École Polytechnique Fédérale de Lausanne

JEDIFUZZ : compiler-based transformative fuzzing

by Lucio Ezechiele Romerio

Master Thesis

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Adrian Herrara
External Expert

Prof. Dr. sc. ETH Mathias Payer
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

March 17, 2022

I have not failed. I’ve just found 10’000 ways that won’t work.
— Thomas A. Edison

Dedicated to whoever is reading this…
May the force be with you!

Acknowledgments

I would like to thank my supervisor, Prof. Mathias Payer, for giving me the opportunity of being part of the "Hive" and introducing me to the research world.

The whole HexHive lab, especially the LLVM people — Jelena, Nicolas and Adrian, our mentor: for the long discussions and the cooperation at unveiling compiler misteries.

My family, for their unconditional support and for always being present when I need them. Even if it is to speak about fuzzing, compilers and other obscure concepts.

Last but not least, my girlfriend, for the incommensurable help and for her limitless patience. Being confined because of Covid-19, she discovered how I deal with bugs: by raging at them and hitting the desk with my head. She silently became an expert at evaluating how bad a bug is from my reactions. A skill that she probably never wanted to acquire.

Lausanne, March 17, 2022 Lucio Ezechiele Romerio

Abstract

Fuzzing is an effective technique to find bugs. It consists in feeding a huge amount of randomly generated input to a target binary, and in monitoring the executions for crashes and unusual behaviour. Although fuzzing scales really well and is easy to deploy, it is limited by coverage walls. In fact, since they rely on randomness, fuzzing engines often struggle to generate inputs that bypass hard-to-satisfy checks — like checksums, hashes and magic values. This prevents them from finding bugs hidden in deep execution paths.

To improve coverage, existing techniques leverage symbolic execution and taint analysis to generate inputs bypassing hard-to-satisfy checks. Those approaches are often complex and imprecise; plus, they are subject to overhead issues and do not scale. Transformative fuzzing has shown to be a better solution to this problem. The state-of-the-art transformative fuzzer is T-Fuzz, which bypasses coverage walls by randomly negating conditional jumps in the program. Although it showed good results, this approach is imprecise and is limited to if statements, lacking in flexibility.

We propose JEDIFUZZ, a compiler-based transformative fuzzer addressing the coverage wall problem through control-flow enforcement. Our system efficiently collects tailored run-time information about check constraints and their relations. When a coverage wall is hit, this information is leveraged to carefully select the most promising non executed branch. Then, thanks to the instrumentation injected at compile time, the program is transformed so that the control-flow is enforced to take that branch.

JEDIFUZZ has been designed to be easily integrated into any C/C++ fuzzer. For the evaluation, we integrated it into MOpt-AFL and compared the performances. Our preliminary results have shown that JEDIFUZZ improves the coverage performance of 25-30% over MOpt-AFL, and that it can more reliably find bugs hidden in deep execution paths.

Contents

Chapter 1

Introduction

Software have bugs [6]: some of them are just harmless errors, others may result in critical vulnerabilities affecting a large amount of users. It is thus crucial to have multiple layers of security, aiming to prevent and detect those bugs as soon as possible. Over the years mitigations like Address Space Layout Randomization (ASLR) [25], executable space protection (NX bit) [26], and Control Flow Integrity (CFI) [1] have been developed, often as a response to the rise of a new attack technique. Mitigations make it harder to perform some type of attacks, but they are often inefficient and it's not uncommon for hackers to bypass them [8]. Software testing remains thus fundamental and widely used; over the last few years, a testing technique has particularly gained traction: fuzzing.

Given a target binary, fuzz testing executes it as many times as possible with random or semi-random data as input. The fuzzing engine monitors each execution detecting crashes, errors, and unusual behaviour. Setting up a fuzzing campaign requires little human effort beside the selection of an initial corpus of seeds, making fuzz testing fast to deploy and easily accessible even to inexperienced users. The fuzzer generates new test cases starting from the initial corpus, and it executes those test cases in parallel. In fact, test cases executions are independent, allowing fuzzing to scale to an arbitrary number of CPUs. Its simplicity and scalability, together with its effectiveness at finding bugs [10], have made fuzzing popular [16] both in the industry and the academia.

In the recent years, coverage-guided fuzzing has become one of the most widely used fuzzing strategies [15]. Fuzzing engines based on this strategy use coverage metrics as a feedback to track their progress. The intuition behind this approach is that bugs are uniformly distributed, thus a higher code coverage results in a better chance to find bugs. To maximize their code coverage, fuzzers usually select inputs that produced new coverage for further mutations and input generation.

Since fuzzers use random inputs, it may be difficult for them to satisfy complex checks over multiple bytes of the inputs (e.g., a checksum). When a fuzzing engine encounters such a check, that prevents it from making further progress, it is said to have hit a coverage wall. Coverage walls are a well known fuzzing problem and over the years several approaches to bypass them have been explored. Some fuzzers proposed to use symbolic execution [22][28] to carefully generate inputs that succeed in passing complex checks. Other works [27][5][19] leverage taint analysis to identify relations between program logics and inputs, allowing the fuzzing engine to better select where and how to mutate the input. Those approaches, although they succeed in generating inputs that satisfy previously failing checks, have some limitations: taint analysis is heavy weight, while symbolic execution introduces a huge overhead and does not scale.

By proposing to transform not only the input, but also the program itself, T-Fuzz [18] introduced a new fuzzing technique: transformative fuzzing. When the fuzzer hits a coverage wall, T-Fuzz mutates the program by inverting one of the repeatedly failing checks. It creates a new version of the binary and stores it in a queue, together with the original and all other transformed programs. At each iteration, one program is selected from this queue and fuzzed. Of course modifying, the program may introduce new bugs. For this reason, each crash found by T-Fuzz goes through a symbolic execution based post-analysis, which takes care of filtering out false positives. Removing symbolic execution from the fuzzing routine makes T-Fuzz more lightweight than the previously presented approaches. Nevertheless, the overhead problem is just postponed to the crash analyser, which becomes the new bottleneck preventing T-Fuzz from scaling. Another limitation of this system is a lack in flexibility and smartness. In fact, it can handle only if statements and selects the check to disable without following any particular policy.

JEDIFUZZ is the first compiler-based transformative fuzzer. Differently from T-Fuzz, it requires access to the source code, but the ability of instrumenting the target binary brings some advantages.

- Checks can be disabled directly in the fuzzed binary, through the instrumentation added at compile time. This dynamic strategy simplifies program transformation, removing runtime patching.
- The binary can be instrumented to collect tailored runtime information about checks constraints and their relations, allowing to smartly select the check to disable.
- Instructions can be added, removed or modified, providing huge flexibility and allowing to handle other cases beside if statements.

Additionally, JEDIFUZZ exploits the source code to perform a static analysis of the program. Through this analysis, it builds a ranking of conditional branches. When the fuzzer hits a coverage wall, JEDIFUZZ leverages the information collected at runtime to identify the conditions that were reached so far and the branches that were taken. Then, thanks to the previously generated ranking, it selects the most promising unexplored branch reached so far, and efficiently transforms the program to take it.

In a preliminary evaluation, we have compared JEDIFUZZ with MOpt-AFL. According to the results, our system reaches bugs hidden in deep execution paths in a more reliable way, and it improves the code coverage performance over MOpt-AFL by 25-30%.

Chapter 2

Background

This section gives an overview of fuzzing and of the LLVM project [24]. We will start with a general discussion about fuzzing, presenting the problem of coverage wall and the different approaches addressing it. Afterwards, the LLVM project and its structure will be briefly introduced.

2.1 Fuzzing

Thousands of bugs are discovered every year [6]: some of them are security flaws which could be exploited by an attacker to compromise the system. The size of modern software and its complexity make it almost impossible to reason about the software itself and its correctness. Manual testing, auditing, and bug bounty programs regularly find bugs, but because they rely on humans, the complexity of today software prevents them from scaling. Those methods are thus insufficient.

Fuzzing is an automated testing technique that generates test cases and feeds them to a target binary. It then monitors the execution to detect crashes and unusual behaviour. Each test case execution is independent, thus multiple test cases can be executed in parallel. As a consequence, fuzzing campaigns can be deployed on an arbitrary amount of CPUs, allowing fuzzing to scale really well. Another strength of fuzzing is its simplicity, which makes it easily accessible even to inexperienced users. In fact, setting up a fuzzing campaign is generally a fast procedure and requires little human effort. Last but not least, it is very effective at finding bugs: to make an example, the Google's OSS-Fuzz project[10] has found thousands of bugs over the last few years by continuously fuzzing open source software. Thanks to its scalability, simplicity, and effectiveness, fuzzing has gained traction and is now widely used, both in the industry and in the academia.

There are different ways to categorise fuzzers; one of them is to consider how they generate new inputs. In this case, they can be roughly divided into two categories: generational and mutational fuzzers. Generational fuzzers [7] [20] [2] acquire knowledge about the input format, often from format specifications provided by the user, and generate new inputs according to it. On the other hand, mutational fuzzers [29] [9] create new inputs by randomly mutating an initial seed. Providing an initial corpus of seeds for mutational fuzzers is fairly easy. But creating format specifications for a generational fuzzer requires a lot of manual work, and may be infeasible for large programs. For those reasons, recent work mainly focuses on mutational fuzzers.

Mutational fuzzers can be further divided into three categories: whitebox, blackbox, and greybox. Whitebox fuzzers assume access to the source code and use it to perform a static analysis of the program. This process usually aims to understand the impact of different inputs on the program execution. Blackbox fuzzers are on the opposite side and do not have access to any of the program internals. Lastly, greybox fuzzers are somehow in the middle: they limit themselves to a lightweight analysis of the binary code of the application, without requiring the source code itself.

An important number of state-of-the-art fuzzers are coverage-guided [15]. This means they use coverage as a metric to track progress and to select future inputs. In software development, code coverage is used to approximate the percentage of code executed by test cases. A coverage of 100%, which in fact is often impossible to reach, does not provide any guarantees about bugs absence. Such an high coverage only implies that each line of code in the program has been executed at least once. In the other hand, a low percentage indicates that a large portion of the codebase has not been tested yet. The intuition behind coverage-guided fuzzing is that bugs are uniformly distributed in the code; thus, to discover more bugs, we need to maximize code coverage. To achieve this, fuzzers identify the inputs that produce new coverage and select them for future mutations.

Fuzz testing is a random process, making it difficult for fuzzing engines to satisfy complex checks over multiple bytes of the input — like checksums. When such a check prevents the fuzzer from making further progress, the latter is said to have hit a coverage wall. Coverage walls are a common problem of coverage-guided fuzzers; several approaches addressing them have been proposed over the years. Driller [22] and QSYM [28] leverage concolic execution (a combination of symbolic and concrete execution) to create inputs which satisfy complex checks. Symbolic execution simultaneously explores multiple paths in the program, keeping track of all its possible states. The number of possible states in a program exponentially increases with the number of branches, causing path explosion and preventing symbolic execution based approaches from scaling [3]. Taint analysis recognizes data dependencies, associating parts of the input with check constraints. Taintscope [27], VUzzer [19], and Angora [5], leverage taint analysis to identify where and how to mutate the input. Although it produces interesting results, this technique is heavy weight and suffers from over- and under-tainting [13].

T-Fuzz [18] introduced the idea of transformative fuzzing, which consists in mutating not only the inputs, but also the program itself. When the fuzzing engine hits a coverage wall, T-Fuzz modifies the program by inverting one of the failing checks. Transforming a program may remove security checks and introduce bugs. To deal with this problem and filter out false positives, T-Fuzz comes with a crash analyser. When a crash is found, both the crashing input and the transformed program are passed to this crash analyser. The latter leverages symbolic execution to verify whether the triggered bug is reproducible in the original binary. Figure 2.1 shows an overview of T-Fuzz.

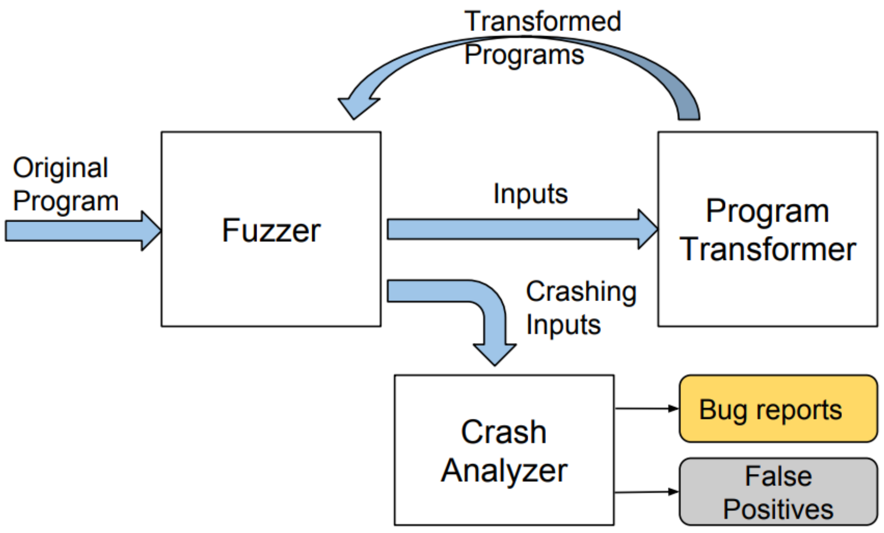

Figure 2.1: Overview of T-Fuzz [18]

T-Fuzz is a greybox fuzzer, thus it does not have access to the source code. It transforms the program by patching the binary on the fly, overwriting some instructions. This is possible because the negation of an if statement results in an instruction of the exact same size as the original one (i.e., one byte). While being smart, this approach is limited to if statements. Patching at a binary level other instructions — that do not preserve their size when negated — is less straightforward, and can become overcomplicated.

2.2 LLVM

The LLVM project [24] is a compiler framework written in C++. It is composed by a series of compiler and toolchain technologies that aims to optimize programs at compile-, linking-, and running time.

The framework comes with different analysis and transformation passes; a pass represents one step in the framework pipeline, which is often composed by a chain of different passes. Passes can be divided in two main categories: analysis and transformation. The first only performs an analysis and does not modify the code. The second type — as suggested by its name — transforms the program by injecting, removing, and/or modifying instructions. Transformation passes can even inject declarations and calls to functions that are not present in the original program. Those functions can then be implemented separately and linked to the binary as external libraries.

The LLVM Intermediate Representation (LLVM IR), around which the project is built, is an ad-hoc representation, independent both from the programming language of the source code, and from the architecture. This common representation, paired with a very precise yet simple structure, makes the LLVM project highly extensible. The biggest component of the LLVM structure is a module, followed by functions, basic blocks and instructions. A module roughly corresponds to a source file and it is composed of functions (the functions in the file). Functions are made of basic blocks, which are compounded of instructions.

A rich abstraction, together with the structure presented above, makes this tool very powerful. In fact, it allows the programmer to work at different granularities and to easily interact with the different components and subcomponents of the project. A fuzzing engine can leverage this framework to instrument target binaries, so that they efficiently collect runtime information. In the case of transformative fuzzing, it can also be used to modify the program so that it can be dynamically transformed at runtime.

Chapter 3

Design

The aim of this section is to present the design of JEDIFUZZ. After introducing its goals, we will give an overview of the whole system, and then we proceed by diving into the different components and their details.

3.1 Goals

Transformative fuzzing is a technique addressing the problem of coverage wall. T-Fuzz, the first transformative fuzzer, showed to perform better than symbolic execution and taint analysis based approaches. Nevertheless, it still has some limitations. Looking at those limitations, we realized that some of them can be addressed by a whitebox compiler-based approach.

- If we transform the program at runtime, we are limited to inverting if statements. But if we do it at compile time, this gives us much more flexibility: instructions can be modified, deleted or injected in any part of the program, enabling us to handle different cases.
- We can instrument the code to enable dynamic control-flow enforcing at runtime. This removes the need for runtime patching, resulting in a much more elegant approach.
- Through a compiler-based approach, the program can be instrumented to efficiently collect tailored information about checks. Those information can then be used, when a coverage wall is hit, to carefully select the check to bypass.
- Having access to the source code, a static analysis can be performed before launching the fuzzing campaign. This allows us to collect even more information about checks and their relations.

JEDIFUZZ is the first compiler-based transformative fuzzer. Its goal is to exploit the above listed advantages to improve the state-of-the-art.

3.2 Overview

JEDIFUZZ is a whitebox transformative fuzzer designed to be easily integrated into any C/C++ fuzzing engine. In fact, to extend an existing fuzzer with JEDIFUZZ, little modifications — apart from including the monitor library presented in section 3.5 — are required.

Figure 3.1 shows how JEDIFUZZ can be roughly divided into three parts: the setup phase, happening once; the fuzzing itself; and the post analysis that each crash will undergo. Each part is further divided into different components, which are briefly introduced here below.

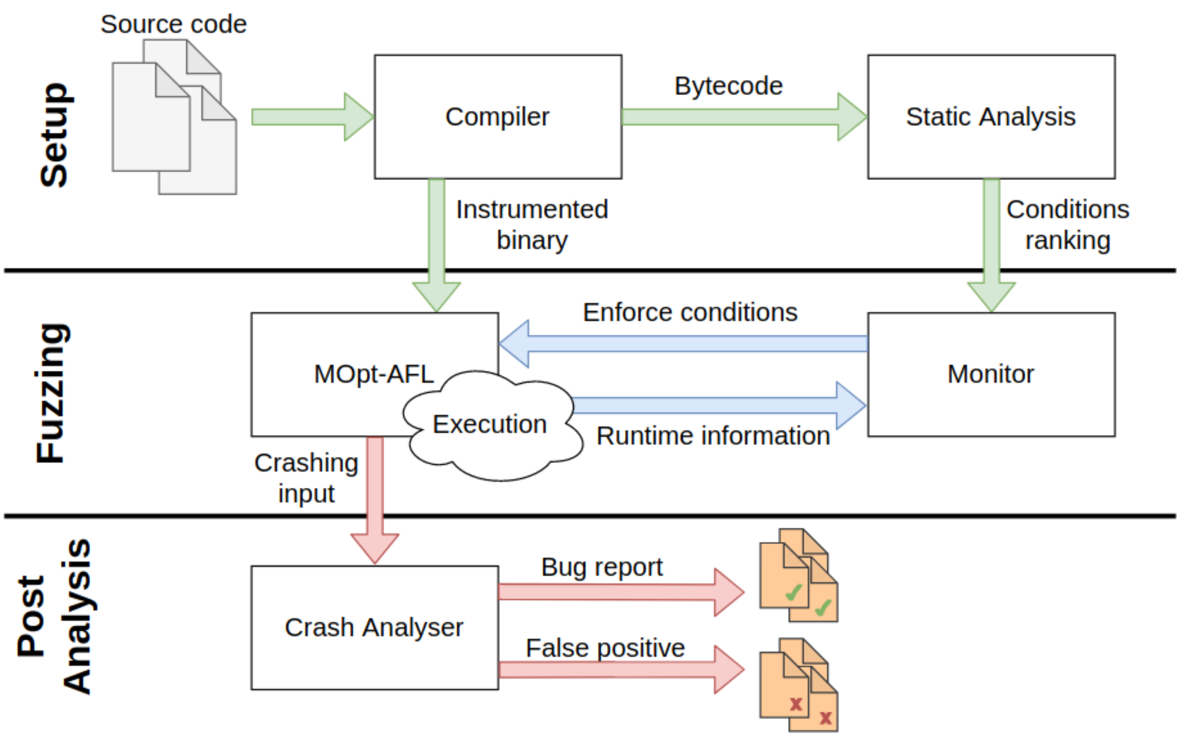

Figure 3.1: JEDIFUZZ overview.

The instrumentation has three purposes: to uniquely identify checks, to add the logic to enforce condition outcomes, and to collect useful runtime information. The type of checks handled by the first version of JEDIFUZZ— presented in this report — are if statements and $switch$ statements.

Before running the fuzzing campaign, a static analysis inspecting the Control Flow Graph (CFG) of the target program is performed. The goal of this analysis is to generate a ranking of the instrumented checks. During the fuzzing campaign, when the fuzzer hits a coverage wall, this ranking will be used to select the most promising check to bypass (i.e., the one with the highest rank).

The monitor consists in a static library containing the logic to interact with the instrumentation. This logic includes: setting up the system, aggregating and storing the information collected at runtime, selecting which check to disable when the fuzzer hits a coverage wall, and transforming the program.

By transforming the program, JEDIFUZZ may introduce new bugs. Thus, to avoid false positives, some kind of post analysis is needed. Nevertheless, this project focuses on the fuzzing part, assuming a manual crash analyser. The crash analyser is thus considered out of scope and will not be discussed further in this report.

3.3 Instrumentation

Our approach towards coverage walls is to enforce the outcome of hard-to-satisfy checks, allowing the fuzzer to bypass them. JEDIFUZZ currently supports bypassing if and $switch$ statements. In order to disable a given check, we need the ability to refer to it in a unique way. For this purpose, JEDIFUZZ assigns a unique ID to all supported checks in the program. Since IDs are assigned in a incremental way, if the total amount of if and $switch$ statements in the program is n, we will end up having conditions with IDs between 0 and $(n-1)$. The motivation behind this approach is to simplify the storage of information about checks and the access to them, as it will be shown in section 4.1.

Unfortunately, in some cases, identifying conditions by unique IDs is not sufficient to bypass hard-to-satisfy checks. An example of this is present in Listing 3.1. At line 29, there is an access out of bound to the t array. The bug is triggered when $b == 1$, i.e. when both $check\_first$ and $check\_second$ return $true$. While the first function returns $true$ in most cases, the second one does it only when the second element of t is equal to *0xBADDCA53*. Selecting an input for which $check\_second$ returns $true$ would thus be difficult for a fuzzer. To understand why IDs are not sufficient in this case, we need to reason about how JEDIFUZZ will threat this example. The only condition in the whole program is the if statement at line five. If we enforce it to return $true$, $check\_first$ will always return $true$, while $check\_second$ will always return $false$. If instead we enforce the if statement to always return $false$, the first function will return $false$, and the second one $true$. Thus, in both cases, the bug will not be reached. In order to trigger the bug, we need to enforce conditions outcomes in a more fine-grained way. In our example, the if statement at line five should be enforced to $true$ only when reached from $check\_second$. One possible solution is to use the calling context, which encodes the path to the currently executed line of code as a list of functions. This list, similarly to an error stack trace, contains all parent functions that were called before reaching that line of code. Going back to the example in Listing 3.1, we can act as follows. First, we identify the condition at line five by its unique ID, then we verify the calling context. If and only if the latter corresponds to

$$[...] \rightarrow foo \rightarrow check\_second \rightarrow equal$$

we enforce the outcome of the conditions to be $true$. This example proofs the usefulness of taking the calling context into account. This allows us to enforce condition in a more fine-grained way, and to succeed at bypassing hard-to-satisfy checks where IDs alone are not sufficient.

To summarize, the instrumentation assigns unique IDs and adds the logic to keep track of the calling context. It instruments all conditional branches in the program, so that their outcome can be enforced, and it injects the code collecting the runtime information for the monitor.

3.4 Static analysis

A key question in transformative fuzzing is which check to bypass when the fuzzer gets stuck. To avoid hitting a new coverage wall a few iterations after transforming the program, it is crucial to optimize this choice. For this purpose, JEDIFUZZ ranks all instrumented checks in the program through a static analysis of the CFG. This analysis is performed only once, before starting the fuzzing campaign, and supports different ranking policies.

Since JEDIFUZZ is a coverage-guided fuzzer, we defined four different policies aiming to maximize code coverage.

- *Simple Blocks* reasons about code in terms of basic blocks. It assigns to each branch a weight that corresponds to the number of basic blocks reachable only by taking that branch.
- *Recursive Blocks* uses the same principle of the *Simple Blocks* policy. Additionally, it recursively inspects code blocks for function calls. For each function call, the number of basic blocks contained in it is considered in the weight computation.
- *Simple Instructions*, given a branch, computes its the weight by counting the number of instructions reachable only by taking that branch. Function calls are ignored and considered as a single instruction.
- *Recursive Instructions* computes the branch weights in the exact same way as *Simple Instructions*, but by recursively taking into account functions calls. Thus, for each function call, the number of instructions composing that function contributes to the final weight.

The difference between basic blocks and instructions based policies can be seen by assuming the following setup: there is one branch unlocking multiple small basic blocks; and another

```c
#include <stdbool.h>

bool equal(int a, int b)
{
    if (a == b)
        return true;
    else
        return false;
}

bool check_first(int v)
{
    return !equal(v, 1337);
}

bool check_second(int v)
{
    return equal(v, 0xBADDCA53);
}

void foo(int t[2])
{
    bool b = check_first(t[0])
    b &= check_second(t[1])

    ...

    // Crash if b == 1
    t[2*b] = 0;

    ...
}
```

Listing 3.1: Pseudo code of a dummy program. This example shows that it may be insufficient to identify the condition to enforce only by its unique ID.

one unlocking a single basic block, but composed by a lot of instructions. Basic blocks based policies will prioritize the first branch, while instructions based policies will prioritize the second one. To motivate the need of recursive policies, we can consider another example. There are two branches: the first one unlocks two instructions; the second one unlocks one instruction, a call to a huge function. The *Simple Instructions* policy will prioritize the first branch, resulting in a small code coverage contribution. On the other hand, *Recursive Instructions* will select the second branch, generating much more code coverage. The performance of this four different policies will be discussed in section 5.6.

We would like to underline that the whole static analysis framework is built in a flexible way. As a consequence, new policies can be easily defined, integrated, and used.

3.5 Monitor

To enable the fuzzer to interact with the instrumentation, we have created a static library implementing a monitor object. This object allows the fuzzing engine to easily access the logic to transform the program. After being initialized, the monitor enables the fuzzing engine to transform the program through a simple function call. When that function is called, the monitor will act as follows.

1. It will inspect the information collected at runtime to identify enforceable conditions. Enforceable conditions are the ones that have been reached and have only one executed branch.
2. It will select the most promising enforceable condition, according to the ranking it was provided with.
3. It will transform the program by enabling the instrumentation to enforce the outcome of the selected condition.

This design makes the integration of JEDIFUZZ into an arbitrary fuzzer a simple process. It suffices to import our library, to create a monitor object, and to add a function call when a coverage wall is hit. In fact, with little work, it should be possible to integrate our compiler-based approach to any fuzzer implemented in C/C++.

3.6 Fuzzing

JEDIFUZZ focuses on leading the fuzzer to discover new regions of code and thus increase its coverage. The input mutation and the task to actually trigger bugs, is let to the underline fuzzer.

For this reason we decided to build JEDIFUZZ on top a of state-of-the-art fuzzer, which focuses on inputs mutation. This choice will allow us to evaluate if our approach is able to improve the performance of the state-of-the-art. We selected MOpt-AFL, which, thanks to its new mutation scheduling schema, is considered among the best existing fuzzers.

Chapter 4

Implementation

We will now give an insight of JEDIFUZZ implementation. First, we will present the data structure used to store the information about conditions; then we will go through the different components in Figure 3.1 one by one.

4.1 Data structure

A key element of JEDIFUZZ is how it stores and accesses the information about conditions. For each condition we want to keep track of which of its branches has been executed. We also want to specify if the condition should be enforced to take a specific branch. To achieve those two points, as shown in Figure 4.1, we store four bits of information for each condition. The first two bits are used to mark whether or not the true and the false branches have been executed at least once: a 0 indicates that the branch has never been executed; a 1 that the branch has been executed at least once. The other two bits are used to enforce the outcome of the condition: if the fourth bit is set, it means the condition has to be forced to assume the value stored in the third bit.

The runtime information are collected by the instrumentation, and accessed by the monitor when seleting which condition to disable; similarly, the monitor stores information about which condition to enforce to which value, and the instrumentation accesses it. This means that the data structure where we store all those information should be accessible by different processes: the monitor, and all the instances of the target binary. This can be achieved using shared memory, which consists in a region of memory shared between different processes. Having multiple processes accessing the same region of memory, one can argue that we need to implement some kind of synchronization to prevent race conditions. Nevertheless, those never happen. In fact, the monitor accesses the shared memory only when there are no running instances of the target binary.

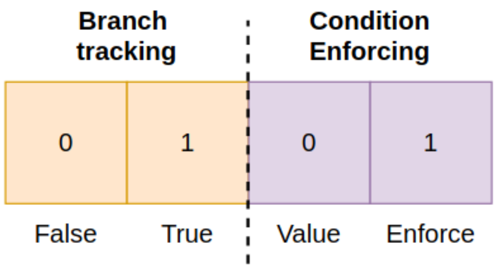

Figure 4.1: Representation of the encoding used to store a condition state on four bits. In this example, only the true branch has been executed yet. The enforce bit is set, specifying that, during the next executions, the condition has to be enforced to 0 (and thus to take the false branch).

As explained in section 3.3, the unique IDs assigned to the conditions start from zero and are incremental. An array is thus the ideal data structure to store checks information. This allows us to access the information of a given check by using its unique ID as index. Unfortunately there, is no basic type composed by four bits in C; we use $uint8\_t$ instead, and store two conditions in each element of the array. Given an ID, we use bitwise operators to extract both the correct element from the array, and the single bits of information from the latter. The number of conditions in the program corresponds to the number of instrumented locations. The size of the array is thus computed at compile time, and specified to both the monitor and the target binary through a dedicated environment variable.

4.2 Instrumentation

The goal of the instrumentation is:

1. to transform $switch$ statements into if statements;
2. to inject a call to a setup function;
3. to instrument all conditional branches.

LLVM provides a series of built in Analysis and Transform passes; one of those is $-lowerswitch$ [24]. This pass takes care of replacing switch instructions with branch instructions, which is exactly one of our goals.

The branch instrumentation needs to access the shared memory described in section 4.1; we thus need to modify the program so that the latter is initialized before being accessed. This is achieved through a Transform pass: it iterates over all functions looking for $main$, and injects a call to *__jedifuzz_setup* as its first instruction. This function opens the shared memory, and maps it so that it can be accessed by the branches instrumentation.

Conditional branches have to be instrumented. The instrumentation has to assign unique IDs to them, to add the logic to keep track of runtime information, and to enable the fuzzing engine to enforce their outcome. A Transform pass iterates over the instructions, detects conditional branches, and instruments them. Listing 4.1 shows how the LLVM IR of a conditional branch looks like. As it can be seen in Listing 4.2, the instrumentation modifies it by adding a call to *__jedifuzz_branch*. The injected function takes two arguments: the first is the value that was originally used as a condition by the branch instruction; the second is the unique ID assigned to this conditional branch. The branch instruction itself is modified to use the return value of *__jedifuzz_branch* as a condition, allowing the instrumentation to enforce which branch to take. Implemented in the JEDIFUZZ runtime library, *__jedifuzz_branch* uses the provided ID to access the array in the shared memory. It verifies whether or not the condition has to be enforced to a given value and, if it is the case, its return value is set accordingly. Then, depending on which value will be returned by the function, the corresponding branch will be marked as executed by setting its bit in the shared memory.

```
%20 = call i64 @strlen(i8* %19) #3
%21 = icmp eq i64 %20, 1
br i1 %21, label %22, label %32
```

Listing 4.1: Conditional branch LLVM IR.

```
%20 = call i64 @strlen(i8* %19) #3
%21 = icmp eq i64 %20, 1
%22 = call i1 @__tfuzz_branch(i1 %21, i32 0)
br i1 %22, label %23, label %33
```

Listing 4.2: Instrumented conditional branch LLVM IR.

One problem with IDs assignment is that the biggest abstraction in LLVM is a module. This means that our transform pass will be applied source file by source file. Since LLVM passes are stateless, we will thus end up having multiple identical IDs. Our solution is to use gllvm [14], which allows us to build a whole program into a single file. Thanks to it, we can extract the bytecode of our target into a single bytecode file, and apply the JEDIFUZZ transform pass to it.

In section 3.3, we explained how keeping track of the calling context would enable our tool to disable checks in a much more fine-grained way. Although this feature is not part of JEDIFUZZ yet, a possible approach toward calling context tracking is the Probabilistic Calling Context(PCC) [4], for which an LLVM implementation already exists [11]. The biggest challenge to face, in order to make JEDIFUZZ aware of the calling context, will thus be how to store and query the data related to the different PCCs.

4.3 Static analysis

To perform a static analysis of the Control Flow Graph (CFG), we need to extract it first. This is done through an Analysis pass that leverages the *GraphTraits* and the *DOTGraphTraits* classes defined in the LLVM project. Thanks to those, it is possible to encapsulate all the interesting information (e.g., the unique ID assigned to a given condition) into the CFG and dump the latter to a file.

Afterwards, all those files (one per function) are parsed from a python script and used to create directed graphs with the help of the $networkx$ library. The static analysis is performed on those graphs, where a node represents a basic block, and an edge represents a branch. A basic block contains one condition at most; all blocks containing a condition have two outgoing branches and a unique ID assigned to them (the condition one). For each branch of a basic block with an ID, the weight is computed according to one of the policies presented in section 3.4.

The weights are used to build a map associating them to the corresponding ID and branch. This map establishes a ranking between the instrumented conditions in the program, where an higher weight results into an higher rank. Once completed, the ranking is dumped to a file, so that it can be used to launch different campaigns without the need of performing the static analysis again. This is the same reason for which the CFGs are dumped files: it allows us to generate rankings with different policies without having to extract the CFGs multiple times.

4.4 Monitor

The monitor library provides all functionalities allowing the fuzzer to interact with the instrumented binary, and also the access to information collected at runtime. The library is implemented in C++, but it comes with a C interface to itself, making it possible to integrate the library both into C and C++ fuzzers.

It allows the fuzzer to create a monitor object, which — in its constructor — parses the conditions ranking from the file generated by the static analysis, and allocates the shared memory. When the fuzzer is stuck, it can query the monitor; as a result, the latter selects the conditions to disable, and updates the shared memory accordingly. To achieve this, the monitor iterates over the conditions in the shared memory, looking at the bits indicating which branch has been executed. For each condition there are three possibilities:

1. Both the true and the false branches are marked as executed; this condition is thus considered as fully explored.
2. None of the branches have been executed; this means that the condition has not been reached yet.
3. Only one of the two branches has the executed bit set.

Whenever a condition falls in the third case, the monitor looks at the weight of the non-executed branch. The one with the highest weight will be selected as the next branch to be enforced. Then, the monitor iterates over the shared memory a second time and updates the latter to enforce the previously selected branch to be taken. As explained in section 4.1, enforcing a condition consists in setting two of its dedicated bits in the shared memory.

AFL labels the most promising test cases it founds as favourite paths. Similarly to T-Fuzz [18] and Driller [22], JEDIFUZZ bases its inference that a coverage wall has been hit on the number of pending favourites paths. When this number is equal to zero, it means that the fuzzer has no more interesting input to fuzz. This can be considered as hitting a coverage wall, because without new favourite paths, the fuzzer is unlikely to produce new coverage. For this reason, JEDIFUZZ infers that a coverage wall has been hit when the number of pending favourite paths stayed at zero for 100'000 or more executions. The purpose of the additional threshold is to avoid transforming the program too early.

Chapter 5

Evaluation

In this section we will detail the results of the evaluation of the JEDIFUZZ current state. We will first discuss some limitations and how we addressed them, then we will proceed to present the evaluation setup and to discuss the results.

5.1 Preconditions

By transforming the program, JEDIFUZZ may introduce new bugs in the code. Thus, like all transformative fuzzers, it requires some kind of post analysis to filter out false positives. This project focuses on the fuzzing part of JEDIFUZZ, leaving the implementation of the post analysis to be done as future work. Our evaluation is thus limited by the fact that it is not reasonable to manually investigate all crashes found during multiple campaigns. As a consequence, whenever a bug is found in a transformed program, we do not know whether the bug is a true or a false positive.

Magma [12] is a ground truth fuzzing benchmark, consisting in a set of targets with front-ported bugs. The benchmark instruments each bug, so that it can asses when bugs are reached and triggered. As long as we consider only the bugs reported by Magma, using this benchmark allows us to solve the problem of false positives. But it would not be fair to evaluate our system by counting the bugs triggered in transformed programs. The perfect post analysis tool does not exist, and, given a true positive, a real world implementation may fail to find a crashing input for the original program. Hopefully, Magma monitors both reached and triggered bugs. A bug is considered reached if the code containing it has been executed at least once. Triggered bugs are bugs that have been reached and that resulted in an actual crash. As said, looking at the bugs triggered by our incomplete fuzzer does not make sense. On the other hand, observing reached bugs allows us to perform a reasonable evaluation of the current prototype of JEDIFUZZ. In fact, JEDIFUZZ focuses on increasing the code coverage of the fuzzing engine, leaving the task of mutating inputs and triggering bugs to the underlying fuzzer (i.e. MOpt-AFL). The number of reached bugs, and the time to reach them, can thus be a good indicator of how well JEDIFUZZ achieves its goal.

When computing the code coverage of a transformative fuzzer, we have to consider that mutating the program may unlock regions of code that are unreachable in the original binary. This present us with a dilemma: whether or not we should aggregate the code coverage of the transformed programs and of the original one. Unreachable code should generally be avoided, and in any case today compilers tend to optimize it away. Therefore, in our evaluation we assume that software does not contain unreachable code and we do aggregate the coverage over all program versions.

5.2 Goals

With the experiments presented in this section we want to assess how JEDIFUZZ improves the code coverage of a fuzzing engine. More precisely, we want to investigate:

1. if and how much, on average, JEDIFUZZ improves the code coverage of the target binary;
2. whether or not JEDIFUZZ makes it easier to find complex bugs (i.e, bugs behind hard-to-satisfy checks and/or hidden in deep execution paths), either by finding new bugs or by reaching bugs earlier.

A secondary goal of our evaluation is to compare the performance of the four ranking policies presented in section 3.4: Simple Blocks (SB), Recursive Blocks (RB), Simple Instructions (SI), and Recursive Instructions (SI). From now on, with JEDIFUZZ *SB* we indicate JEDIFUZZ under the Simple Blocks policy; with JEDIFUZZ *RB* we indicate JEDIFUZZ under the Recursive Block policy; and so on for the other policies.

We used MOpt-AFL as baseline fuzzer and compared its performance to the JEDIFUZZ one. Given our system built on top of it, MOpt-AFL was the logical choice.

5.3 Setup and Architecture

We evaluated five systems: *MOpt-AFL*, JEDIFUZZ *SB*, JEDIFUZZ *RB*, JEDIFUZZ *SI*, and JEDIFUZZ *RI*. As targets, we used the Magma versions of *libtiff (tiff_read_rgba_fuzzer)* and *libxml2 (libxml2_xml_read_memory_fuzzer)*. Six 24 hours long campaigns were run for each system-target combination, for a total of 60 campaigns and 1'440 CPU-hours of fuzzing. During our experiment, the Magma monitor utility was configured to pool information about reached and triggered bugs every five seconds. This means the results determine the first time a bug was reached or triggered with a precision of five seconds.

All campaigns have been run on an Ubuntu 18.04.3 LTS 64-bit machine, with an Intel® Xeon® Gold 5218 CPU (32 cores) and 64 GB of RAM.

5.4 Code Coverage

While fuzzing, AFL [29] — and all fuzzers built on top of it — keeps track of different statistics: one of those is the overall achieved coverage. To evaluate the performance of the different fuzzers, we extracted this statistic and aggregated the results of the different runs. The arithmetic means of the achieved code coverages is reported in Table 5.1. This table also includes the average improvement of the fuzzers over MOpt-AFL. To asses the results validity, we computed the standard deviations, which are shown in Table 5.2.

Fuzzer	**libtiff**		**libxml2**		**Average**
	Coverage	Improvement	Coverage	Improvement	Improvement
MOpt-AFL	10.57%	0.00%	21.95%	0.00%	0.00%
JEDIFUZZ SB	13.81%	31.79%	27.68%	26.56%	29.17%
JEDIFUZZ RB	13.90%	27.15%	27.73%	26.38%	26.77%
JEDIFUZZ SI	13.95%	29.52%	27.40%	25.88%	27.70%
JEDIFUZZ RI	13.92%	28.95%	27.73%	26.33%	27.64%

Table 5.1: Arithmetic means of the achieved code coverage, and average coverage improvement obtained over MOpt-AFL.

Fuzzer	**libtiff**	**libxml2**
MOpt-AFL	0.19	0.08
JEDIFUZZ SB	0.22	0.11
JEDIFUZZ RB	0.34	0.09
JEDIFUZZ SI	0.22	0.05
JEDIFUZZ RI	0.20	0.07

Table 5.2: Standard deviations of the coverages obtained by the systems over the six performed runs.

The results show that JEDIFUZZ consistently achieves a coverage improvement of 25-30% over MOpt-AFL, independently from the used policy. The overall small values of the standard deviations indicate that there is little variation between the different runs. This already suggests that the results are reliable; nevertheless, we performed the Mann-Withney U test to verify their statistical significance. This test aims to investigate whether or not two random samples were selected from different populations. For all policies, performing the test between them and MOpt-AFL, results in a *p-value* of around 0.0025. This confirms that the results in Table 5.1 are statistically significant.

We can thus conclude that, from a code coverage point of view, JEDIFUZZ reliably improves MOpt-AFL performance of 25-30%.

We underline that our results have been collected through a set of 24 hours long campaigns. Thus, for future work, it may be interesting to investigate how this coverage difference evolves over longer campaigns.

5.5 Reached Bugs

To compare fuzzers performances at reaching bugs, we need a metric encoding the expected time to reach a bug. Because of the random nature of fuzzing, performance can vary between campaigns sharing the same configuration. And, in some cases, the fuzzing engine may fail to reach a bug that was reached in other campaigns. In general, it is better to have a system consistently finding a bug after one hour of fuzzing, than a system finding the same bug after ten seconds but only once every ten campaigns. When defining the expected time to reach a bug, this should be taken into account.

The authors of Magma proposed to model the time to reach a bug as a random variable X with exponential distribution. The expected time to reach a bug is then defined as the expected value $E(X)$ of that variable. Let N be the total number of runs, M the number of runs that reached the bug, T the duration of a run, and $\bar{t}$ the arithmetic mean of the measured times to reach the bug; then $E(x)$ is computed as follows.

$$E(X) := \frac{M \times \bar{t} + (N - M) \times \frac{T}{\lambda_t}}{N} \quad \text{where} \quad \lambda_t = \ln(\frac{N}{N - M})$$

Using the above formula, we computed the expected time to reach the different bugs. Table 5.3 presents those findings, pairing them with the percentage of campaigns that reached each bug. We notice that all fuzzers reached the majority of the bugs during the first few minutes of fuzzing. JEDIFUZZ starts transforming the program only after that the first coverage wall is hit. As a consequence, up to that moment, JEDIFUZZ behaves exactly as MOpt-AFL. Thus, it is not surprising that the expected time to reach bugs found early is almost identical for all fuzzers.

Unfortunately, only the four bugs underlined in blue in Table 5.3 took more than ten minutes to be reached. For *AAH009* the performances of the fuzzers are roughly the same. The same holds for *AAH016*, except that JEDIFUZZ SI and JEDIFUZZ RI found the bug only in one of the six runs, instead of twice like the other fuzzers. As a consequence, the expected time to reach the bug is significantly higher. Given the small amount of executed runs, it is hard to determine if this difference is due to the randomness inherent to fuzzing, or directly to the policies underperforming.

The results for the remaining two bugs are more relevant, with JediFuzz performing in general better than MOpt-AFL. *AAH010* is particularly interesting: for this bug, JediFuzz obtains significantly better results under all policies.

Four bugs are not enough to draw any kind of conclusion. Nevertheless, JediFuzz obtained better results than MOpt-AFL in 50% of the cases, while performing as well as the latter in the remaining 50%. This suggest that our system can find bugs hidden in deep execution paths in a more reliable way. To further assess this trend, more evaluations — against more targets — should be performed.

	Bug ID	**MOpt-AFL**		**JediFuzz SB**		**JediFuzz RB**		**JediFuzz SI**		**JediFuzz RI**	
		Reached	Avg. time	Reached	Avg. time	Reached	Avg. time	Reached	Avg. time	Reached	Avg. time
libtiff	AAH009	33%	42h	33%	44h	33%	44h	33%	41h	33%	42h
	AAH010	16%	112h	50%	23h	66%	14h	50%	20h	83%	9h
	AAH011	100%	15s	100%	15s	100%	15s	100%	15s	100%	15s
	AAH015	100%	20s	100%	19s	100%	24s	100%	27s	100%	30s
	AAH016	33%	42h	33%	43h	33%	44h	16%	112h	16%	113h
	AAH018	100%	41m	100%	21m	100%	29m	100%	49m	100%	1h
	AAH020	100%	10s	100%	10s	100%	10s	100%	10s	100%	10s
	AAH022	100%	20s	100%	19s	100%	25s	100%	27s	100%	30s
libxml2	AAH024	100%	15s	100%	42s	100%	50s	100%	1m	100%	2m
	AAH026	100%	15s	100%	15s	100%	15s	100%	15s	100%	15s
	AAH029	100%	15s	100%	8m	100%	55s	100%	16m	100%	7m
	AAH031	100%	20s	100%	43s	100%	49s	100%	52s	100%	1m
	AAH032	100%	15s	100%	15s	100%	15s	100%	15s	100%	15s
	AAH034	100%	15s	100%	23s	100%	25s	100%	25s	100%	33s
	AAH035	100%	15s	100%	33s	100%	40s	100%	40s	100%	50s
	AAH037	100%	10s	100%	10s	100%	10s	100%	10s	100%	10s
	AAH041	100%	15s	100%	15s	100%	15s	100%	15s	100%	15s

Table 5.3: Percentage of runs reaching the bug, and the expected time to reach it. Blue rows indicate bugs found after the first ten minutes of fuzzing. Bugs that were not reached by any system are not reported.

5.6 Ranking Policies

Looking at Table 5.1 and Table 5.3, we have the impression that the four different policies do not have a significant difference in performance. To have a better insight of how different policies work, we computed the arithmetic mean of the number of transformations performed by each policy. The results are shown in Table 5.4.

We notice that during the *libtiff* campaigns, the program was transformed more times than during the *libxml2* ones. Nevertheless, according to the data in Table 5.1, those additional transformations did not reflect into higher coverage improvement. This suggests that the performance

Fuzzer	libtiff	libxml2
JEDIFUZZ SB	37	12
JEDIFUZZ RB	44	12
JEDIFUZZ SI	43	13
JEDIFUZZ RI	48	13

Table 5.4: Arithmetic means of the number of transformations performed by the different policies.

improvement brought by JEDIFUZZ is mainly due to the first transformations. As further investigation of this hypothesis, we selected one sample run of *libxml2* for each policy. Then, for each sample run, we computed the coverage increase obtained after each of the first ten transformations. The results, presented in Table 5.5, confirm the hypothesis: each transformation tends to be less effective than the previous one. This is not surprising considering that, at each transformation, JEDIFUZZ selects the most promising condition to enforce. Thus, the more transformations are done, the less promising the newly selected conditions will be.

Those results suggest that combining multiple policies may be a better approach than using the same one over the whole fuzzing campaign. Exploring new policies, and changing ranking method every few transformations, is thus a promising direction for future work.

Transformation #	JEDIFUZZ sb	JEDIFUZZ rb	JEDIFUZZ si	JEDIFUZZ ri
1	0.0	0.07	0.06	0.01
2	0.13	0.15	0.02	0.17
3	0.01	0.02	0.13	0.02
4	0.0	0.06	0.02	0.12
5	0.02	0.0	0.02	0.02
6	0.01	0.0	0.02	0.03
7	0.0	0.02	0.01	0.0
8	0.02	0.02	0.0	0.01
9	0.02	0.0	0.0	0.01
10	0.02	0.0	0.0	-

Table 5.5: Coverage increases — in percentage — obtained after the first ten transformations during one sample run against *libxml2*.

Chapter 6

Related Work

Over the years, improvements to different aspects of fuzzing where proposed. JEDIFUZZ addresses the problem of coverage wall; thus, in this section, we will only focus on related work trying to solve the same problem.

6.1 Feedback based fuzzing

Coverage-guided fuzzers like AFL [29] and libFuzzer [23] use code coverage as a progress metric. Additionally, they monitor the code coverage contribution of the inputs and use it as a feedback. If an input generates new coverage, they deduce it has passed a check that was previously failing: that input is thus prioritized for future mutations.

The main limitation of feedback based approaches is that they are based on "hindsight": to generate inputs passing a check, the check has to be passed first. In other words, they make it easier to pass a check a second time, but they do not help to pass it in the first place. Additionally, they have difficulties at handling checksums and other values computed on the fly. AFL pioneered the field of fuzzing with this approach, which is still used by many fuzzers — including JEDIFUZZ— under the hood. Nevertheless, a feedback based approach alone is not sufficient to solve the coverage wall problem.

6.2 Symbolic and concolic execution based fuzzing

Symbolic execution reduces passing a check to solving a set of constraints. More precisely, it encodes all checks along a path into a series of constraints, and tries to solve the resulting logical formula. Concolic execution combines symbolic execution with concrete execution. Several fuzzers [22][28] leverage symbolic execution in different ways. Among those, the closest to JEDIFUZZ— and transformative fuzzers in general — is Driller. When the fuzzer hits a coverage wall, Driller uses concolic execution to find an input passing the failing check.

Although symbolic execution reliably succeed in finding interesting inputs, it is subject to path explosion and, as a consequence, it does not scale. JEDIFUZZ, inversely, does not have any scaling problem.

6.3 Taint analysis base fuzzing

Taint analysis can identify the relations between the input and the logic of the program. Taintscope [27] leverages taint analysis to detect checksums and bypass them. Other works, like VUzzer [19] and Angora [5], use taint analysis in conjunction with other methods. VUzzer uses static and dynamic analysis to extract control-flow and data-flow features. The firsts are used to prioritize deep paths; the seconds to determine where and how to mutate the input. Angora sees finding an interesting input as a search problem. It leverages taint analysis to relate conditional statements to the byte offsets that undergo them. It then forwards the information to a search algorithm based on gradient descent, which tries to find an input satisfying the provided constraint.

While providing interesting results, taint analysis lacks in flexibility and is heavy weight, a problem that is even accentuated when paired with other form of analysis. JEDIFUZZ is more flexible and, thanks to its instrumentation, can handle different checks with low runtime overhead.

6.4 Transformative fuzzing

T-Fuzz [18] was the first work proposing to mutate not only the inputs, but also the program. When the fuzzer hits a coverage wall, T-Fuzz generates a new version of the program by disabling one of the failing checks. To avoid false positives, each crash found is forward to a crash analyser. The latter leverage symbolic analysis to verify whether the crash is a true positive (i.e. if it can be triggered in the original program) or a false positive.

T-Fuzz has shown to perform better than the symbolic execution and the taint analysis based approaches. Nevertheless, it has some limitations.

- It can address only if statements, lacking in flexibility.
- It transform the program in a suboptimal way: it blindly selects and negates a conditional jump instruction that was not negated yet, without optimizing the choice.

- Its program transformation procedure, involving runtime patching, is somehow inelegant and requires to store multiple versions of the program.
- Although symbolic execution was removed from the fuzzing routing, the crash analyser is still using it.

JEDIFUZZ addresses and solves the first three limitations that are listed above. In fact, it can handle multiple types of check. It allows the fuzzing engine to dynamically and efficiently transform the program. And, when a coverage wall is hit, it carefully selects the condition to enforce, trying to optimize the coverage outcome of each transformation.

Chapter 7

Future work

The version of JEDIFUZZ presented in this document is a first prototype. As a consequence, there are different directions for future work, with multiple improvements and extensions that could be integrated into the system. Those are summarized here below.

7.1 Post analysis

Like all transformative fuzzers, JEDIFUZZ may introduce false positives. The usual approach to deal with them is to have some kind of post analysis investigating crashes and filtering out false positives. As pointed out in section 3.2, the current version of our system does not include a real post analysis procedure. The implementation of the latter should be prioritized over all future work. This is fundamental to make JEDIFUZZ a complete fuzzer, and to be able to compare it with other transformative fuzzers.

7.2 Instrumentation

A key motivation behind the idea of JEDIFUZZ is the flexibility that the compile time instrumentation can bring. The current implementation does not take full advantage of its compiler-based approach, thus leaving space for improvements. In the following we will try to point out some of the possible improvements.

Switch statements

switch statements are currently transformed into *if* statements through the *lowerswitch* LLVM pass. This approach simplifies both the design and the implementation of the fuzzer, but it probably introduces overhead. It would be interesting to design an alternative solution which handles *switch* statements without lowering them, and then compare its performance with the current implementation.

Loops

Loops are common in programs and they all come with at least one check: the loop condition. Right now a loop condition, like all other conditions, is considered to be fully explored once both its true and false branches are executed. This means that, as long as at least one iteration is performed, a loop condition is considered fully explored after a single execution. This is not the case, since the code behaviour may change depending on the number of performed iterations. Another direction for future work can thus be to extend JEDIFUZZ to handle loops in a smarter way, taking into account the number of iterations.

Calling context

In section 3.3, we pointed out how making JEDIFUZZ aware of the calling context will allow it to enforce conditions in a much more fine-grained way. A possible approach to this, as proposed in section 4.2, is to use PCCs [4].

7.3 Static analysis

In order to enhance flexibility, JEDIFUZZ static analysis supports different ranking policies — as described in section 3.4.

All four policies implemented in the current version of the system focus on improving code coverage. This follows the philosophy of coverage-guided fuzzing, which leverage code coverage as a performance metric. Nevertheless, not all policies have to be coverage based. A valid option may be to follow the approach proposed by ParmeSan [17] — a sanitizer-guided fuzzer — and to create some sanitizers-based policies. Sanitizers are tools that instrument binaries to detect security violations and dangerous behaviours at runtime. As a consequence, their instrumentation can be seen as an indicator that the code is more prone to bugs. To make a practical example, a possible policy could be to use the number of locations instrumented by AddressSanitizer [21] — a sanitizer detecting memory errors — as a weight.

Ranking are used to always select the most promising condition to enforce. This implies that each time we select a condition, we expect it to be less promising than the previous one (see section 5.6). Solutions to reduce this trend, like changing policy every few iterations, are yet another interesting unexplored path.

7.4 Other fuzzers

We built our system on top of the state-of-the-art fuzzer MOpt-AFL. Nevertheless, as underlined in section 3.2, one of the strengths of JEDIFUZZ is that integrating it into any C/C++ fuzzer requires little work. A wide performance evaluation of different fuzzers extended with JEDIFUZZ can thus be of interest for future work. Such investigation would provide a much better insight on JEDIFUZZ strengths and weaknesses.

Chapter 8

Conclusion

We developed JEDIFUZZ, a whitebox compiler-based transformative fuzzer. When the fuzzing engine hits a coverage wall, JEDIFUZZ efficiently transforms the program to bypass a carefully selected hard-to-satisfy check. To maximize the coverage outcome of each transformation, our system selects the check to bypass according to a qualitative ranking. This ranking is obtained through a one time static analysis that supports different policies.

The whole system is designed to be easily integrated into any C/C++ fuzzer. In fact, the logic to transform programs is wrapped into a library. As a result, the work required to extend an arbitrary fuzzer with JEDIFUZZ is reduced to the insertion of a few lines of code.

In the evaluation, we compared JEDIFUZZ with MOpt-AFL, the state-of-the-art fuzzer on top of which we implemented our system. The results show that JEDIFUZZ consistently improves the code coverage over MOpt-AFL of 25-30%. Also, it seems that our system is able to reach bugs hidden in deep execution paths in a more reliable way. Further evaluations are needed to validate those trends, but the preliminary results remain promising.

We want to underline that the version of JEDIFUZZ presented in this report consists in a first prototype of the system, and is thus far from being fully optimized. As discussed in chapter 7, there are several interesting directions for future work aiming to improve the performance of JEDIFUZZ. Nevertheless, concerning future work, the priority should be given to the implementation of a proper crash analyser.

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. “Control-flow integrity principles, implementations, and applications”. In: *ACM Transactions on Information and System Security (TISSEC)* 13.1 (2009), pp. 1–40.

[2] Dave Aitel. “An introduction to spike, the fuzzer creation kit”. In: *Presentation slides, Aug* 1 (2002).

[3] Roberto Baldoni, Emilio Coppa, Daniele Cono D’elia, Camil Demetrescu, and Irene Finocchi. “A survey of symbolic execution techniques”. In: *ACM Computing Surveys (CSUR)* 51.3 (2018), pp. 1–39.

[4] Michael D Bond and Kathryn S McKinley. “Probabilistic calling context”. In: *Acm Sigplan Notices* 42.10 (2007), pp. 97–112.

[5] Peng Chen and Hao Chen. “Angora: Efficient fuzzing by principled search”. In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 711–725.

[6] CVE Details. *Security Vulnerabilities Published In 2019*. URL: `https://www.cvedetails.com/vulnerability-list/year-2019/vulnerabilities.html` (visited on 06/29/2020).

[7] Michael Eddington. “Peach fuzzing platform”. In: *Peach Fuzzer* 34 (2011).

[8] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. “Jump over ASLR: Attacking branch predictors to bypass ASLR”. In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2016, pp. 1–13.

[9] Google. *Honggfuzz*. URL: `https://github.com/google/honggfuzz` (visited on 06/13/2020).

[10] Google. *OSS-Fuzz: Continuos Fuzzing for Open Source Software*. URL: `https://github.com/google/oss-fuzz` (visited on 06/12/2020).

[11] Adrian Herrera. *Probabilistic Calling Context*. URL: `https://github.com/adrianherrera/probabilistic-calling-context` (visited on 06/25/2020).

[12] HexHive. *Magma: A Ground-Truth Fuzzing Benchmark*. URL: `https://hexhive.epf.ch/magma` (visited on 06/26/2020).

[13] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. “Dta++: dynamic taint analysis with targeted control-flow propagation.” In: *NDSS*. 2011.

[14] SRI International's Computer Science Laboratory. *Whole Program LLVM in Go.* URL: `https://github.com/SRI-CSL/gllvm` (visited on 06/23/2020).

[15] Jun Li, Bodong Zhao, and Chao Zhang. "Fuzzing: a survey". In: *Cybersecurity* 1.1 (2018), p. 6.

[16] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. "Fuzzing: State of the art". In: *IEEE Transactions on Reliability* 67.3 (2018), pp. 1199–1218.

[17] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "ParmeSan: Sanitizer-guided Greybox Fuzzing". In: *29th {USENIX} Security Symposium ({USENIX} Security 20).* 2020.

[18] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. "T-Fuzz: fuzzing by program transformation". In: *2018 IEEE Symposium on Security and Privacy (SP).* IEEE. 2018, pp. 697–710.

[19] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. "VUzzer: Application-aware Evolutionary Fuzzing." In: *NDSS.* Vol. 17. 2017, pp. 1–14.

[20] Juha Röning, M Lasko, Ari Takanen, and R Kaksonen. "Protos-systematic approach to eliminate software vulnerabilities". In: *Invited presentation at Microsoft Research* (2002).

[21] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. "AddressSanitizer: A fast address sanity checker". In: *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12).* 2012, pp. 309–318.

[22] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution." In: *NDSS.* Vol. 16. 2016. 2016, pp. 1–16.

[23] LLVM Team. *libFuzzer - a library coverage-guided fuzz testing.* URL: `https://llvm.org/docs/LibFuzzer.html` (visited on 06/28/2020).

[24] LLVM Team. *The LLVM Compiler Infrastructure.* URL: `https://llvm.org/` (visited on 06/14/2020).

[25] PaX Team. *PaX address space layout randomization (ASLR).* 2003. URL: `http://pax.grsecurity.net/docs/aslr.txt` (visited on 06/11/2020).

[26] Arjan van de Ven and Ingo Molnar. *Exec Shield.* 2004. URL: `https://static.redhat.com/legacy/f/pdf/rhel/WHP0006US_Execshield.pdf` (visited on 06/11/2020).

[27] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection". In: *2010 IEEE Symposium on Security and Privacy.* IEEE. 2010, pp. 497–512.

[28] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. "{QSYM}: A practical concolic execution engine tailored for hybrid fuzzing". In: *27th {USENIX} Security Symposium ({USENIX} Security 18).* 2018, pp. 745–761.

[29] Michal Zalewski. *American Fuzzy Lop.* 2015. URL: `http://lcamtuf.coredump.cx/afl` (visited on 06/12/2020).