TU WIEN Informatics

Erzeugung von mehrsprachigen, semantisch äquivalenten Fuzzer-Benchmarks aus Labyrinthen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Logic and Computation

eingereicht von

Jana Chadt, BSc
Matrikelnummer 01528032

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Univ. Prof. Dr. Maria Christakis

Wien, 27. Februar 2024

Jana Chadt	Maria Christakis

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Generating Multi-lingual, Semantically Equivalent Fuzzer-Benchmarks from Mazes

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Logic and Computation

by

Jana Chadt, BSc
Registration Number 01528032

to the Faculty of Informatics

at the TU Wien

Advisor: Univ. Prof. Dr. Maria Christakis

Vienna, February 27, 2024

Jana Chadt	Maria Christakis

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Jana Chadt, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 27. Februar 2024

Jana Chadt

Danksagung

In erster Linie will ich mich bei meiner Betreuerin, Univ. Prof. Dr. Maria Christakis für ihre Zeit und Unterstützung, als auch für ihren wertvollen Rat während dieser Arbeit, bedanken. Zusätzlich will ich mich bei Valentin Wüstholz bedanken, der mich bei Fragen mit Rat und Diskussion unterstützt hat. Besonderer Dank geht auch an Hannes Siebenhandl, Samuel Pilz und Christoph Hochrainer, für zahllose Diskussionen, konstruktive Kritik und emotionale Unterstützung. Zu guter Letzt, will ich mich bei meiner Familie für ihre Unterstützung während meines Studiums bedanken, besonders bei meiner Oma, die nie aufgegeben hat, die Bedeutung von 'Logic and Computation' verstehen zu versuchen.

Acknowledgements

First and foremost, I am deeply grateful to my supervisor, Univ. Prof. Dr. Maria Christakis for her time and support, as well as her valuable advice during my work on this thesis. I also want to thank Valentin Wüstholz for his thoughtful advice and for repeatedly taking the time to answer my questions. Special thanks go to Hannes Siebenhandl, Samuel Pilz and Christoph Hochrainer for countless discussions, constructive criticism and emotional support. Last but not least, I want to thank my family for their support throughout my studies, especially my grandmother for never giving up in trying to understand what logic and computation actually means.

Kurzfassung

Fuzzers sind Werkzeuge zur automatisierten Durchführung von Softwaretests, die unerwartetes oder falsches Verhalten von Software aufdecken, indem diese wiederholt mit verschiedenen Inputs ausgeführt wird. Die Entwicklung von Fuzzern ist ein reger Forschungsbereich und neue Fuzzer und Fuzzingmethoden werden laufend eingeführt; es ist daher wichtig, Stärken und Schwächen verschiedener Fuzzer evaluieren zu können. Es gibt bereits viele verschiedene Fuzzer-Benchmarking Projekte, allerdings vergleichen diese immer Fuzzer einer bestimmten Programmiersprache, was es schwierig macht, Fuzzer verschiedener Sprachen vergleichen zu können. Wir führen Muzzle ein, ein Fuzzer-Benchmarking Werkzeug, das fehlerhafte, semantisch äquivalente `C` und `Solidity` Programme generiert. Muzzle evaluiert Fuzzer beider Sprachen mithilfe von Benchmarks aus Programmen, die es generiert. Wir verwenden Muzzle, um einen solchen Benchmark aus Programmen zu generieren und vergleichen damit verschiedene `Solidity` und `C` Fuzzer basierend auf ihren Fähigkeiten und ihre Geschwindigkeit, Fehler in den Programmen zu finden. Unsere Ergebnisse zeigen, dass `C` Fuzzer wesentlich besser als `Solidity` Fuzzer abschneiden, allerdings gibt es auch innerhalb der `Solidity` Fuzzer starke Unterschiede.

Abstract

Fuzzers are testing tools which discover unexpected or incorrect behaviour in software applications by repeatedly executing software with different inputs. Development of fuzzers is an active research area and new fuzzers and fuzzing techniques are introduced constantly, therefore it is important to evaluate strengths and weaknesses of different fuzzers. Various fuzzer benchmarking projects already exist, but existing tools only compare fuzzers of a specific programming language, making it hard to tell how fuzzers of different languages compare. We introduce Muzzle, a fuzzer benchmarking tool which generates buggy, semantically equivalent `C` and `Solidity` programs. Muzzle evaluates fuzzers of both languages using the programs it generates as a benchmark. We generate a benchmark of programs using Muzzle and evaluate different `Solidity` and `C` fuzzers based on their ability and speed in finding bugs. Our results show that `C` fuzzers one-sidedly outperform `Solidity` ones, yet even between the evaluated `Solidity` fuzzers there are stark differences.

Contents

CHAPTER 1

Introduction

With increasing complexity and size of software applications, the demands on software quality assurance have increased in turn. Fuzzing [Micheal et al., 2007] is a method of increasing trust in the correctness of software. In particular, it is the process of finding bugs and vulnerabilities by repeatedly testing programs with *fuzzed* or modified inputs.

The effectiveness of fuzzing software or *fuzzers* can be compared by executing fuzzers on different programs and comparing their ability to find bugs in these programs. The amount of found bugs or how fast bugs are found can then be used as the performance evaluation factor.

There already exist various projects with the goal of benchmarking different fuzzers, which differ by the types of programs that are supplied to the fuzzers. Some use real-world programs that are known to have vulnerabilities, such as the `C` fuzzer benchmarking tools: Fuzzbench Metzman et al. [2021], Magma Hazimeh et al. [2021] and UNIFUZZ Li et al. [2020]. Others, such as FIXREVERTER Zhang et al. [2022], take existing real-world programs and inject bugs into these programs. There also exist benchmarking projects that completely synthesise the input programs, such as HyperPut Felici et al. [2022] which generates `C` input programs using bugs as seeds or Daedaluzz Wüstholz [2023] which generates `Solidity` input programs from mazes.

Being able to synthesise buggy programs is useful for testing and comparing existing fuzzers as this allows the generation of a varied and fair input program set. Currently, there exist fuzzer benchmarking projects which produce buggy code of specific programming languages in order to compare fuzzers of that language but, to the best of our knowledge, there exists no tool, able to compare fuzzers of different programming languages.

Fuzzle [Lee et al., 2022] is a `C` programming language fuzzer benchmarking tool which uses the `Python` library mazelib [maz, 2014] as a starting point for program generation. It generates code based on mazes constructed by mazelib, where a goal specified within the maze represents a bug in the program and the start of the maze represents the

program's entrypoint. The maze's cells are translated to functions, where function a can be entered from function b in the generated program if cell a is a neighbour of cell b in the corresponding maze. Intuitively, a fuzzer trying to find a bug in the generated program then represents an agent trying to find the goal while traversing the maze.

Smart contracts are programs deployed on the blockchain and often represent financially critical systems. It is therefore important to have ways of gaining trust in smart contract software. `Solidity` is a popular language for writing smart contracts deployed on the Ethereum Wood et al. [2014] blockchain. Fuzzing tools for `Solidity` have been introduced in recent years, but they are young compared to long existing `C` fuzzers. It is therefore hard to know how good `Solidity` fuzzers are at finding vulnerabilities.

In the course of this work, we introduce Muzzle, an extension to the existing Fuzzle project, able to generate buggy `C` programs as well as semantically equivalent buggy `Solidity` programs based on mazes supplied by mazelib. This will allow cross-language comparison and evaluation of fuzzers. We then use input programs generated with Muzzle to compare two `C` fuzzers and three `Solidity` fuzzers based on the amount of bugs found by each fuzzer and the time taken to find these bugs.

Currently, Fuzzle benchmarks AFL [Zalewski, 2019], AFLGo [Böhme et al., 2017], AFL++ [Fioraldi et al., 2020], Eclipser [Choi et al., 2019] and Fuzzolic [Borzacchiello et al., 2021a,b]. We plan to add the `Solidity` fuzzers Echidna [Grieco et al., 2020], ItyFuzz [Shou et al., 2023] and Foundry [fou, 2022] and compare them to the `C` fuzzers AFL++ Fioraldi et al. [2020] and libfuzzer [Project, 2018].

The developed extension will be published as open source software in support of open science.

In Chapter 2 we introduce the benchmarked fuzzers, discuss different fuzzer benchmarking projects and research into evaluation of fuzzer benchmarking. In Chapter 3 we explain relevant concepts for reading this thesis, such as the `Solidity` programming language and fuzzing, going into detail on commonly used fuzzing techniques. We also provide a description of Fuzzle, introducing our terminology when talking about mazes and describing Fuzzle's program generation process. Chapter 4 discusses Muzzle, our contribution, first providing some motivation and then describing the program generation and fuzzer benchmarking implementation of Muzzle. We then describe technical challenges encountered during Muzzle's development. Finally, we explain how we constructed our benchmark set and discuss the results in Chapter 5.

CHAPTER 2

State of the Art

In this chapter, we will introduce the fuzzers, we evaluate using our Muzzle benchmarking tool. We will first introduce the `C` fuzzers, then the `Solidity` fuzzers. Then, we will move on to discuss existing fuzzer benchmarking libraries and finally proceed to research on evaluation of different fuzzer benchmarking tools.

2.1 `C` fuzzers

In the following section, we introduce all `C` fuzzers that are evaluated using the Muzzle benchmarking tool.

2.1.1 The AFL family fuzzers

American Fuzzy Lop, or AFL for short, is a brute-force, grey-box fuzzer, developed by Zalewski which combines compile-time instrumentation with genetic algorithms, in order to discover test cases which cover a high variety of internal program states and can thus help in increasing a project's code coverage. The fuzzer is open source and can be found on GitHub AFL [2013], though it is read-only and has not been maintained since 2020.

AFL is succeeded by AFL++ [Fioraldi et al., 2020] which has improved performance and additional compatibility features compared to its predecessor, it is also open source software and available on GitHub AFL [2019].

While AFL and AFL++ utilise a brute-force approach, AFLGo, as introduced by Böhme et al., is an extension to AFL which targets specific locations in the code and aims to generate inputs in order for these targets to be reached while not wasting resources on irrelevant parts of the program.

2.1.2 LibFuzzer

LibFuzzer [Project, 2018] is an in-process C-fuzzing engine which is integrated into LLVM. It is not part of the group of C fuzzers tested in Fuzzle, but we include it into Muzzle due to its popularity. This coverage-guided, genetic fuzzer provides the option to supply corpus data containing sample inputs, which the fuzzer then mutates. This can be useful when the tested program expects complex input structures. LibFuzzer is shipped with clang but has been put on maintenance mode in 2022 and is therefore no longer actively being developed.

2.2 Solidity fuzzers

In the following section, we introduce the Solidity fuzzers that are evaluated using Muzzle.

2.2.1 Echidna

Echidna [Grieco et al., 2020] can be used for property testing of Solidity programs by allowing the specification of assertions in the code. The fuzzer then reports any assertions it falsifies during its run.

2.2.2 Foundry

Foundry [fou, 2022] is a toolkit for Ethereum Wood et al. [2014], which provides a testing framework called *forge* that natively supports fuzzing of specified functions in the test-suite. Forge's fuzzing functionality is applied to a project's test-suite, where each unit test taking at least one parameter is treated as a property-based test.

2.2.3 ItyFuzz

ItyFuzz Shou et al. [2023] is a snapshot-based fuzzer, meaning it stores snapshots of states and singleton transactions instead of sequences of transactions. This allows exploration of the code by starting from different stored states, instead of having to re-execute inputs in order to reach these states. To deal with the issue of runtime memory introduced by storing too many snapshots, Shou et al. design a feedback mechanism to classify more interesting snapshots. ItyFuzz is the first smart-contract fuzzer to support on-chain testing, i.e. testing a contract deployed on the blockchain, as well as testing in local deployment.

2.3 Fuzzer benchmarking libraries

There exist many different benchmarking libraries for fuzzers of the C programming language.

One of the most well-known C benchmarking tools is Google's FuzzBench [Metzman et al., 2021] which provides a service that evaluates fuzzers using real-world projects. It consists of an API which allows users to integrate different fuzzers to be measured by running them on oss-fuzz [oss, 2016] projects. These projects are real-world open-source codebases and the fuzzers applied to can then be compared based on code coverage or found bugs. The results of the benchmark are provided by a reporting library in graph and table format.

Magma is a ground-truth fuzzing benchmark developed by Hazimeh et al. which provides real world programs with known bugs as fuzzer input. These real-world programs are selected projects from the aforementioned oss-fuzz project, that are still actively developed. After cloning the Magma repository, a user can manage fuzzing campaigns using Magma's *captain* scripts. The results of each measured fuzzer can then be read from a JSON file, showing how often each bug was reached and triggered respectively.

UNIFUZZ [Li et al., 2020] provides an open-source platform which includes 20 real-world programs and measures 35 different C fuzzers. They measure the fuzzers' respective performance based on six metrics, including quantity and quality of found bugs, speed and stability of finding these bugs, as well as coverage and the resource cost of the fuzzers. Li et al. observe that no single fuzzer outperforms the others in all of these metrics.

Zhang et al. created a benchmark set, called FIXREVERTER, which consists of 10 input C programs that contain over 8000 bugs in total. These input programs were created by using bugfix patterns to inject bugs into existing programs, taken from Google's fuzzbench as well as the GNU Binutils.

Similarly to FIXREVERTER, HyperPUT by Felici et al. also uses bugs as the baseline to create benchmarks for C fuzzers. But instead of injecting bugs into existing programs, HyperPUT takes a so called 'seed' bug and then creates an input program by adding program structures to this bug until the program is of a sufficient size. Since these PUTs or ProgramUnderTests, can be configured to have certain properties, this method allows fuzzers to be tested according to their suitability for finding bugs in programs with these properties.

Fuzzle, developed by Lee et al. in 2022, is a fuzzer benchmarking tool which generates buggy C programs and uses them to evaluate C fuzzers. The input programs are constructed by generating mazes using the Python library, mazelib maz [2014], and translating these mazes to C code. The fuzzer acts as an agent in the maze, trying to find the maze's exit, which is translated to a program crash in the code. Thus, the agent finding the exit of the maze is equivalent to the fuzzer finding an execution path to the bug in the translated program. Lee et al. ensure realistic benchmarks by using CVEs in SMT-LIB format to generate additional path constraints for the test-suite. Fuzzle is used to evaluate five state-of-the-art fuzzers: AFL, AFLGo, AFL++, Eclipser and Fuzzolic.

Recently, there has also been development in benchmarking tools for smart-contract fuzzers. In 2023, Daedaluzz [Wüstholz, 2023] was introduced, which uses mazes containing multiple bugs to generate input programs for Solidity fuzzers.

2.4 Research in fuzzer benchmark evaluation

In this section, we will discuss research in the area of fuzzer benchmark evaluation, i.e. how well suited certain types of programs are as input programs in order to evaluate the effectiveness of fuzzers.

Bundt et al. examine the value of synthetic bug injection for benchmarking fuzzers by comparing eight different fuzzers on 20 targets from different sources. These targets contain real-world bugs as reported in CVEs, and a significant result of the evaluation is that none of the measured fuzzers were able to detect any of these 50 real-world bugs. The authors conclude that synthetic bugs remain significantly easier to discover by fuzzers compared to organically occurring ones, but still provide significant insight into different fuzzers' strengths and weaknesses. They go on to analyse flaws in current bug injection techniques and suggest improvements.

Most fuzzer benchmarking tools look into a fuzzer's ability to find specific points in the programs where synthetic or real-world bugs are located. Böhme et al. examine the correlation between the covered code and the number of bugs found by the fuzzer. They evaluate ten fuzzers on 24 input programs and conclude that there is indeed a strong correlation between code coverage and the amount of found bugs but only a moderate level of agreement. They also highlight the difference of a fuzzer reaching a fault in the code in terms of code-coverage reaching the respective line and the fuzzer actually being able to expose or trigger the bug.

CHAPTER 3

Background

In this chapter, we will introduce background information on concepts used throughout this thesis. First, we will introduce `Solidity`, the language we generate programs in. We will then move on to the concept of fuzzing while discussing different techniques commonly applied by fuzzers. Lastly, we will introduce Fuzzle, the project we extend to implement Muzzle.

3.1 `Solidity`

Smart contracts are programs deployed on the blockchain that have a balance of tokens and expose a public interface, allowing them to accept transactions from any party. These transactions are calls to the contract's public functions or token transfers to the deployed contract, which can be triggered by any party. Each transaction executes a function which can change the smart contract's state and is irreversible once completed.

`Solidity` is one of the most popular languages for writing smart contracts. It is a high-level object-oriented language, first proposed in 2014 by Wood et al.. The language was designed to write smart contracts to be deployed on the Ethereum blockchain, but smart contracts written in `Solidity` can also be deployed on other blockchains. In order to run `Solidity` programs on the Ethereum blockchain, these programs first have to be compiled to *Ethereum Virtual Machine* byte code.

3.2 Fuzzing

Fuzzing is an automated software testing approach with the goal of generating inputs which expose unwanted behaviour in the software. Tools which use this approach are called *fuzzers*. Fuzzers repeatedly execute a program under test with different inputs to

find crashes, memory leaks or other behaviour deviating from the specification of the program.

In the simplest sense these different inputs can be seen as random, but most fuzzers use more sophisticated techniques. These techniques are often aimed at achieving the highest possible code coverage, since this allows finding bugs in as many places as possible.

3.2.1 Genetic algorithms

One of the techniques to maximise code coverage is applying genetic algorithms during the fuzzing process. Genetic algorithms represent algorithms where the solution space undergoes some selection using a so-called fitness function during runtime. The possible solutions selected by this fitness function are then mutated in some way and used as the solution space in the next iteration. Mutations are predefined alterations to elements of the solution space, such as random bit-flips.

In fuzzing, instead of randomly generating new inputs in each iteration, any fuzzed inputs which discover new paths or branches in the tested program are selected for mutation. The selected inputs are then mutated and added to the queue of inputs to be executed in the next fuzzing run.

3.2.2 White-, grey- and black-box fuzzing

Fuzzers can be categorised into *white-*, *grey-* or *black-box*. This term refers to how aware the tools are of the tested program's structure, where black-box fuzzers are entirely unaware of the program's structure, treating it like a black-box. White-box fuzzers on the other hand, analyse the tested programs to be able to generate inputs in a more targeted way, often trying to maximise code coverage. Black-box fuzzing is often unable to discover more complex bugs while white-box fuzzing often carries a significant performance overhead. Grey-box fuzzing is a method with less performance overhead aimed at being able to discover more complex bugs than black-box fuzzing using instrumentation with a low overhead rather than program analysis tools like symbolic execution used in white-box fuzzing, which provides information to guide the fuzzer into finding better program inputs.

3.2.3 Symbolic execution

Symbolic execution is a program analysis technique where a program execution is abstract, where instead of inputs, it is executed with input classes. These represent possible inputs to reach a certain point in the program, meaning inputs that lead through a certain execution path in the program. The result of such a symbolic execution is represented by symbolic formulas, which describe inputs satisfying the branch conditions to traverse a certain path of the program.

Concolic testing

Analysing a program with symbolic execution needs a lot of resources and the exploration often remains shallow without entering deeper paths of the program. Concolic testing is a testing technique which applies symbolic execution but replaces the symbolic inputs with concrete ones in order to reduce performance overhead and find these deeper paths.

3.3 Fuzzle

Fuzzle is a fuzzer benchmarking tool developed by Lee et al., which evaluates different `C` fuzzers by generating buggy programs from mazes as benchmark inputs.

3.3.1 Maze terminology

We define a maze to be a two-dimensional matrix of either *cells* or *walls*. Two cells in the maze are *neighbours* if they are adjacent, note that diagonal cells are not adjacent. In our case there is exactly one cell which is the *entry point* of the maze and exactly one cell which is the *goal* of the maze. An agent traversing the maze starts at the entry point and tries to find a *path* to the goal, where it can only move from one cell to another if it is a neighbour of their current cell. Let the agent's position be at some cell in the maze, then we say there is a branching point in the maze if there exist two different cells the agent can travel to from its current one.

3.3.2 Program generation

This method of generating input programs for fuzzers stems from the observation that finding a bug in any given software resembles solving a maze. Intuitively, we can see a resemblance between branching paths in a maze and branches in a program and, similarly, maze cells as program instructions. The authors utilise the `Python` library mazelib maz [2014] which provides functionality for generating mazes, using different types of generation algorithms. The library also provides capability for solving mazes, which Lee et al. utilise to be able to provide a ground truth for the input program they generate from said mazes. Additionally, the library can be used to generate an image from the maze in order to provide a visualisation correlating with the possible paths through the input program. We provide an example for such a visualisation in Figure 3.1 where the entry point of the program is marked by a green arrow and the goal is marked by a red bug.

The mazelib-generated maze is translated into a `C`-program such that for each cell in the maze, a function is generated. The functions have an identifying suffix using the maze cell's indices, where the top-left cell is labelled zero and the bottom-left cell is labelled n, where $n = w \times h$ with w, h being the width and height of the maze respectively. A function b can be entered from another function a in the input program if and only if the corresponding cell b is a connected neighbour of cell a in the maze. A simplified example of functions generated corresponding to the highlighted cells in the maze visualised in

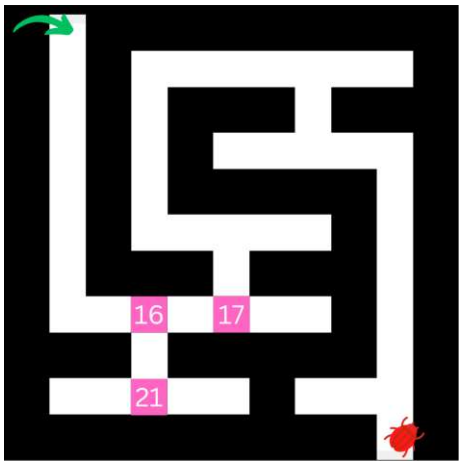

Figure 3.1: 5x5 maze.

```
1  void func_16(signed char *input){
2      if ([          ]) {
3          func_21(input);
4      }
5      else if ([          ]) {
6          func_17(input);
7      }
8  }
9
10 void func_17(signed char *input){
11     if ([          ]) {
12         func_12(input);
13     }
14     else if ([          ]) {
15         func_18(input);
16     }
17 }
18
19 void func_21(signed char *input){
20     if ([          ]) {
21         func_20(input);
22     }
23     else if ([          ]) {
24         func_22(input);
25     }
26 }
```

Figure 3.2: `C` code example of translated functions.

Figure 3.1 can be seen in Figure 3.2. The branching point at cell 16 is represented by the conditionals of `func_16` seen in Line 1. Depending on the input, either `func_17` or `func_21` is entered next. The yellow boxes represent placeholders for some conditionals, which are randomly filled in with predefined formulas during the generation process.

The function corresponding to the maze's exit is the only function in the generated program which calls the function containing the bug/vulnerability to be detected by the fuzzer. Mazes used for program generation always have exactly one starting cell and one exit cell, meaning the generated programs have exactly one starting function and one function, able to call the bug function. The starting function is called from the generated program's entry point, which is provided the fuzzed input. The fuzzers benchmarked by Fuzzle receive the fuzzed input via standard input, store it in an array, and pass this array through the cell functions. Each cell function accesses this array at a tracked index, which increases after every read operation, this allows for different input data in each function call. If there are not enough inputs to pass through the whole maze, the fuzzing run is aborted without finding a bug.

Fuzzle always generates mazes where the exit is reachable from the start, i.e. there exists

a path for an agent traversing the cells leading to the maze's exit. By the input programs' construction, there exists a path through the program, where a fuzzer calls the functions corresponding to the cells in the agent's exit path. Similarly, since every generated maze contains a path to the exit, a fuzzer can trigger the bug in the program by traversing the corresponding functions in the program.

3.3.3 Configurability

Fuzzle provides various parameters for fine-tuning the types of programs to be generated. Most of this configurability directly stems from deciding what kind of initial maze is generated. Firstly, as already mentioned, there are different algorithms Fuzzle can use for generating the starting maze, namely Backtracking, Kruskal, Prims, Wilsons or Sidewinder. The shape of the initial maze and thus the shape of the resulting program is impacted by the choice of algorithm. In addition, performance of the maze generation and thus of the program generation as a whole may differ, depending on which maze-generation algorithm is selected. The size of the initial maze is also configurable in Fuzzle, the minimum size being four by five or five by four. This allows the generated `C` programs to be of different sizes.

The resulting program may contain different amounts of cycles, this can be configured in percentages, starting from zero up to 100 percent. These percentages are based on the maximum number of cycles that may be inserted into the maze, where Fuzzle first inserts a connection b to a for each connection from a function a to another function b and then removes these connections based on the parameter. Removing no connections would result in 100 percent cycles, and removing them all would result in zero percent cycles.

Lastly, to be able to provide more realistic buggy programs, i.e. ones that are more similar to real-world ones, Fuzzle may take a CVE file to use as 'inspiration' for generating more realistic program branching conditions. A CVE, which is short for *Common Vulnerability and Exposures*, is a known, publicly disclosed vulnerability. Fuzzle uses symbolic execution to extract path constraints which describe these vulnerabilities, and then uses these constraints to construct additional paths in the generated functions which have to be traversed in order to reach the bug.

3.3.4 Fuzzer benchmarking

Fuzzle provides an automated process of measuring `C` fuzzers on a set of input `C`-programs. This process spawns a docker container for each input program combined with each fuzzer. Fuzzle supports the `C` fuzzers: afl [Zalewski, 2019], aflgo [Böhme et al., 2017],afl++ [Fioraldi et al., 2020], fuzzolic [Borzacchiello et al., 2021a,b] and eclipser [Choi et al., 2019] and provides docker images and container management scripts to benchmark each of them.

CHAPTER 4

Muzzle - Multilingual Fuzzle

In this chapter, we will first motivate the need for Muzzle, then proceed to discuss its implementation. We will move on to describe the technical challenges encountered in the development of Muzzle and finally discuss benchmarking results when using Muzzle to compare `C` and `Solidity` fuzzers.

4.1 Motivation

Blockchain and smart contract technology have seen an increase in popularity in recent years and are often used in financially critical systems. `Solidity` is a language for writing smart contrasts, deployed on the Ethereum Wood et al. [2014] blockchain and has, in turn, seen a rise in popularity. Since smart contracts can have their state altered by calls to their public interface, ensuring trust is especially critical for smart contract languages. Trust in `Solidity` software can be increased using testing tools such as fuzzers, but since `Solidity` is a very young language, introduced in 2015, so are its tools. The `C` programming language and its tools, on the other hand, have been developed for much longer, but to the best of our knowledge, there is no way to compare `C` fuzzers to the comparatively newly developed `Solidity` ones.

We introduce Muzzle, a way to gain insight into `Solidity` fuzzers' performance compared to `C` fuzzers. Muzzle generates semantically equivalent benchmarks for `C`- as well as `Solidity` fuzzers and measures the performance of different fuzzers of both languages.

4.2 Muzzle implementation

We extend Fuzzle to be able to generate semantically equivalent `Solidity` programs in addition to the existing `C` program generation. This will enable automated, fair comparison of `C` and `Solidity` fuzzers using synthetically generated input programs.

4.2.1 Program generation

We implement this extension by starting out from the same initial maze and generating the C-program as Fuzzle does, but we generate a semantically equivalent Solidity program at the same time.

Similarly to the C program generation, we create a function for each cell to construct the Solidity program. A simplified example of a generated Solidity program can be seen in Listing 4.1, the maze consists of the function func_0 as defined in line 6, representing the starting cell, the function func_bug, defined in line 19 and a dead end in the maze which is the cell represented by func_1 in line 16. We consider that Solidity fuzzers do not have a specific entry point in the fuzzed program, but instead call all exposed functions in random order and with random input. Therefore, we only expose one external function, called step, as can be seen in line 23 of Listing 4.1, to be called by the fuzzer which relays the fuzzed input to the inner 'cell' function. Which of the inner functions is called depends on the 'maze's' progression state, this state is tracked in the func_req variable, which stores the integer corresponding to the suffix of the function which is to be entered next. In other words, if we view the fuzzer as an agent traversing the maze, the exposed function tracks the agent's progress through the maze and calls the function corresponding to the agent's current cell accordingly.

Unlike the input provided by the C fuzzers, which is passed through the standard input, Solidity fuzzers directly fuzz the parameters of the external functions they call. Therefore, we define the external function called by the fuzzers to have a single array function parameter. This array is passed to the internal functions and a **require** statement at the start of each inner function ensures that the array contains enough input for the inner function such that each array access is valid, the function body is not executed if this statement is not satisfied. This can be seen in line 7 of Listing 4.1, as in the next line an array access occurs which could trigger an unwanted crash to be detected by the fuzzer if the input array was empty.

We label the 'cell' functions in the same way we do for the C functions, with an identifying suffix according to the corresponding cells and generate the start function func_start and the goal function func_bug which triggers the bug.

In a similar manner as the generated C programs, generated Solidity programs contain a path from the function corresponding to the entry point of the maze to the func_bug function by construction. Thus, there exists a path to trigger the bug in each Solidity program which can be discovered by a fuzzer. In the case of our example, this path would simply be func_0 ⇒ func_bug with the fuzzed input being some array containing an entry greater or equal to zero at index zero.

Muzzle offers the same configurability options for program generation of Solidity and C programs as Fuzzle.

```
1  contract Maze {
2
3      bool public bug = false;
4      int8 func_req = 0;
5
6      function func_0(int8[] memory inp) internal {
7        require(inp.length >= 1);
8          if (inp[0] >= 0) {
9                  func_req = -1;
10         }
11         else if (inp[0] < 0) {
12           func_req = 1;
13       }
14     }
15
16     function func_1(int8[] memory inp) internal {
17     }
18
19     function func_bug(int8[] memory inp) internal {
20       bug = true;
21     }
22
23     function step(int8[] calldata inp) external {
24         if (func_req == 0) {
25             func_0(inp);
26             return;
27         }
28         if (func_req == 1) {
29             func_1(inp);
30             return;
31         }
32         if (func_req == -1) {
33             func_bug(inp);
34             return;
35         }
36     }
37 }
```

Listing 4.1: `Solidity` code generated for simple example maze with entry, goal and one dead-end cell.

4.2.2 Fuzzer benchmarking

Each fuzzer tested by Muzzle has its own docker image, which allows creating isolated containers to execute a fuzzing campaign. The docker image defines two scripts specific to the fuzzer. The first script runs the specific fuzzer on the given input program and times how long it takes the fuzzer to find a bug. The second script interprets the fuzzer's results and stores them in a file with a format shared between all the different fuzzers' containers. A docker container which receives the program input and executes a fuzzing run on that program input is then spawned per campaign per fuzzer. If the fuzzing run has not completed after some allocated time, the command is interrupted. The fuzzing run's results are then collected to the result file, which is copied to a shared, local directory. Each results file contains information about whether the bug was found, the time taken by the fuzzer to find it, and whether the fuzzer crashed unexpectedly for the specific input program and fuzzer.

Muzzle extends the automatic benchmarking process of Fuzzle to be able to support benchmarking of fuzzers of different languages.

In addition to the `C` fuzzers already supported by Fuzzle, Muzzle adds docker images and container management scripts for the `Solidity` fuzzers: Echidna [Grieco et al., 2020], Foundry [fou, 2022] and ItyFuzz [Shou et al., 2023] as well as the popular `C`-fuzzer LibFuzzer [Project, 2018].

4.3 Technical challenges

In this chapter, we will discuss the technical challenges encountered during the development of Muzzle. Firstly, we will discuss the properties required of input programs of the benchmarked `Solidity` fuzzers and how Muzzle generates code according to them. We will then move on to describe how we adapted data types in the generated code to be able to generate semantically equivalent programs in both languages. Lastly, we will discuss how Muzzle generates branch constraints from CVE files for `C` files and how we implemented an equivalent process for `Solidity` code generation.

4.3.1 Fuzzer compatibility

This section describes differences between `C` and `Solidity` fuzzers and the effect this has on how `Solidity` code is generated in Muzzle.

Fuzzed input

The `C` fuzzers benchmarked in Fuzzle provide an input stream via the standard input and call the main function. The main function then reads the fuzzed input and passes it to the function corresponding to the entry point of the maze.

`Solidity` fuzzers, on the other hand, call any external functions of the contract in an arbitrary order and provide fuzzed inputs for any of their parameters. An example of

```
1  contract Maze {
2      bool public bug = false;
3      int8 func_req = 0;
4
5      function func_0(int8[] memory inp) external {
6          require(func_req == 0);
7          require(inp.length >= 1);
8          if (inp[0] >= 0) {
9                  func_req = -1;
10         }
11         else if (inp[0] < 0) {
12           func_req = 1;
13       }
14     }
15
16     function func_1(int8[] memory inp) external {
17         require(func_req == 1);
18     }
19
20     function func_bug(int8[] memory inp) external {
21         require(func_req == -1);
22         bug = true;
23     }
24 }
```

Listing 4.2: Alternative `Solidity` implementation of the maze.

this can be seen in Listing 4.2, here all cell functions are external and can be called by the fuzzer at any time during the fuzzing process. To ensure, that the fuzzer is only able to call neighbouring cells, **`require`** statements as in line 6 prevent a function body from being executed unless `func_req` has the value of the function's identifying suffix. Without this statement, the fuzzer could simply circumvent 'traversing' the maze and call `func_bug` directly. Since each fuzzing call executes a function at random, there may be several calls to `func_1` before `func_0` is called for the first time and any progress in the maze is made. Comparatively, the generated `C` code enforces a strict order on when functions are called. Here, the fuzzers cannot call functions which make no progress through the maze. In order to guarantee fairness when comparing `C`- and `Solidity` fuzzers, we define only one external function such that fuzzers don't spend unnecessary time calling the wrong functions without making any progress in the maze.

We implement this, via the `step` function, as mentioned in Section 4.2 and shown in line 23 of Listing 4.1. This function tracks the progress through the maze using a variable which stores the identifying suffix of the next function to be called and calls that function accordingly, passing along the fuzzed input. This external function takes an array as its sole input parameter, which is fuzzed by the fuzzers and passed along to the internal functions. Each internal function checks that the size of the passed array is sufficient before accessing the input using a **`require`** statement, as can be seen in Line 7.

```
1 function echidna_bug() external returns (bool) {return !bug;}
```

Listing 4.3: Bug property, defined for echidna.

```
1  import "forge-std/Test.sol";
2  contract TestMaze is Test {
3      Maze m;
4      function setUp() external {
5          m = new Maze();
6      }
7      function invariant_no_bug() external {
8          if (m.bug()) { fail(); }
9      }
10 }
```

Listing 4.4: Bug property, defined for foundry.

Bug implementation

Since we want to evaluate the fuzzers based on their ability to find bugs in the code, we need to define something fuzzers recognise as a bug in the code we generate in Muzzle. In the `C` code, generated by Fuzzle, this is a simple `abort();` call in the goal function of the maze.

The `Solidity` fuzzer, Echidna, on the other hand, requires a specific function with a name starting with the prefix `echidna_` which defines some property under test. Echidna then reports any such properties that were falsified during its fuzzing run. ItyFuzz is designed to be compatible with code written for, Echidna and therefore also treats the properties designed for Echidna in the same way. We therefore simply write such a function as shown in Listing 4.3 at the end of the generated `Solidity` code.

Since Foundry's forge is a tool for writing `Solidity` test-suites, its fuzzer is designed to audit a given test-suite. The fuzzer implementation is designed to report any falsified invariants over some contract under test defined in the test-suite contract. The test suite contract then fuzzes the external functions of the contract under test and reports falsified invariants.

To be able to support the differing requirements of these fuzzers, Muzzle generates two different `Solidity` programs. One to be tested by Echidna and ItyFuzz which contains the already mentioned property defined in Listing 4.3 and the other one to be compatible with Foundry. The generated Foundry-compatible code contains a test contract, `TestMaze` as can be seen in Listing 4.4. This contract is defined to test the `Maze` contract. We define an invariant as can be seen in Line 7 such that the fuzzer reports when the goal of the maze is reached and the bug is found. We then ensure that each `Solidity` fuzzer is called with the input program defined as required.

4.3.2 Translation of data types

The C code generated in Fuzzle reads fuzzed inputs from the standard input and stores them in an array. Singular inputs are read from the array and assigned to variables of type **char**.

There is no obvious semantically equivalent data type to translate this to when generating Solidity code, since the **char** data type is not fully defined in the C specification. The **char** data type may be signed or unsigned depending on the platform ISO [2018], therefore neither **int8** nor **uint8** is always the correct choice for a semantically equivalent data type in Solidity. Since we cannot find a correct solution in the Solidity code generation, we fix the input array in the generated C code in Muzzle instead. We change the data type fuzzed inputs are written to **signed char** instead to avoid the ambiguity. The change has also been merged into the original Fuzzle Lee et al. [2022] project. This ensures that the generated C code's behaviour is not platform dependent while allowing us to use the semantically equivalent input data type, **int8** when generating Solidity code at the same time.

4.3.3 CVE translation

As mentioned in Section 3.3, Fuzzle uses *Common Vulnerabilities and Exposures* or CVE files to inject more realistic path constraints into branching conditionals of the generated code. Muzzle is able to use these CVE files to generate semantically equivalent C and Solidity branch constraints.

Some of the CVEs used in Muzzle revolve around integer casting semantics, in particular over- and underflow semantics. We will not go into detail on the syntax of the used SMT files, but it is important to know that in the generated C code, integer casts are translated from *signed* and *zero extensions* of bit vectors in the SMT files. Let us consider the bit vector 1010 with length 4. We can increase the length of this vector to 8 using zero extension. Zero extension adds leading zero bits to reach the required size, resulting in 0000 1010. On the other hand, we can apply signed extension to increase the size of the same 4 bit integer to size 8. Then the bit vector is interpreted as a signed binary integer in two-complement, and the added leading bits depend on the sign of the integer. Leading 0 bits are added if the represented integer is positive, while leading 1 bits are written if the integer is negative. This would result in the bit vector 1111 1010, in our case, since the 1010 vector would be interpreted as the integer -6.

This behaviour can be emulated in C by applying integer casts, which is how Muzzle translates the signed and unsigned extensions when generating C programs. In order to emulate the same zero extension of some variable `x`, we call `(uint8_t)x`, casting the variable to an unsigned integer of bit width 8. Similarly, we emulate the signed extension by calling `(int8_t)x`, casting `x` to a signed integer of the same bit width. Thus, to implement a zero extension of an 8 bit vector in C by 24 bits, we write `(uint32_t)x` and for signed extension, we write `(int32_t)x`.

	int8	**uint8**
signed extend-8	`int16(x)`	`int16(int8(x))`
zero extend-8	`uint16(uint8((x))`	`uint16(x)`
signed extend-24	`int32(x)`	`int32(int8(x))`
zero extend-24	`uint32(uint8(x))`	`uint32(x)`
signed extend-56	`int64(x)`	`int64(int8(x))`
zero extend-56	`uint64(uint8(x))`	`uint64((x))`

Table 4.1: Intermediate casts applied to emulate zero and signed extensions in `Solidity`.

We now represent an 8 bit vector in `Solidity` as the variable **`int8`** `x` and want to emulate zero extension in the same way. We can try to cast in the same way as in `C` by calling **`uint32`**`(x)` but this does not compile in `Solidity`. `Solidity` does not allow casting from unsigned to signed or signed to unsigned while changing the bit size of the integer at the same time. Therefore, we have to apply intermediate casts when translating signed and zero extensions to `Solidity`. In order to zero extend `x` to a 32 bit width, we first cast `x` to an unsigned integer of the same bit width and then extend the bit width to 32 by calling **`uint32`**`(`**`uint8`**`(x))`. To signed extend `x` we can simply cast to the desired size without any intermediate steps by calling **`int32`**`(x)`. In order to be able to apply intermediate casts correctly, we have to keep track of the last cast of each variable or constant during the generation process.

In Table 4.1 we show the intermediate casts used to emulate signed and zero extensions in `Solidity`. It shows how signed and unsigned integers of bit width 8 are extended to different bit widths using signed and unsigned extension. The columns represent the type of some variable `x` which can emulate either a signed or unsigned bit vector of width 8. The rows represent whether to apply signed or unsigned extension and how many bits to extend `x` by.

CHAPTER 5

Experimental Results

In this chapter, we will first go into detail on how we conduct our benchmarks, including which fuzzers are measured, and the program set the fuzzers are tested with. We will then proceed to show our results and discuss them.

5.1 Experimental setup

The benchmark consists of 120 different programs, generated by Muzzle. This program set is composed of four different maze dimensions:

- 5×5,
- 10×10,
- 15×15,
- and 20×20.

We generate 15 mazes for each dimension using the Prims algorithm. Each maze is then used to generate two programs, one using a random CVE file to generate branch constraints and one using trivial branch constraints, taken from a predefined set of boolean expressions. Therefore, there are 30 different input programs generated per maze dimension.

A campaign represents a single execution of one fuzzer on one input program with a specific starting seed passed to the fuzzer. We run five such campaigns per generated program per fuzzer. Thus, in total, 600 campaigns are executed per fuzzer. Each campaign is interrupted after thirty minutes.

We benchmark three `Solidity` fuzzers: Foundry fou [2022], Echidna Grieco et al. [2020] and ItyFuzz Shou et al. [2023] and two `C` fuzzers: AFL++ [Fioraldi et al., 2020] and LibFuzzer Project [2018].

The benchmarks are run on an AMD Epyc 7702 processor, which has 64 cores and a single core performance of 2.0 GHz up to 3.35 GHz, on a machine with 512 GB DDR4 RAM.

5.2 Results

Figure 5.1 shows how many bugs each of the benchmarked fuzzers was able to find during all executed campaigns. The evaluated fuzzers are represented on the y-axis and labelled using their names. The x-axis represents the number of bugs found by each fuzzer, which are grouped by the maze dimensions used to generate the input program the fuzzer found the bug in. The total number of bugs to be found per fuzzer and maze dimension is 150 since we ran five campaigns for each of the 30 input programs of a maze dimension.

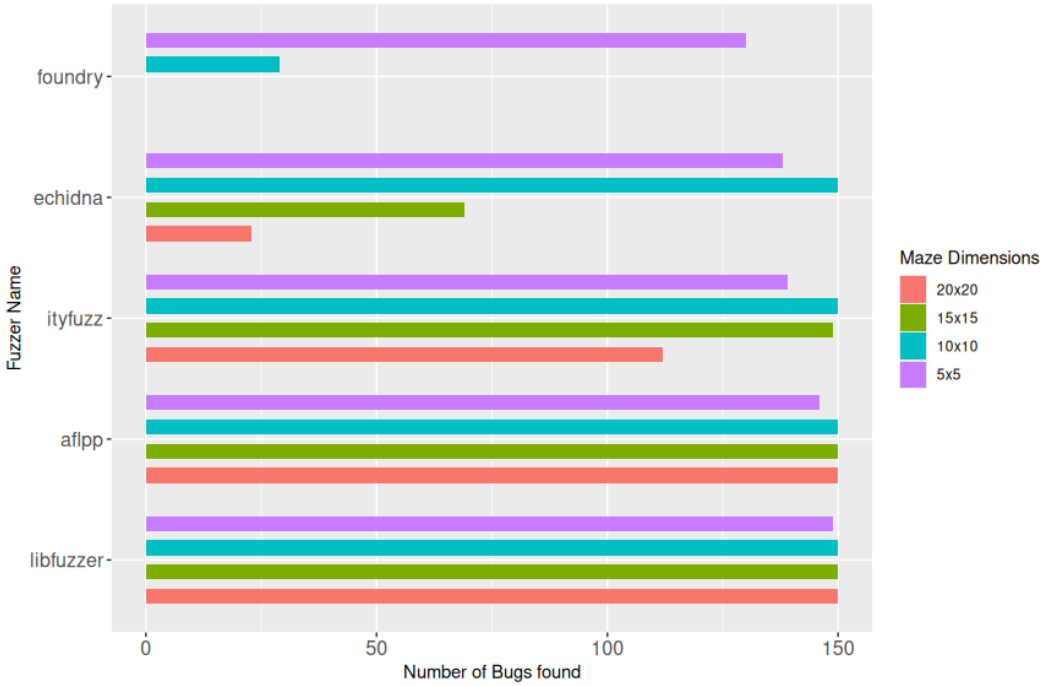

Figure 5.1: Number of bugs found by each fuzzer.

The three benchmarked `Solidity` fuzzers can be seen at the top of Figure 5.1, namely Foundry, Echidna and ItyFuzz. Only Echidna and ItyFuzz were able to solve any of the input programs generated from 15×15 and 20×20 mazes. The `C` fuzzers on the other hand were able to solve all of them. It is interesting to note that some of the programs generated from the smaller 5×5 mazes were not solved by the `C` fuzzers, this

is due to complex branch constraints generated from some of the CVE files. ItyFuzz is the best performing `Solidity` fuzzer, being able to solve all of the 10×10 mazes and significantly more 15×15 and 20×20 mazes than the second best performing `Solidity` fuzzer, Echidna. Echidna is also able to solve all 10×10 mazes but solves less than half of the 15×15 mazes. The fuzzer even manages to solve some of the 20×20 mazes, but significantly less than ItyFuzz. On the other hand, Foundry is outperformed heavily by both Echidna and ItyFuzz, being able to solve only less than a third of the 10×10 mazes.

Figure 5.2 shows a cactus plot describing how long each fuzzer took to solve the mazes it found bugs in. The x-axis describes the number of mazes the fuzzer was able to solve in the time described by the y-axis. Here, the maximum number of bugs each fuzzer could find surmounted to 600 since there were 120 programs to find bugs in and five campaigns were run per fuzzer per program. The longest a fuzzer could take to solve a maze was thirty minutes, as a fuzzing run was interrupted after that time. We show the time on a logarithmic scale, as the fuzzers differ too greatly in how long they take to find the bugs to be able to read the `C` fuzzers' and ItyFuzz's times otherwise.

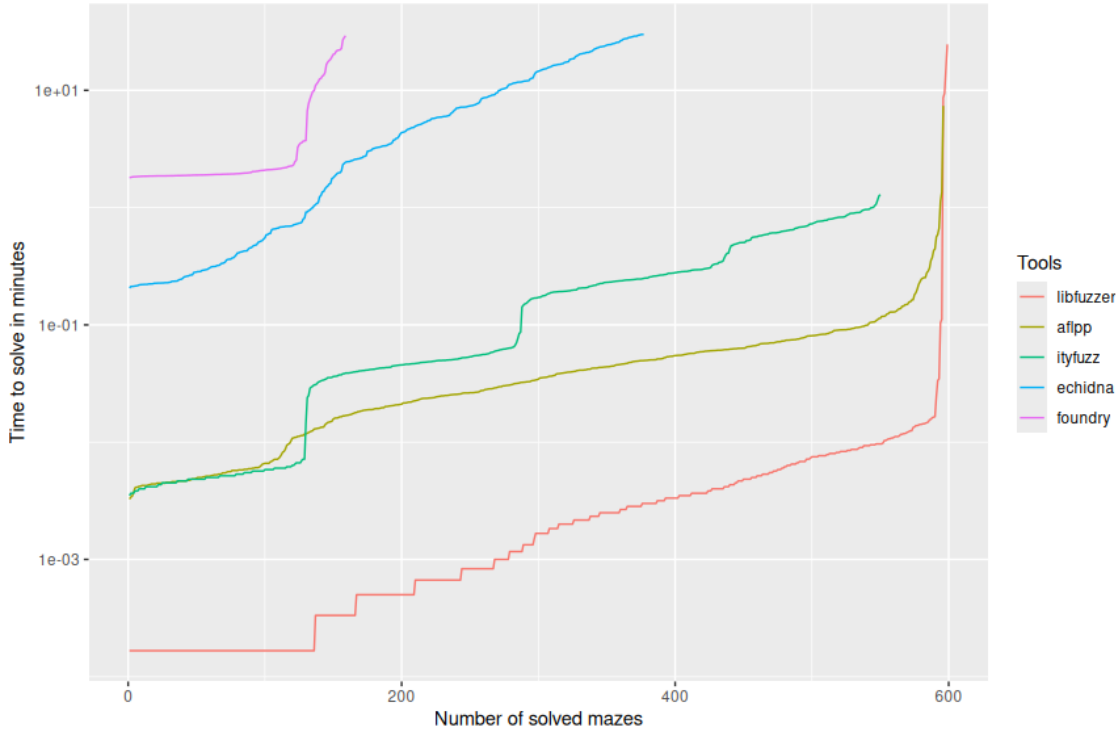

Figure 5.2: Time taken for the fuzzers to find bugs.

As can be seen, Foundry not only finds the fewest bugs but also takes more time to find each of them, it takes a long time to find bugs in even the simplest instances. Echidna on the other hand, exhibits runtimes that increase with the size and depth of the programs it is executed on. While ItyFuzz is significantly quicker on all of the instances it is able

to solve than Echidna and Foundry, it still lags behind the `C` fuzzers when it comes to more complex input programs. The `C` fuzzers and ItyFuzz take similar amounts of times in solving the smaller mazes, but the difference is that LibFuzzer and AFL++ solve more mazes than ItyFuzz and are quicker when it comes to solving the larger and more complex mazes.

CHAPTER 6

Conclusion

Fuzzle Lee et al. [2022] is a `C` fuzzer benchmarking tool which generates `C` programs from mazes. In this thesis, we introduced Muzzle, an extension to Fuzzle which allows benchmarking of `C` and `Solidity` fuzzers. Muzzle generates highly configurable mazes and uses them to construct semantically equivalent `C` and `Solidity` programs. Intuitively, mazes are translated to programs such that the entry point of the maze corresponds to the entry point of the program and the goal of the maze corresponds to some bug in the program. A fuzzer, which takes such a program as its input, can then be seen as an agent traversing the maze and trying to find the maze's goal. We use programs generated by Muzzle to evaluate different `Solidity` and `C` fuzzers on whether they can find the bugs and how fast they find them if they do.

We used Muzzle to generate 120 different input programs to evaluate the `Solidity` fuzzers: Echidna Grieco et al. [2020], Foundry fou [2022] and ItyFuzz Shou et al. [2023] and compare them to the `C` fuzzers: AFL++ Fioraldi et al. [2020] and LibFuzzer Project [2018]. We observed that all `Solidity` fuzzers were outperformed by the `C` fuzzers in terms of how many bugs they were able to find as well as how quickly they were able to find the bugs. Within the `Solidity` fuzzers, ItyFuzz was not only able to find the most bugs, but it was also the only `Solidity` fuzzer which came close to the `C` fuzzers in terms of speed. While Echidna was not able to solve as many mazes as ItyFuzz, it still managed to solve almost half of the 15×15 mazes, however it did so with significantly worse runtimes. Foundry on the other hand, was heavily outperformed by the other `Solidity` fuzzers in both performance and its ability to find bugs in the input programs. It was the only fuzzer unable to solve any 15×15 mazes and only solved a small amount of the 10×10 ones.

6.1 Future Work

We believe Muzzle can be a valuable tool to guide researchers in improving existing fuzzers or even when developing completely new ones.

In order for researchers to be able to use Muzzle in this way, there should be a modular and easy-to-use process for integrating a new fuzzer to be benchmarked by Muzzle. We have already provided a way to add a new fuzzer to the benchmarking process where researchers only need to add a docker image for the fuzzer to be run in, a script describing how the fuzzer should be executed and a collection script which extracts the relevant information from the fuzzer's output. Additionally, it would be helpful to modularise the program generation such that fuzzer-specific pieces of code like the bug-function or main can be defined when adding a new fuzzer.

As there are many more `C` and `Solidity` fuzzers available in the language's respective ecosystems, adding them to the default set of supported fuzzers is a worthwhile extension of Muzzle.

A more ambitious extension to Muzzle would be to generate benchmarks in even more different languages. Popular languages such as `Python`, `Java`, `JavaScript` or `C++` could be supported, since it would be interesting to see how their fuzzers compare to `C` ones. On the other hand, there are also more young or niche languages similar to `Solidity` which could benefit from options for evaluating the performance of their respective tools.

Muzzle could also be improved in its evaluation metrics, since we currently only evaluate fuzzers based on their speed and ability to find bugs. A common evaluation metric for fuzzers is code coverage, which is currently not supported by Muzzle due to some of the `Solidity` fuzzers not providing any such information. However, Fuzzle does evaluate code coverage of the `C` fuzzers and some of the `Solidity` fuzzers such as Echidna provide code coverage information as well. Therefore, it could be useful to add an option to the benchmark configuration file to define that a benchmark should collect code coverage information with the prerequisite that all measured fuzzers for that benchmark are able to provide code coverage information.

The results from our benchmarks could be used to compare the features of weaker fuzzers to better performing ones and improve the former.

As previously mentioned in Section 2.4 there is some research on comparing fuzzer benchmarking tools. Using a similar method to compare Muzzle to other state of the art fuzzer benchmarking tools could provide some valuable insights into how well constructed and varied our input program generation methods actually are.

List of Figures

List of Tables

List of Listings

Bibliography

Afl, 2013. URL https://github.com/google/AFL.

Mazelib, 2014. URL https://github.com/john-science/mazelib.

Oss-fuzz, 2016. URL https://google.github.io/oss-fuzz/.

Aflplusplus, 2019. URL https://github.com/AFLplusplus/AFLplusplus.

Foundry, 2022. URL https://github.com/foundry-rs/foundry.

Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2329–2344. ACM, 2017. doi: 10.1145/3133956.3134020. URL https://doi.org/10.1145/3133956.3134020.

Marcel Böhme, László Szekeres, and Jonathan Metzman. On the reliability of coverage-based fuzzer benchmarking. *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 1621–1633, 2022. URL https://api.semanticscholar.org/CorpusID:247014519.

Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. FUZZOLIC: mixing fuzzing and concolic execution. *Comput. Secur.*, 108:102368, 2021a. doi: 10.1016/j.cose.2021.102368. URL https://doi.org/10.1016/j.cose.2021.102368.

Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. Fuzzing symbolic expressions. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 711–722. IEEE, 2021b. doi: 10.1109/ICSE43902.2021.00071. URL https://doi.org/10.1109/ICSE43902.2021.00071.

Josh Bundt, Andrew Fasano, Brendan Dolan-Gavitt, Will Robertson, and Tim Leek. Evaluating synthetic bugs. *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021. URL https://api.semanticscholar.org/CorpusID:232435337.

Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. Grey-box concolic testing on binary code. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 736–747. IEEE / ACM, 2019. doi: 10.1109/ICSE.2019.00082. URL https://doi.org/10.1109/ICSE.2019.00082.

Riccardo Felici, Laura Pozzi, and Carlo A. Furia. Hyperput: generating synthetic faulty programs to challenge bug-finding tools. *Empirical Software Engineering*, 29, 2022. URL https://api.semanticscholar.org/CorpusID:252222319.

Andrea Fioraldi, Dominik Christian Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In Yuval Yarom and Sarah Zennou, editors, *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*. USENIX Association, 2020. URL https://www.usenix.org/conference/woot20/presentation/fioraldi.

Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. Echidna: effective, usable, and fast fuzzing for smart contracts. In Sarfraz Khurshid and Corina S. Pasareanu, editors, *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, pages 557–560. ACM, 2020. doi: 10.1145/3395363.3404366. URL https://doi.org/10.1145/3395363.3404366.

Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. In Longbo Huang, Anshul Gandhi, Negar Kiyavash, and Jia Wang, editors, *SIGMETRICS '21: ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems, Virtual Event, China, June 14-18, 2021*, pages 81–82. ACM, 2021. doi: 10.1145/3410220.3456276. URL https://doi.org/10.1145/3410220.3456276.

ISO. *ISO/IEC 9899:2018 Information technology — Programming languages — C*. Fourth edition, June 2018. URL https://www.iso.org/standard/68564.html.

Haeun Lee, Soomin Kim, and Sang Kil Cha. Fuzzle: Making a puzzle for fuzzers. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pages 45:1–45:12. ACM, 2022. doi: 10.1145/3551349.3556908. URL https://doi.org/10.1145/3551349.3556908.

Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem A. Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In *USENIX Security Symposium*, 2020. URL https://api.semanticscholar.org/CorpusID:222132978.

Jonathan Metzman, László Szekeres, Laurent Maurice Romain Simon, Read Trevelin Sprabery, and Abhishek Arya. FuzzBench: An Open Fuzzer Benchmarking

Platform and Service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1393–1403, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385626. doi: 10.1145/3468264.3473932. URL https://doi.org/10.1145/3468264.3473932.

Sutton Micheal, Greene Adam, and Amini Pedram. *Fuzzing: Brute Force Vulnerability Discovery*. Pearson Education, 2007. ISBN 9780321680853.

LLVM Project. libfuzzer, a library for coverage-guided fuzz testing, September 2018. URL https://llvm.org/docs/LibFuzzer.html. Online; accessed: 02 Apr. 2024.

Chaofan Shou, Shangyin Tan, and Koushik Sen. Ityfuzz: Snapshot-based fuzzer for smart contract. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 322–333, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702211. doi: 10.1145/3597926.3598059. URL https://doi.org/10.1145/3597926.3598059.

Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

Valentin Wüstholz. Daedaluzz, 2023. URL https://consensys.net/diligence/blog/2023/04/benchmarking-smart-contract-fuzzers/.

Michal Zalewski. American fuzzy lop, 2019. URL https://lcamtuf.coredump.cx/afl/.

Zenong Zhang, Zach Patterson, Michael W. Hicks, and Shiyi Wei. Fixreverter: A realistic bug injection methodology for benchmarking fuzz testing. In *USENIX Security Symposium*, 2022. URL https://api.semanticscholar.org/CorpusID:252564161.