," *International Conference on Data Science in Cyberspace*, 2022.

[15] D. Andriesse, *Practical binary analysis: build your own Linux tools for binary instrumentation, analysis, and disassembly.* no starch press, 2018.

[16] L. de Moura and N. Bjørner, "Z3: an efficient SMT solver," *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pp. 337–340, Mar. 2008.

[17] Y. Liu, X. Zhou, and W.-W. Gong, "A Survey of Search Strategies in the Dynamic Symbolic Execution," vol. 12, p. 03025, Jan. 2017.

[18] S. Person, G. Yang, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*, vol. 46, no. 6, pp. 504–515, Jun. 2011.

[19] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," *Sensors Applications Symposium*, pp. 95–111, Sep. 2011.

[20] A. Y. Gerasimov, "Directed Dynamic Symbolic Execution for Static Analysis Warnings Confirmation," *Programming and Computer Software*, vol. 44, no. 5, pp. 316–323, Sep. 2018.

[21] J. Kim and J. Yun, "Poster: Directed Hybrid Fuzzing on Binary Code," *Conference on Computer and Communications Security*, pp. 2637–2639, Nov. 2019.

[22] V. Chipounov, V. Kuznetsov, and G. Candea, "The S2E Platform: Design, Implementation, and Applications," *ACM Transactions on Computer Systems*, vol. 30, no. 1, p. 2, Feb. 2012.

[23] L. Borzacchiello, E. Coppa, and C. Demetrescu, "FUZZOLIC: Mixing fuzzing and concolic execution," *Computers & Security*, vol. 108, p. 102368, Jun. 2021.

[24] Andrea Fioraldi and Luigi Paolo Pileggi, "FuzzSplore: Visualizing Feedback-Driven Fuzzing Techniques," *ArXiv*, 2021.

[25] M. Grishin and I. Korkin, "Human-Controlled Fuzzing With AFL," 2022.

[26] Aftab Hussain and Mohammad Amin Alipour, "FMViz: Visualizing Tests Generated by AFL at the Byte-level," *ArXiv*, 2021.

[27] C. Zhou, M. Wang, J. Liang, Zhe Liu, Zhe Liu, Z. Liu, Chengnian Sun, C. Sun, Chengnian Sun, and Y. Jiang, "VisFuzz: understanding and intervening fuzzing with interactive visualization," *International Conference on Automated Software Engineering*, pp. 1078–1081, Nov. 2019.

[28] Z. Chen, S. Guo, and D. Fu, "A Directed Fuzzing Based on the Dynamic Symbolic Execution and Extended Program Behavior Model," *2012 Second International Conference on Instrumentation, Measurement, Computer, Communication and Control*, pp. 1641–1644, Dec. 2012.

[29] M.-D. Nguyen, Manh-Dung Nguyen, Manh-Dung Nguyen, Sébastien Bardin, S. Bardin, Sébastien Bardin, R. Bonichon, Roland Groz, R. Groz, and M. Lemerre, "Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities," *arXiv: Cryptography and Security*, Feb. 2020.

[30] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, no. 1, pp. 1–13, 2018.

[31] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 50–59.

[32] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "NAUTILUS: Fishing for Deep Bugs with Grammars." *Network and Distributed System Security Symposium*, Feb. 2019.

[33] P. Wang, Pengfei Wang, Pengfei Wang, Xu Zhou, and X. Zhou, "SoK: The Progress, Challenges, and Perspectives of Directed Greybox Fuzzing." *arXiv: Cryptography and Security*, May 2020.

[34] W. You, P. Zong, K. Chen, X. Wang, XiaoFeng Wang, XiaoFeng Wang, X. Liao, P. Bian, and B. Liang, "SemFuzz: Semantics-based Automatic Generation of Proof-of-Concept Exploits," *Conference on Computer and Communications Security*, pp. 2139–2154, Oct. 2017.

[35] M. E. Garbelini, Chundong Wang, C. Wang, and S. Chattopadhyay, "GREYHOUND: Directed Greybox Wi-Fi Fuzzing," *IEEE Transactions on Dependable and Secure Computing*, no. 99, pp. 1–1, Jan. 2021.

[36] Y. Wang, J. Xiangkun, Xiangkun Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, Dinghao Wu, and P. Su, "Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization." *Network and Distributed System Security Symposium*, Jan. 2020.

[37] X. Zhu and M. Böhme, "Regression Greybox Fuzzing," *Conference on Computer and Communications Security*, Nov. 2021.

[38] P. Jiaqi, F. Li, L. Feng, F. Li, B. Liu, L. Xu, L. Binghong, K. Chen, and W. Huo, "1dVul: Discovering 1-Day Vulnerabilities through Binary Patches," *Dependable Systems and Networks*, pp. 605–616, Jun. 2019.

[39] Y. Zhao, Y. Li, Y. Li, Y. Tengfei, H. Xie, Haiyong Xie, and Haiyong Xie, "Suzzer: A Vulnerability-Guided Fuzzer Based on Deep Learning," *Conference on Information Security and Cryptology*, pp. 134–153, Dec. 2019.

[40] Y. Li, S. Ji, C. Lv, Y. Chen, J. Chen, Q. Gu, and C. Wu, "V-Fuzz: Vulnerability-Oriented Evolutionary Fuzzing." *arXiv: Cryptography and Security*, Jan. 2019.

[41] Haijun Wang, H. Wang, X. Xie, Y. Li, C. Wen, Y. Li, Y. Liu, Y. Liu, S. Qin, H. Chen, and Y. Sui, "Typestate-guided fuzzer for discovering use-after-free vulnerabilities," *International Conference on Software Engineering*, pp. 999–1010, Jun. 2020.

[42] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: Exploring Deep State Spaces via Fuzzing," *IEEE Symposium on Security and Privacy*, pp. 1597–1612, May 2020.

[43] Josh Bundt, Andrew Fasano, Brendan Dolan-Gavitt, W. Robertson, and T. Leek, "Homo in Machina: Improving Fuzz Testing Coverage via Compartment Analysis," *ArXiv*, 2022.

[44] D. Maier and L. Seidel, "JMPscare: Introspection for Binary-Only Fuzzing," Jan. 2021.

[45] "Binary Ninja." [Online]. Available: https://binary.ninja/

[46] "UnicornAFL," Apr. 2023, original-date: 2021-11-29T14:43:39Z. [Online]. Available: https://github.com/AFLplusplus/unicornafl

[47] D. Maier, L. Seidel, and S. Park, "Basesafe: Baseband sanitized fuzzing through emulation," in *Proceedings of the 13th ACM conference on security and privacy in wireless and mobile networks*, 2020, pp. 122–132.

[48] "qilingframework/qiling," Apr. 2023, original-date: 2019-08-22T13:22:15Z. [Online]. Available: https://github.com/qilingframework/qiling

[49] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A Survey of Symbolic Execution Techniques," *ACM Computing Surveys*, vol. 51, no. 3, p. 50, May 2018.

[50] Y. Shoshitaishvili, M. Weissbacher, L. Dresel, C. Salls, R. Wang, C. Kruegel, and G. Vigna, "Rise of the HaCRS: Augmenting Autonomous Cyber Reasoning Systems with Human Assistance," *arXiv: Cryptography and Security*, Aug. 2017.

[51] J. Song and J. Alves-Foss, "The DARPA Cyber Grand Challenge: A Competitor's Perspective," *IEEE Security and Privacy*, vol. 13, no. 6, pp. 72–76, Nov. 2015.

[52] ——, "The DARPA Cyber Grand Challenge: A Competitor's Perspective, Part 2," *IEEE Security and Privacy*, vol. 14, no. 1, pp. 76–81, Jan. 2016.

[53] Newton Lee and N. Lee, "DARPA's Cyber Grand Challenge (2014–2016)," pp. 429–456, Jan. 2015.

[54] David Brumley, David Brumley, and D. Brumley, "The Cyber Grand Challenge and the Future of Cyber-Autonomy." *Log in*, vol. 43, Jan. 2018.

[55] S. securitylearner, "Your tool works better than mine? Prove it." Aug. 2016. [Online]. Available: https://blog.trailofbits.com/2016/08/01/your-tool-works-better-than-mine-prove-it/