Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Improving Code Instrumentation to Circumvent Roadblocks in Generator-based Fuzzing

Bachelor Thesis

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von: Adrian Schiller
geboren am: 18.11.1997
geboren in: Berlin

Gutachter/innen: Prof. Dr. Lars Grunske
M.Sc. Yannic Noller

eingereicht am: 28.02.2023 verteidigt am: .....

Abstract

This bachelor thesis presents JQF-Smart, an extension of the Zest generator-based fuzzer [23] that provides a solution for the ”magic value problem”, specifically for string comparisons. The magic value problem in coverage-guided fuzzing occurs during the execution of the tested program when specific values of the input (so-called ”magic values”) are required to execute certain hard-to-reach code paths. It is a persistent challenge in the field of fuzzing since it can be a roadblock preventing the fuzzer from triggering behaviour leading to potentially vulnerable code. Previous research provided several approaches to address this problem, including [14, 15, 6].

JQF-Smart employs code instrumentation to generate targeted feedback for string comparison methods that cause the magic value problem. It introduces novel adaptive input generators that allow the fuzzer to dynamically switch between different string generation strategies at specific locations of the input. JQF-Smart iteratively discovers the magic values based on the targeted feedback of previously generated inputs by adjusting the string generation strategy accordingly.

To evaluate the fuzzing performance of JQF-Smart, we conducted an empirical study using three open-source Java programs. Our results demonstrate considerable improvements in the effectiveness of the Zest fuzzer when using JQF-Smart, specifically in covering hard-to-reach parts of the code. Overall, the study demonstrates the positive impact that the modified instrumentation paired with the employment of adaptive generators has on the performance of generator-based fuzzing. Our approach has shown potential in addressing the magic value problem in coverage-guided fuzzing, with a focus on Java’s string comparison methods.

Contents

1 Introduction **3**

2 Background **6**

2.1 ASM 6

2.2 Property Testing 8

2.3 Coverage-Guided Fuzzing 11

2.3.1 Overview 11

2.3.2 Semantic Fuzzing 12

2.4 laf-intel 14

3 Related Work **19**

3.1 Magic Values 19

3.2 Structure Aware Fuzzing 21

4 Approach **23**

4.1 Overview 23

4.2 Adaptive Generators 23

4.3 Smart Guidance 26

4.4 String Compare Coverage 29

5 Evaluation **35**

6 Conclusion **40**

1 Introduction

In the software development process, testing the software is a crucial step in order to ensure its functionality and security. It has become an increasingly difficult task over the past few years with the increasing complexity of the tested systems. A common approach to detect software vulnerabilities is coverage-guided fuzzing (CGF) [7], which has gotten a lot of popularity with the introduction of several testing tools, such as those listed in [28, 22, 24, 23, 16]. Coverage-guided fuzzers provide the software under test (SUT) with randomly generated inputs and use feedback in the form of code coverage to measure the behaviour of the SUT. Inputs that trigger new behaviour, i.e. inputs that cover previously undetected program paths, are saved and mutated to iteratively execute more functionalities and find vulnerabilities. Generator-based fuzzers also known as semantic fuzzers, such as the Java-based fuzzer Zest [23] use generators in the style of QuickCheck [10] to generate inputs based on a predefined grammar.

However, a common limitation of traditional CGF tools can be of reaching certain branches that are guarded by complex conditions (potentially leading to unexpected behaviour), such as multiple-byte comparisons [25]. These conditions can be satisfied by specific values of the input, which are known as ”magic values”. A simple example of the magic value problem is a string comparison, in which the input must be equal to a specific value, which will be referred to as ”magic string”. Figure 1.1 illustrates this concept with a simple code example. It shows a string comparison using the `equals` method from the Java class `String` to determine whether the string `input` is equal to the value `"BADSTRING"`. If the `input` happens to match that value, the if-statement leads to unexpected behaviour in line 2. Otherwise, the intended functionality is executed in line 5.

```
1    if (input.equals("BADSTRING")){
2        /* buggy code */
3    }
4    else {
5        /* good code */
6    }
```

Figure 1.1: Code Snippet to illustrate the magic value problem

Since generator-based fuzzers often rely on some sort of dictionary to generate strings, it is impossible to reach the true branch if the magic string is not contained in the dictionary. Randomly generating a required string is an arduous task due to the large search space. The probability P of randomly generating the string value `"BADSTRING"` can be calculated by the size A of the alphabet raised to the negative power of the length L of the string.

Length of `"BADSTRING"`:	$L = 9$
Size of the ASCII alphabet:	$A = 128 = 2^8$
Probability:	$P = A^{-L} = 2^{8^{-9}} = 2^{-72}$

Furthermore, the coverage feedback does not provide information on how similar the compared strings are. For example the input values `"123abcxyz"`, `"BADabcxyz"` and `"BADSTRxyz"` would all cover the same branch in our example (line 5). Only, if the string exactly matches the value `"BADSTRING"` the true branch (5 would be covered, thus changing the feedback that the fuzzer receives.

We introduce JQF-Smart, a modified version of the Zest fuzzer integrated into the fuzzing framework JQF [22]. JQF-Smart addresses the magic value problem for strings, by extending the instrumentation of the SUT to provide more accurate coverage feedback related to certain string comparison methods. We further introduce adaptive input generators that dynamically adapt the string generation strategy, based on the feedback from the program execution. Specifically, our fuzzer keeps track of all string comparisons and periodically targets a string comparison that has never been evaluated to be true for any input. To evaluate the targeted comparison, JQF-Smart uses a custom method that computes the return value while also counting, how many characters of the compared values are equal. When

the fuzzer is targeting a comparison, the adaptive generator produces the string character by character at a particular input position, rather than retrieving it from a dictionary, which leads to the evolutionary discovery of the magic value. We refer to this string generation strategy as *search-based string generation*.

In this thesis, we investigate the impact of our modified instrumentation and the integration of adaptive generators on the performance of fuzzing. Our research question is:

What is the impact of the modified instrumentation combined with the integration of adaptive generators on the performance of generator-based fuzzing?

Since we assume that JQF-Smart will have an overhead compared to Zest, due to the more detailed feedback and the adaptive input generation, we suggest a combination of Zest and JQF-Smart to enhance the performance of JQF-Smart. We evaluate JQF-Smart's performance by comparing it to Zest. To measure the performance of both tools, we use the achieved coverage on three benchmark programs. We expect that the employment of JQF-Smart increases the code coverage that was achieved by Zest, through the exploration of branches that were impossible to reach by Zest.

The paper is structured as follows: Chapter 2 provides the required background information on the instrumentation software used by Zest, property testing, generator-based fuzzing and another approach that addresses the magic value problem. Chapter 3 covers related work that inspired our approach to implementing JQF-Smart. Chapter 4 discusses the implementation approach. Chapter 5 presents the evaluation methodology and results. Chapter 6 concludes our thesis and answers the research question.

2 Background

2.1 ASM

JQF uses the Java bytecode manipulation framework ASM [8] in order to instrument the classes that are being tested. To understand the instrumentation process, we have to take a look at the ASM library.

In order to manipulate classes ASM provides three building blocks. The abstract class `ClassVisitor` provides methods to mirror the bytecode instructions, i.e. an instruction `xxx` is represented by the method `visitXxx`. The `MethodVisior` and the `FieldVisitor` classes are used to represent methods and fields, respectively and provide the necessary methods. Any Java class can be represented by the corresponding sequence of methods. The visitor classes delegate these method calls to another instance of themselves and act as a kind of filter. To manipulate a given class, the programmer provides extensions for the visitor classes that serve as adapters by overriding certain methods. The `ClassReader` class parses the byte code. It has an `accept` method, which takes a `ClassVisitor` instance as an argument and calls the corresponding methods to represent the byte code. The `ClassWriter` class is an extension of `ClassVisitor` that receives the delegated method calls and generates a byte array that contains the compiled class. Figure 2.1 illustrates the process of byte-code manipulation, using the adapter `ClassAdapter`.

JQF provides a class adapter that uses a method adapter to rewrite certain instructions. The essential instructions that are rewritten are method instructions (i.e. invocations), jump instructions and table switch instructions (which correspond to switch-statements in the source code). The adapter adds a static method call at certain locations in the code in order to monitor every method invocation

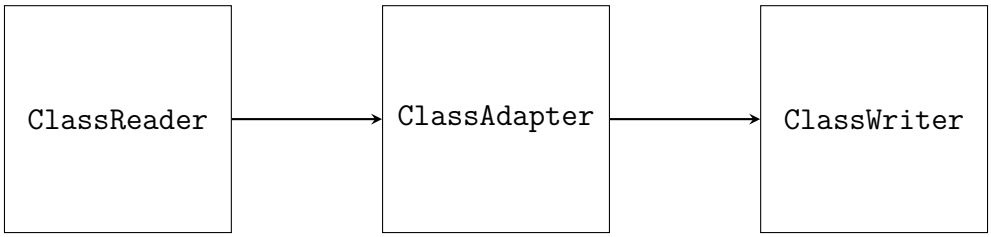

Figure 2.1: ASM byte code manipulation process. From [8].

and each branch of jump and table switch instructions. Each instruction gets assigned an instruction ID to differentiate the various coverage points that are being monitored. Figure 2.2 shows the overridden method `visitMethodInsn` from the method adapter (simplified version to illustrate the concept). The arguments for this method are the owner (i.e. the class), the name and the description (i.e. a string containing information about input and output types) of the method that is invoked by the bytecode instruction. The adapter uses an instance `mv` of the `MethodVisitor` class to call the actual methods corresponding to the instructions. Before the original method is called (line 6) we go through the following sequence of steps. For every instruction, the adapter pushes a unique instruction ID and the value `0` onto the stack (lines 3 and 4). Then, the static method `LOGJUMP` is called which takes the two integer values on top of the stack as input arguments. Additional instruction IDs get allocated when instrumenting jump instructions and table switch instructions and the second argument of the `LOGJUMP` method refers to the branch.

```
1 @Override
2 public void visitMethodInsn(String owner, String name, String desc) {
3     addBipushInsn(mv, incAndGetInstructionId()); // push instruction ID
4     mv.visitInsn(ICONST_0);                      // push value 0
5     mv.visitMethodInsn("FastCoverageSnoop", "LOGJUMP", "(II)V");
6     mv.visitMethodInsn(owner, name, desc);
7 }
```

Figure 2.2: JQF instrumentation method `visitMethodInsn`

Figure 2.3 illustrates the instrumentation. It shows, how the code snippet shown in figure 1.1 would look after instrumenting the byte code. The instrumentation

```
1     FastCoverageSnoop.LOGJUMP(1, 0);
2     if (input.equals("BADSTRING")){
3         FastCoverageSnoop.LOGJUMP(2, 0);
4         /* buggy code */
5     }
6     else {
7         FastCoverageSnoop.LOGJUMP(2, 1);
8         /* good code */
9     }
```

Figure 2.3: Example of instrumented code by JQF

adds invocations of the method `FastCoverageSnoop.LOGJUMP` to the code and generates the instruction IDs. The `equals` method (line 2) is assigned instruction ID 1 and is tracked in line 1. The jump instruction of the if-statement is assigned instruction ID 2. The true-branch is logged in line 3 and the false-branch in line 7. The `FastCoverageSnoop.LOGJUMP` method uses the instruction ID and an integer representing the branch as input arguments.

2.2 Property Testing

The concept of property-based testing was proposed by Claessen and Hughes [10] with QuickCheck, an automatic testing tool for Haskell programs. They introduced a testing technique in which the programmer defines certain properties, that have to hold true for all kinds of randomly generated inputs. It is also possible to define conditional properties that have to hold true under certain circumstances. The idea is to define pre- and postconditions, and thus the behaviour of the test program, by the properties. Finding an unfulfilled property after the execution of an input means that the test program performs exceptional behaviour, i.e. a malfunction is discovered. To test the properties of programs with complex inputs, QuickCheck also introduced input generators that produce syntactically valid inputs by construction. Using pseudo-random choices these generators hierarchically assemble structured inputs.

The Java library JUnit QuickCheck [13] provides a testing framework for Java

```
1    @Property
2    public void squareRootTest(double x) {
3        assumeTrue(x >= 0);              // Precondition     x ≥ 0
4        double root = Math.sqrt(x);
5        assertEquals(root * root, x);    // Postcondition    x = root²
6    }
```

Figure 2.4: JUnit QuickCheck example: Properties of the square root function from `java.lang.Math`

that integrates property-based testing into the well-known testing library JUnit [2]. With certain test drivers, the programmer defines the properties that the program under test must satisfy. Pre- and Postconditions are specified with methods from the `Assume` and `Assert` classes, respectively. Figure 2.4 shows an example of a JUnit QuickCheck property test. The goal is to validate the functionality of `Math.sqrt` which is supposed to calculate the square root of a given argument. The test method executes `Math.sqrt` with randomly generated inputs of the type `double` (line 4). The precondition is defined in line 3 and serves to only test the functionality for positive inputs since `Math.sqrt` cannot return complex numbers. After the execution, the functionality is checked by comparing the input to the square of the calculated root (line 5).

To allow for the testing of more complex programs with longer inputs, JUnit QuickCheck provides a generator-building interface that follows the same principles as QuickCheck. Figure 2.5 shows a simple example of a generator, which generates an XML document based on that interface. The generator relies on pseudo-random choices (lines 16, 22, 28) for producing primitive types (e.g. `int` or `boolean`) and constructs a syntactically correct XML document with nested nodes through recursive function calls (line 18). Any input can also be presented as a sequence of choices that determine the specific input values. For the parts of the inputs that will be interpreted as Strings, the generator relies on a dictionary from which randomly selected Strings are taken. This is referred to as *dictionary-backed* string generation and can also take place in a separate generator used by the main generator. In practice, the dictionary can be configured with a set of keywords to produce a specific type of XML document, such as files of the POM format.

```java
public class XmlDocumentGenerator extends Generator<Document> {

    private String[] dict = {"hello", "node_1", "node_2", "root"};

    @Override
    public Document generate(SourceOfRandomness random) {
        Element root = genElement(random)
        return new Document(root);
    }
    private Element genElement(SourceOfRandomness random, int depth) {
        // Generate an Element with a random name
        String name = makeString(random);
        Element node = new Element(name);
        if (depth < MAX_DEPTH) {
            // Randomly generate a random number of child nodes
            int n = random.nextInt(MAX_CHILDREN);
            for (int i = 0; i < n; i++) {
                node.appendChild(genElement(random, depth+1);
            }
        }
        // Maybe insert text inside the element
        if (random.nextBoolean()) {
            node.addText(makeString(random));
        }
    }
    private String makeString(SourceOfRandomness random) {
        // Randomly choose a String from the dictionary
        int index = random.nextInt(dict.length);
        return dict[index];
    }
}
```

Figure 2.5: Simplified generator for XML documents. Adapted from [23]

2.3 Coverage-Guided Fuzzing

2.3.1 Overview

Coverage-Guided Fuzzing (CGF) is a widespread technique in the field of automatic software testing that was popularized by the C testing framework AFL [28] and is now used in various testing applications [15, 22, 23, 28, 21, 20, 14, 6].

CGF uses grey-box analysis in order to gain feedback on the behaviour of the tested program during execution. The software under test (SUT) is fed with randomly generated inputs in order to find bugs or execute program behaviour that was not triggered by previously generated inputs. Interesting inputs that cover new paths in the execution of the program are saved and mutated to iteratively uncover more and more of the program's behaviour. The goal is to explore the control flow graph of the program to search for unexpected behaviour. In order to measure the behaviour of a program code coverage is utilized. That usually considers which program paths have been executed and how many times each branch has been reached, i.e. the hit count of a branch. The total coverage of the program consists of the set of branches reached by all the generated inputs and the highest hit count of each branch.

The guidance determines based on the coverage feedback, which inputs are going to be saved for further mutation. To illustrate this process, figure 2.6 shows the general workflow of coverage guidance, i.e. the fuzzing loop. The fuzzing loop is initiated with a set of seed inputs or with a randomly generated input. First, an input is used to execute the SUT. If the execution caused a program crash or hang, the respective input gets saved in a list of failures and the next input gets executed. Otherwise, the guidance updates the total coverage with the coverage of that execution, i.e. the run coverage. If new branches have been discovered or the hit count of a branch was increased, that input will be saved to the fuzzing queue. To produce new inputs, the guidance takes the next input of the queue and performs low-level mutations on that input, such as bit flips or changing byte values.

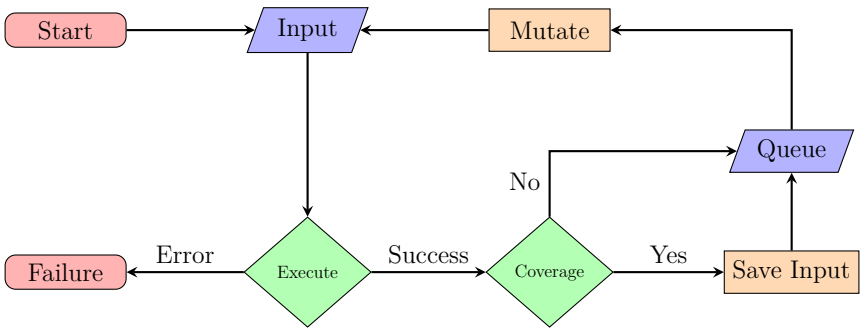

Figure 2.6: Workflow diagram for coverage-guided fuzzing.

2.3.2 Semantic Fuzzing

JQF is a coverage-guided fuzzing framework for Java [22]. The project makes use of JUnit QuickCheck to perform property-based testing on the inspected program, allowing an easy way for practitioners to write custom test drivers for their respective programs. JQF also provides interfaces to allow researchers to implement new CGF algorithms or specialize existing ones, e.g. by altering the guidance, making changes to the fuzzing heuristics or applying new methods to track the coverage. Notable fuzzers that have been implemented in JQF include AFL [28], PerfFuzz [16] and Zest [23]. The JQF-Zest project combines CGF with specialized input generators from JUnit QuickCheck in order to produce more complex syntactically correct inputs and explore the semantic analysis stage of the SUT.

Parametric Generators Zest transforms a random input generator from the JUnit QuickCheck interface into an equivalent *parametric generator*. These parametric generators utilize untyped byte streams (i.e. parameter sequences) which are equivalent to the aforementioned choice sequences, to produce the inputs. Each random choice gets turned into a deterministic choice that is controlled by the parameters and each parameter sequence translates to a syntactically valid input. Low-level mutations of bytes in the parameter sequence result in high-level structural mutations of the actual input. To illustrate this concept we look at the following sequence of calls from the `SourceOfRandomness` instance after the invocation of the `generate` method from the previously discussed generator from figure 2.5:

Call	→ result	Context	
`nextInt(dict.length)`	→ 3	`String name = "root"`	(line 12)
`nextInt(MAX_CHILDREN)`	→ 1	`int n = 1`	(line 16)
`nextInt(dict.length)`	→ 1	`String name = "node_1"`	(line 12)
`nextInt(MAX_CHILDREN)`	→ 0	`int n = 0`	(line 16)
`nextBoolean()`	→ true		(line 22)
`nextInt(dict.length)`	→ 0	`node.addText("hello")`	(line 23)
`nextBoolean()`	→ false		(line 22)

These choices represent a path in the generator that leads to the following XML document:

```
d = <root><node_1>hello</node_1></root>
```

To produce the exact same output with the equivalent parametric generator, the random choice generator reads its results from a byte stream which represents the choice sequence `x = [3, 1, 1, 0, true, 0, false]`.
The Zest guidance handles inputs in the form of byte arrays and performs mutations to produce new inputs on that array, which preserves the required input structure. For instance, could a mutation of the bytes corresponding to the third choice of `x` lead to the choice sequence `x' = [3, 1, 2, 0, true, 0, false]` and produce the mutated document:

```
d' = <root><node_2>hello</node_2></root>
```

As mentioned, in practice the parameter sequence consists of untyped bytes so that mutations that alter the actual structure of an input (e.g. the number of children from a node) don't lead to type conflicts.

Zest-Algorithm The purpose of Zest is to explore the field of semantically valid inputs to find more and more paths and reveal bugs in the semantic analysis stage of the test program. To achieve this goal Padhye et al. extended the basic CGF process we discussed earlier in this section with two main ideas. The first one is the incorporation of random input generators by turning them into parametric input generators.

The second idea involves the handling of the results after the execution of the SUT with a specific input. The Zest algorithm not only tracks the total coverage but also the coverage that was reached by semantically correct inputs, also known as the *valid coverage*. To execute the inputs Zest uses a JUnit QuickCheck test driver written by the practitioner. Malfunctions and bugs in the program can either be detected by violations of properties defined in the assertion methods of the framework or by exceptions produced by the program itself. In addition to the assertions, the user can also define assumptions, which are properties related to the semantic validity of the inputs. The Zest algorithm differentiates between three types of results after executing an input. If an input finds a bug in the program it is labelled `FAILURE` and saved separately. Inputs that don't discover bugs are either labelled `INVALID` - meaning that they are semantically incorrect, as they caused an assumption violation - or `SUCCESS` when there was no violation. After that, inputs that cause new code coverage are saved to the fuzzing queue for later mutation. To further explore the space of semantically correct inputs, the coverage of valid inputs is additionally tracked and inputs that cause new valid coverage are also saved and mutated more often.

Coverage To track the code coverage, JQF maintains a map with keys referring to coverage points in the code and their respective hit counts as values. To track the coverage points, JQF uses ASM to add static snooping methods to the bytecode as we showed in section 2.1. At runtime, these methods invoke logging methods from the coverage instance and the instruction ID gets added to the branch value to produce a unique key. Every time a logging method is called, it increases the hit count of that key. If new keys get added to the map or the hit count of a key reaches a new peak, it means that the related input executed new program behaviour.

2.4 laf-intel

An extension of the widely used fuzzing tool AFL is LAF-INTEL [3], which addresses the magic value problem by code modifications in the SUT. The project provides LLVM passes that change the tested software at compile time increasing the number of coverage points that can be tracked. The essential point is that

complex comparisons, such as the comparison between two strings, which affect the control flow are split into several more simple ones that, when combined, form the original comparison. The LAF-INTEL team implemented this concept with three different LLVM passes.

Split-Compares The "split-compares-pass" targets comparison operators and splits comparisons that compare multiple bytes into single-byte comparisons. To illustrate this, consider the code snippets shown in figure 2.7. The left-hand side shows a comparison between the variable `input` and a byte value, that would lead to "buggy code" if the value of input was equal to the four-byte value `0xabad1dea`. The right-hand side splits the comparison into four one-byte comparisons. The "split-compares-pass" also transforms comparisons in the program so that it only contains the following comparisons (`==, !=, <, >`) and all signed comparisons are replaced with their unsigned equivalent.

```
1 if(input == 0xabad1dea) {
2     /* buggy code */
3 }
```

```
1 if (input >> 24 == 0xab) {
2   if ((input & 0xff0000) >> 16 == 0xad) {
3     if ((input & 0xff00) >> 8 == 0x1d) {
4       if ((input & 0xff) == 0xea) {
5           /* buggy code */
6 } } } }
```

Figure 2.7: LAF-INTEL Split-Compares-Pass. From [15]. On the left is the original code and on the right is the modified code.

Split-Switches The next modification we are going to look at is the "split-switches-pass". An easy approach for transforming switch-statements would be to rewrite them into a series of if-statements and then apply the split-compares-pass. `laf-intel` provides a more elegant solution that avoids redundant code in cases where the deciding byte of two cases is not at the last position. To understand the concept, we take a look at the code snippets in figure 2.8. The code on the left shows a switch-statement with two cases. Values required to reach these cases differ in the next-to-last byte and are equal in the last byte. The code on the right shows how LAF-INTEL transforms the switch-statement. This is a similar concept as in

the split-compares-pass, but the comparison of the last byte takes place before the next-to-last bytes are compared (line 3). Only if `input` has the value `0xff` as its last byte, the two cases are evaluated (lines 4 and 7).

```
1  switch(input) {
2   case 0x11ff:
3     /* case 0x11ff */
4     break;
5   case 0x22ff:
6     /* case 0x22ff */
7     break;
8   default:
9     /* handle default */
10 }
```

```
1  if(input >> 24 == 0) {
2    if((input & 0xff0000) >> 16 == 0x00) {
3      if((input & 0xff) == 0xff) {
4        if((input & 0xff00) >> 8 == 0x11) {
5          /* case 0x11ff */
6          goto after_switch;
7        } else if((input & 0xff00) >> 8 == 0x22) {
8          /* case 0x22ff */
9          goto after_switch;
10 } } } }
11 default_case:
12     /* handle default */
13 after_switch:
```

Figure 2.8: LAF-INTEL Split-Switches-Pass. From [15]. On the left is the original code and on the right is the modified code.

Compare-Transform Lastly, we look at the "compare-transform-pass". This pass rewrites the code for the C-methods `strcmp` and `memcmp`, which return 0 if the arguments contain the same values. To visualize this pass, consider the code snippet from figure 1.1, again. The equivalent C code with the `strcmp` method is shown in figure 2.9. Figure 2.10 shows this if-condition after the transformation. Like before, the concept is to split complex comparisons into several simple comparisons, so the two strings are compared character by character in a number of nested if-statements. The compare-transform-pass only works when one of the strings is a literal, since the characters and the number of comparisons have to be known at compile time. For `memcmp` the size parameter (how many bytes are compared) has to be known at compile time.

Applying these transformations will increase the computational cost of running the code due to the additional conditional jumps. However, the benefit is that of getting much more detailed feedback on the similarity of the two values because the coverage gets tracked for every if-condition. For example the input of `"BADSTRxyz"`

```
1    if (!strcmp(input, "BADSTRING")) {
2        /* buggy code */
3    }
```

Figure 2.9: String Comparison in C code. Adapted from [15]

```
1    if (input[0] == 'B') {
2      if (input[1] == 'A') {
3        if (input[2] == 'D') {
4          if (input[3] == 'S') {
5            if (input[4] == 'T') {
6              if (input[5] == 'R') {
7                if (input[6] == 'I') {
8                  if (input[7] == 'N') {
9                    if (input[8] == 'G') {
10                     if (input[9] == 0 ) {
11                         /* buggy code */
12   } } } } } } } } } }
```

Figure 2.10: LAF-INTEL Compare-Transform-Pass. Adapted from [15]

would have higher coverage (lines 1 to 7) than the input of `"BADabcxyz"` (lines 1 to 4).

Our approach is inspired by the compare-transform-pass used to transform the `strcmp` method. In contrast to `laf-intel`, JQF-Smart replaces several Java string comparison methods in the compiled software in order to execute a character-wise comparison and count the matching characters.

3 Related Work

3.1 Magic Values

The research community has developed multiple fuzzers that are capable of finding magic values that satisfy certain conditions and hence lead to coverage of particular hard-to-reach branches guarded by those conditions. A very common approach to address this problem is a white-box analysis of the system under test (e.g. taint tracking or symbolic execution) [14, 11, 25, 24, 9]. However, these fuzzers tend to have relatively high overhead and, thus slow down the fuzzing speed. Other, more lightweight techniques targeting the same problem are for example the aforementioned LAF-INTEL or REDQUEEN [6]. We will take a closer look at REDQUEEN and CONFETTI because they introduce interesting techniques that we adopted in our approach.

CONFETTI In their 2022 publication of CONFETTI [14] Kukucka et al. proposed a technique to boost the efficiency of concolic fuzzing guidances. The core idea of CONFETTI is the combination of grey-box fuzzing with white-box analysis and the novel mechanism of global hinting. It is implemented as an extension for the semantic fuzzer Zest.

To maintain the fuzzing efficiency, CONFETTI divides the fuzzing campaign into three processes and executes them in parallel. The first process is the grey-box fuzzer, which is responsible for producing new inputs (i.e. parameters for the generator), running the test program and tracking the coverage. Inputs that covered new branches are not only saved to the fuzzing queue but also delegated to the second process - the coordinator. The coordinator delegates the collected

inputs to a white-box analyzer (KNARR), where the inputs are executed on a separate instrumented version of the SUT. The purpose of the analyzer is taint tracking and collecting of constraints. Taint tracking is a technique to analyse a program, by "tainting" certain parts of the input (e.g. adding labels to bytes of the parameter stream). KNARR uses taint tracking to analyze how the data flow of the program works. Since they are propagated through data flow, the taints of a specific variable at a specific state reveal which parts of the input are responsible for its value. KNARR also collects constraints from the input and returns them together with the collected taint flows back to the coordinator. To help cover new branches, an SMT solver is used to compute the magic values by solving the negated constraints. The established procedure, *local hinting*, involves inserting the values at their respective location within the original input. The newly produced inputs are then passed to the fuzzer and added to the queue. The fuzzer also has access to all the magic values collected so far and during input mutation, inserts magic values at random positions within any input, a technique referred to as *global hinting*.

As a part of their evaluation, Kuckuka et al. compared their program with and without global hinting and came to the conclusion that the new strategy can help to find inputs with new coverage and thus improve the effectiveness of the fuzzer. We apply the simple concept of global hinting in our approach in order to make use of discovered string values as much as possible.

Redqueen While many other fuzzers rely on complex white-box analyses to find magic values, Aschermann et al. [6] proposed a fairly simple technique that sufficiently manages to mimic taint analysis and symbolic execution. Their publication of REDQUEEN is built on the concept of *input-to-state correspondence*, implying that the states of a program strongly depend on the input. In other words, the values of parts of the input (or mutations of those values) can be found in certain variables during the program execution.

The core idea is to look at comparison instructions and track the compared values back to the responsible part of the input. If one side of the comparison could be mapped to a part of the input, that part gets mutated to the value of the other side of the comparison. In practice values from the input often arrive at a state in

a modified form, hence the program also tries typical byte encodings schemes when replacing the value(s), and for certain comparisons such as "greater than" a bit can be added or subtracted to the replaced value.

In some cases, the inputs can be very large and the comparison can't be nailed down to a specific position of the input, e.g. inputs that largely consist of bytes with the same value. To prevent too many unnecessary mutations that cause a big overhead, REDQUEEN applies a simple technique in which it tries to replace as many bytes as possible with randomly generated values, without changing the execution path compared to the original input. The authors of REDQUEEN refer to those inputs as "colourized inputs", which can be viewed as a very low-budget version of taint tracking. During the execution of a colourized input, the comparison instructions are observed again and the responsible positions of the input can be narrowed down to certain parts through input-to-state correspondence.

In our implementation, we also make use of input-to-state correspondence, by applying it to strings and employing an adjusted way of colourized inputs to track down which parts need to be mutated.

3.2 Structure Aware Fuzzing

One of the challenges, we faced was the input representation, since our fuzzer needed to support dictionary-backed and search-based string generation. The latter consumes more bytes from the parameter stream than the former (i.e. one integer choice per character and one to determine the length of the string instead of only one choice for the index in the dictionary). A major concern when exercising the search-based string generation was to maintain the structure of the input, hence we decided to split these parameters from the main parameter stream.

With BEDIVFUZZ [20] Nguyen et al. introduced a technique to separate the choice sequences (i.e. parameter streams) consumed by the generators used in Zest into *structure-choices* and *value-choices*. As the names suggest, they differentiate between choices of which mutations on, lead to changes in the structure and in the values of inputs, respectively. The concept behind BEDIVFUZZ is a strategy that performs mutations that change the input structure to explore new branches and mutations that preserve the structure in order to explore specific branches with

different input values. For example, in the generator from figure 2.5 the choices that determine how many children a node has (line 16) and whether a node has text inside (line 22) are structural. In line 28 is a value choice, since it determines the value of a string by choosing the index for the dictionary.

BeDivFuzz introduces the interface `SplitInput` on which the input representation of our approach, as well as some of the input handling functionality, is based. Further, BeDivFuzz implemented an extension of the parameter stream reader and pseudo-random choice generator used by Zest, that utilizes two choice sequences, which also inspired our approach.

4 Approach

This project presents JQF-Smart, an automatic testing tool combining the power of semantic fuzzing with the refinement of string comparisons proposed as a part of LAF-INTEL [15] to address the magic value problem. In this section, we take a look at our approach to building JQF-Smart.

4.1 Overview

Our fuzzer is capable of solving certain string comparisons by finding the necessary "magic" string values. We implemented a specialized class adapter to instrument the SUT and an extension of the coverage used by JQF in order to keep track of the progress made in a string comparison, i.e. the number of matching characters or the *string-coverage*. We further propose a novel mechanism to dynamically adapt the string generation strategy and applied this mechanism to the `XmlDocumentGenerator` shipped with JQF. Finally, we built an extension of the Zest guidance that periodically targets unsolved string comparisons and applies search-based string generation at the responsible position of the input.

4.2 Adaptive Generators

To utilize search-based string generation at input locations, we provide a convenient way to change an existing input generator that uses a dictionary-backed string generator to produce strings. It can be used by any input generator that relies on dictionary-backed string generation to produce strings. To achieve this, we implemented an adaptive string generator that can dynamically switch between

different kinds of generation.

The adaptive generator keeps track of how many strings have been produced while generating an input, in order to target certain locations of the input. We will refer to those locations as *string IDs*. When generating a string, it can either apply dictionary-backed generation, search-based generation or insert a specific string at a given string ID. The adaptive generator produces inputs from two different parameter streams. The first parameter stream provides the bytes to determine an integer value, that is used to choose a string from the dictionary (line 14). The second parameter stream is responsible for search-based string generation, which means that the bytes are used to determine the length of the string and then each character. Consider the `makeString` method of the XML document generator from figure 2.5. The method in figure 4.1 shows, how the adaptive generator switches the string generation on the fly. If the string ID of the currently generated string matches the targeted string ID, the method `makeRandomString` is invoked (line 4). This method uses the secondary random source, which is part of the random instance and contains the parameters for the search-based string generation.

Besides the parameter streams the adaptive generator also has access to a hash map that maps string IDs to actual strings. This is used to insert certain strings at certain locations in an input. If the current string ID is contained in the map, the string that is mapped to that key will be returned (line 9): This functionality is utilized for local and global hinting as well as for colourizing inputs. Each time a string is generated with another strategy than dictionary-backed generation, an integer choice from the primary parameter stream is consumed to maintain the structure of the input.

To represent the two parameter sequences the implemented guidance uses the class `SmartInput`, which is based on the `SplitInput` interface provided by BeDivFuzz. These inputs consist of a primary and a secondary input. The primary input contains the parameters, that are used to control the choices of the input generator. The secondary input contains the parameter sequence that is used for the search-based string generation. Instances of `SmartInput` can also contain a string ID at which the search-based string generation should be applied and a hash map that maps string IDs to string values in order to insert specific strings at specific input locations. Considering the example from before (section 2.3.2)

```
1    private String makeString(SmartSourceOfRandomness random) {
2        String string;
3        // search-based generation:
4        if (STRING_ID == TARGET_STRING_ID) {
5            random.nextInt();
6            string = makeRandomString(random.secondary);
7        }
8        // insert a specific string:
9        else if (HINT_MAP.containsID(STRING_ID)) {
10           random.nextInt();
11           string = HINT_MAP.getString(STRING_ID);
12       }
13       // dictionary-backed generation:
14       else {
15           int index = random.nextInt(dict.length);
16           string = dict[index];
17       }
18       STRING_ID++;
19       return string;
20   }
```

Figure 4.1: Adaptive string generation.

with the choice sequence: `x = [3, 1, 1, 0, true, 0, false]`, producing the document:

```
d = <root><node_1>hello</node_1></root>
```

In comparison, consider a `SmartInput` instance, containing the primary sequence `x`, the secondary sequence: `x.secondary = [5, 'w', 'o', 'r', 'l', 'd']` and the target string ID: `2`. The adaptive generator would produce the following document:

```
d = <root><node_1>world</node_1></root>
```

4.3 Smart Guidance

JQF-Smart instruments string comparison methods, such as `String.equals` or `String.contains` to keep track of all the *solved* and *unsolved* string comparisons (i.e. comparisons that returned true for any input and comparisons that have been false for all inputs respectively) throughout the fuzzing campaign. The algorithm periodically aims to solve a specific comparison by changing the mutation strategy at a related location of the input and applying search-based string generation at this location. The fuzzing algorithm of our implementation is built into the established Zest algorithm and consists of a few steps that lead to the string-solving process. The workflow diagram in figure 4.2 presents an extended version of the CGF workflow diagram previously discussed (figure 2.6). It helps to understand how the fuzzing process of JQF-Smart works and how it is integrated into the fuzzing process of Zest. When our process is triggered the following steps are executed.

Get Candidates From the set of *unsolved keys* (i.e. coverage points that represent unsolved string comparisons), a random key is chosen which we refer to as our *target key*. By looking at their coverage, we choose a number of inputs from the set of saved inputs that reached the target key and select them as a set of candidate inputs. To prevent the algorithm from trying to solve the same comparison for the same input over and over again, the target key is also saved in each of the candidate inputs.

Setup Target Queue After acquiring the candidate inputs, the next step is to track, which string ID from the generator is responsible for the comparison of the target key. To trace a comparison back to a string ID we applied a brute force algorithm, that relies on the concept of input-to-state correspondence and an adapted version of colourizing inputs. For every string ID of a candidate input, the fuzzer produces a colourized input by inserting a specific colourizing string at that location, respectively. After the SUT was executed with the colourized, the fuzzer receives feedback on whether the colourizing string reached the targeted comparison. If the comparison was reached, a copy of the candidate input containing information on the currently colourized string ID is saved into a queue of target inputs, which will be the starting point to solve the targeted string comparison.

String Solving The string-solving process begins after the target queue has been set up and it is essentially a nested fuzzing loop that tries to solve the specific target comparison.

The fuzzer mutates the string parameters of the next input from the target queue. To increase the efficiency of the string mutation, we adapted the heuristics to mutate parameter sequences. Our string mutation method includes the following features. When the original input provided string coverage, it means that the length of the generated string is correct, so the first four bytes (providing the length of the string) will not be mutated. We made sure that every mutation changes exactly one character. The probability of mutating the wrong characters increases, the more characters are discovered, hence the number of mutations decreases with the amount of discovered characters.

After mutating an input the adaptive generator applies the search-based string generation at the targeted string ID of that input to produce the arguments and the SUT gets executed. Detected malfunctions and bugs are saved and the algorithm selects the next input from the target queue. Assuming there was no unexpected behaviour during the execution, the string coverage of that run contains the amount of matched characters and updates the set of solved keys. When we look at the ”String Coverage” node of the workflow diagram (4.2) there are three options. If the string coverage did not track any new progress, we go back to the target queue and continue mutating. If the string coverage recognizes that new characters

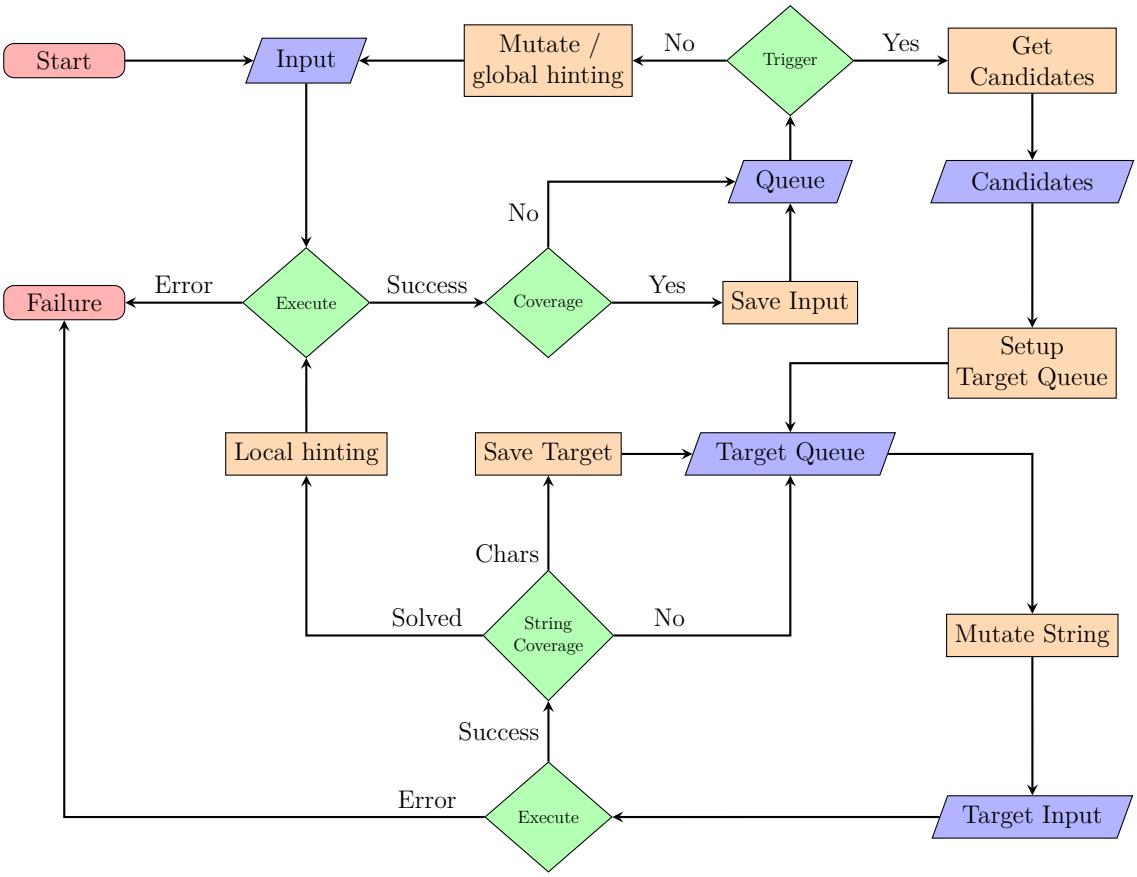

Figure 4.2: Workflow diagram for fuzzing with JQF-Smart.

were discovered, the mutated input gets saved to the target queue. Lastly, if the target key occurs in the set of solved keys after a run, it means that the targeted string comparison has been solved. In that case, our fuzzer applies local hinting by inserting the newly discovered magic string into the respective positions of the original candidate inputs that were saved to the target queue. To prevent it from getting stuck in an endless loop, e.g. when the targeted comparison is not solvable, the string-solving loop also terminates after a certain amount of trials without finding a new matching character and the process continues with the main fuzzing loop by mutating inputs from the queue of saved inputs. During the main fuzzing loop, the guidance uses a coin flip to apply global hinting by inserting magic strings at random positions of random inputs.

4.4 String Compare Coverage

JQF-Smart instruments the SUT to provide more detailed feedback regarding string comparisons. We extended the existing instrumentation from JQF to replace all instructions that invoke certain string comparison methods, with instructions to invoke respective static methods. The static methods delegate the string, the comparison argument(s) and the instruction ID to a special coverage instance that produces feedback and evaluates the comparison. Namely, the replaced methods are:

```
String.equals(Object)
String.equalsIgnoreCase(String)
String.contains(CharSequence)
String.contentEquals(CharSequence)
String.contentEquals(StringBuffer)
String.endsWith(String)
String.startsWith(String)
String.startsWith(String, int)
```

We particularly extended the JQF method adapter to override the method from figure 2.2. A simplified version of our implementation is shown in figure 4.3. After the usual snooping method is executed, we check if the instrumented method belongs to the set of comparison methods. In that case, the instruction to invoke the corresponding static method from the class `SmartCoverageSnoop` is inserted. The input arguments are the string, then the argument(s) for the original method and then the instruction ID, so the method descriptor gets adapted, e.g. the descriptor for the `equals` method gets changed from `"(LJava/lang/String;)Z"` to `"(LJava/lang/String;LJava/lang/String;I)Z"`.

To illustrate the instrumentation, consider the code snippet from figure 1.1 once again. Figure 4.4 shows, how the decompiled byte code looks like, after instrumenting it with our implementation. The `equals` method, comparing the input string to the value `"BADSTRING"`, is replaced by an invocation of the static method `SmartCoverageSnoop.LOGEQUALS` which returns the same boolean value. Since the string `input` and `"BADSTRING"` are already on the stack, the instrumentation just

```
@Override
public void visitMethodInsn(String owner, String name, String desc) {
    int iid = incAndGetInstructionId()
    addBipushInsn(mv, iid);
    mv.visitInsn(ICONST_0);
    mv.visitMethodInsn("FastCoverageSnoop", "LOGJUMP", "(II)V");
    if (owner.equals("java/lang/String") {
        switch (name) {
            case "equals":
                addBipushInsn(mv, iid);
                mv.visitMethodInsn("SmartCoverageSnoop", "LOGEQUALS",
                    "(LJava/lang/String;LJava/lang/String;I)Z");
                break;
            case "contains";
                addBipushInsn(mv, iid);
                mv.visitMethodInsn("SmartCoverageSnoop", "LOGCONTAINS",
                    "(LJava/lang/String;LJava/lang/CharSequence;I)Z");
                break;
            // Handle the remaining string comparison methods
            // that should be replaced
            default:
                mv.visitMethodInsn(owner, name, desc);
        }
    }
    else {
        mv.visitMethodInsn(owner, name, desc);
    }
}
```

Figure 4.3: JQF-Smart instrumentation method `visitMethodInsn`

```
1     FastCoverageSnoop.LOGJUMP(1, 0);
2     if (SmartCoverageSnoop.LOGEQUALS(input, "BADSTRING", 1){
3         FastCoverageSnoop.LOGJUMP(2, 0);
4         /* buggy code */
5     }
6     else {
7         FastCoverageSnoop.LOGJUMP(2, 1);
8         /* good code */
9     }
```

Figure 4.4: Example for instrumented code by JQF-Smart.

pushes the last instruction ID (which belongs to the comparison method) onto the stack and inserts the instruction to invoke the new method, which takes the elements from the stack as input arguments.

At runtime, these arguments get passed to the respective method of our custom coverage instance, where the return value gets computed and the feedback is obtained. Figure 4.5 illustrates, how these methods work on the example of the `equals` method. If the string comparison already has been solved (or in the particular case of the `equals` method if the provided argument `anObject` is not an instance of the `String` class) the method just returns the original method that was replaced. To keep track of the solved and unsolved comparison, every time a comparison evaluates to false, the associated key is saved to a set of unsolved keys (lines 16 and 27) and when it is solved the key gets removed from the set of unsolved keys (lines 12 and 23) and put into a set of solved keys (lines 11 and 22). In case the guidance is currently providing colourized inputs to the SUT, the string coverage checks if the colourizing string reached the target comparison 7. The feedback, delegated to the guidance instance, contains information on whether the colourizing string reached the target comparison and what the hit count of the target key was at that moment. That way, the string coverage feedback can be limited to a particular invocation of a particular string comparison for the currently colourized input location.

During the search-based string generation, our coverage uses a custom comparison method to execute the targeted comparison which is the core idea of the

implementation. Our custom method counts the matching characters and computes the return value of two strings. We implemented the method `logStringCmp` which is used for all string comparisons, so certain adaptations of the input arguments are applied. For instance, type transformations from `CharSequence` into `String` or if we want to know if a string starts with a prefix, only the first characters (depending on the prefix length) of the string are passed to `logStringCmp`.

Figure 4.6 shows, how our comparison method is constructed. The private field `MATCHES` is used to track the number of characters with the same value. If the compared values are not of the same length, the method evaluates to false and `MATCHES` stays at its default value `0`. Otherwise, we go through all the elements side by side and count, how many of them contain the same value (line 6). If the amount of matched characters is less than the length of the string, the counted value (increased by `1`) is passed to the field `MATCHES` and the method also evaluates to false. Increasing the value by one serves the purpose of differentiating between comparisons where the strings have the length and comparisons where they do not have the same length. If the counted matches are equal to the length of the strings it means that both strings are completely equal, thus the method returns the value true. In this case, it is not necessary to track the counted matches, since the comparison is solved and the coverage key is put into the set of solved keys.

Considering the if-statement from figure 4.4. The input value of `"BADabcxyz"` would result in a string coverage feedback with the `MATCHES` field containing the value `4`, while the input of `"BADSTRxyz"` would result in the `MATCHES` field containing the value `7`. When a string comparison was solved with the search-based string generation, the coverage instance provides the fuzzer with the magic value to use for the hinting strategies.

```
1  public boolean logEquals(String str, Object anObject, int iid) {
2      if (SOLVED_KEYS.contains(iid) || !(anObject isinstanceof String)) {
3          return str.equals(anObject);
4      }
5      else if (TARGET_KEY == iid) {
6          if (COLORIZED_INPUT) { //
7              CHECK_COLOR(str, (String) anObject);
8          }
9          else if (STRING_SOLVING) {
10             if (logStringCmp(str, (String) anObject)) {
11                 SOLVED_KEYS.add(iid);      // add to solved keys
12                 UNSOLVED_KEYS.remove(iid); // remove from unsolved keys
13                 return true;
14             }
15             else {
16                 UNSOLVED_KEYS.add(iid);    // add to unsolved keys
17                 return false;
18             }
19         }
20     }
21     if (str.equals(anObject)) {
22         SOLVED_KEYS.add(iid);      // add to solved keys
23         UNSOLVED_KEYS.remove(iid); // remove from unsolved keys
24         return true;
25     }
26     else {
27         UNSOLVED_KEYS.add(iid);    // add key to unsolved keys
28         return false;
29     }
30 }
```

Figure 4.5: Custom method to control the string coverage feedback for the `equals` method.

```
1     private int MATCHES = 0;
2     private boolean logStringCmp(String str1, String str2) {
3         int len = str2.length();
4         if (len == str1.length()) {  // compare lengths
5             int matches = 0;
6             for (int i = 0; i < len; i++) {
7                 if (str1.charAt(i) == str2.charAt(i)) {
8                     matches += 1;     // count matching characters
9                 }
10            }
11            if (matches < len) {
12                MATCHES = matches+1;  // log the string coverage
13                return false;
14            }
15            else {
16                return true;
17            }
18        }
19        return false;
20    }
```

Figure 4.6: Custom string comparison method evaluating the comparison and counting the matching characters.

5 Evaluation

For our evaluation, we compared the performance of the fuzzers JQF-Smart and JQF-Zest. A common metric used to measure the performance of coverage-guided fuzzing is the code coverage achieved over time. To apply our adaptive XML document generator, we conducted our experiments on three XML processing Java programs that are shipped with the JQF project:

Apache Ant [4]
Ant automates software build processes, including compiling and testing applications.

Apache Maven [18]
Maven is a dependency manager that uses an XML file of the POM format to define project configurations, dependencies, and build settings.

Apache Tomcat [26]
Tomcat uses XML documents to configure a web server and deployed web applications can be configured with web.xml files. Our evaluation tests the functionality to configure web applications.

We ran our experiments on the computing server "gruenau7" of the Humboldt-Universität zu Berlin [1]. The technical details are:

Model: Asus ESC4000
CPU: Xeon 6354
CPUs / Cores / Threads: 2 / 36 / 72
Clock Frequency: 3,6GHz
RAM: 1TB

Following the suggestions of the Zest research paper [23] we ran the two fuzzers over a total of three hours, as Zest does not find a significant amount of new paths during the last hour (i.e. generally less than 1%). To gain statistically valid results, we ran each campaign with 20 repetitions, as also suggested in the Zest paper. We used the terminal multiplexer tmux [17] to run 20 fuzzing campaigns simultaneously.

To evaluate the performance of our fuzzer, we first fuzzed the benchmarks for two hours with Zest and employed JQF-Smart only for the last hour in order to further explore the collected input space. We figured that this would be an efficient way to use our tool since we expected diminishing fuzzing performance after Zest reached a coverage plateau so a potential overhead of JQF-Smart would not have a great impact on the fuzzing efficiency.

During the evaluation process, we observed discrepancies in the number of covered branches between the two tools, when running them on the same set of seed inputs, with Zest covering up to 10% more branches. To overcome this issue for our evaluation, we implemented another extension of the guidance used by Zest which saves the input arguments produced by the generator and the respective time stamps. We argue that this alteration does significantly affect the efficiency of Zest since it saves a small number of files over a three-hour fuzzing campaign and does not impose a significant runtime overhead.

A third extension reproduces the fuzzing campaigns by running all the collected input arguments of both tools independently and logs the achieved coverage. We used the coverage data from the reproduced campaigns and the collected timestamp information from the original fuzzing campaigns to map the achieved coverage over time for both tools.

We plotted our results in a Jupyter Notebook [3] using the python libraries Pandas [19], Seaborn [27], Matplotlib [5] and Numpy [12]. Our tool, the results of the experiments and the scripts to run the experiments are available at `https://gitlab.informatik.hu-berlin.de/sillerad/jqf-smart`.

Results The figures 5.1, 5.2 and 5.3 show the average coverage achieved (in coverage points) across 20 repetitions as a function of fuzzing time, for Ant, Maven and Tomcat, respectively. The blue and orange plots represent the average coverage of our JQF-Smart and Zest, respectively and the shaded areas refer to the 95%

confidence intervals. For all three benchmarks, the rate at which new branches are covered by Zest is very high in the beginning and slows down over time to a point where new branches a discovered very slowly. This observation implies that the alteration we made to Zest in order to log the generated arguments, does not compromise the validity of our results. The coverage plots of Ant and Tomcat are relatively stable, whereas Maven's coverage plot exhibits more fluctuations and displays a less prominent coverage plateau. This could mean that our alteration has more impact on the fuzzing efficiency when fuzzing Maven or that the server we used to conduct the experiments was occupied during the fuzzing campaigns for that benchmark. To find out more about that, we would have to run the experiments in a closed environment.

Since we used Zest to collect inputs for JQF-Smart, both coverage plots of all three benchmarks display high similarity during the first two hours. At that time the coverage rate of Zest has slowed down significantly and our fuzzer is activated. The plots show that the coverage rate increases sharply after JQF-Smart is employed and surpasses the overall coverage of Zest by a large margin. Specifically, JQF-Smart reached at least 10% more coverage points on average for all three benchmarks. The confidence intervals of the coverage plots indicate that JQF-Smart is not as robust as Zest in achieving new code coverage but this does not affect the validity of our results, since the discrepancies of the coverage plots are considerably high on all three benchmarks. Our tool discovered an average of 3.7 strings for Ant with a maximum of 5, an average of 2.75 strings for Maven with a maximum of 3, and an average of 4.05 strings for Tomcat with a maximum of 8 during the search-based string generation. This indicates that the application of local and targeted hinting in our tool was able to reach branches that were impossible to reach by Zest.

It should be noted that we did not analyze the bugs found during the fuzzing campaigns. We assume that the code coverage serves as a valid metric to measure the performance of fuzzers since it represents how many program paths have been executed. Obviously, the purpose of software testing is to find bugs but the idea of coverage-guided fuzzing is to explore the behaviour of the tested software.

Overall, our experiments show that JQF-Smart outperforms Zest in the last hour of the fuzzing campaigns in terms of effectiveness and efficiency in covering new program paths. On all three benchmark programs, JQF-Zest reached at least

10% more coverage points on average at the experiments. We assume that the additional coverage points of JQF-Smart represent hard-to-reach branches that could not be covered by Zest. However, the validity of our results depends solely on code coverage as a metric and we did not analyze the bugs found during the fuzzing campaigns.

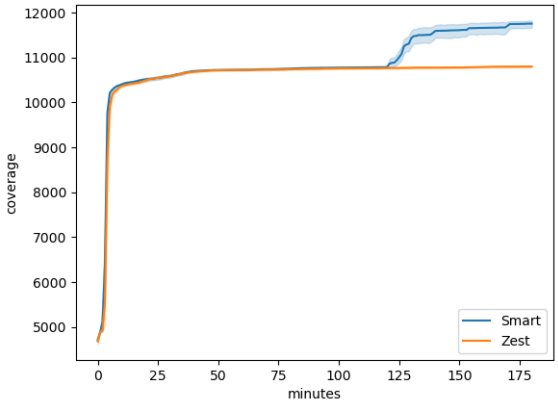

Figure 5.1: Ant

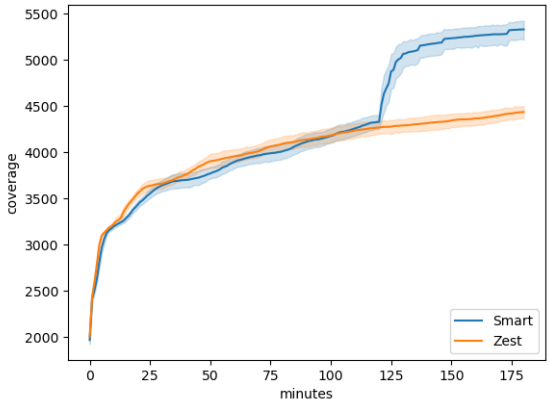

Figure 5.2: Maven

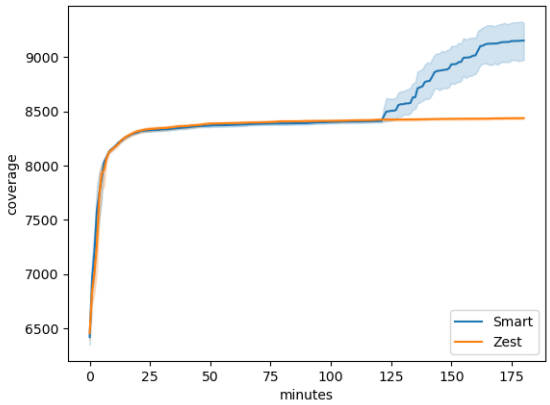

Figure 5.3: Tomcat

6 Conclusion

Concluding this bachelor thesis, we presented JQF-Smart, an extension of the generator-based fuzzer Zest that addresses the magic value problem for strings specifically, which has been a persistent challenge of coverage-guided fuzzing. The problem occurs when the executed software reaches a string comparison that requires a specific value to be provided by the input.

JQF-Smart modifies the software instrumentation and introduces adaptive input generation to circumvent the magic value problem. The modified instrumentation provides additional feedback regarding string comparisons to locate and find magic values. The adaptive generators enable the fuzzer to dynamically utilize different string generation strategies based on the fuzzing state and the provided coverage feedback.

The evaluation of the conducted experiments demonstrated significant improvements in the effectiveness of Zest after the employment of JQF-Smart. The results suggest that JQF-Smart covered hard-to-reach code paths by generating input values that could not be generated by Zest. However, it should be acknowledged that JQF-Smart has not undergone a comparative evaluation against other Java-based fuzzers, such as CONFETTI [14], that specifically tackle the magic value problem.

Therefore, we see a need for further research to evaluate, how the performance of JQF-Smart fares in comparison to other solutions to address the magic value problem in Java fuzzing. It is important to note that while the evaluation of JQF-Smart was comprehensive, it did not consider the number of bugs found during the fuzzing campaigns, which could be another subject of further research. Future work could also aim to find out why the code coverage of Zest and JQF-Smart had discrepancies when running them both on the same seed inputs and potentially

fix that problem. Another problem that should be addressed is that in certain cases of string comparisons, such as `startsWith`, it is important, which side of the comparison was generated to retrieve the magic value that should be saved for global and local hinting. Additionally, the impact of global hinting on the fuzzing performance JQF-Smart could also be evaluated in order to determine, whether that feature is useful in our context.

Finally, this thesis aimed to answer the research question of how the combination of our modified instrumentation and adaptive generators impacts the performance of generator-based fuzzing. The evaluation results presented in this thesis demonstrate that **the integration of the modified instrumentation and adaptive generators has led to considerable improvements in the performance of generator-based fuzzing.** This suggests that our approach provides a promising solution for the magic value problem in coverage-guided fuzzing, specifically for string comparisons in Java programs.

In summary, JQF-Smart has the potential to improve the effectiveness and efficiency of generator-based fuzzing, particularly in executing hard-to-reach code paths, which could be critical in ensuring the security and reliability of software systems.

Bibliography

[1] 2022. URL: https://www.informatik.hu-berlin.de/de/org/rechnerbetriebsgruppe/dienste/hpc/computeserver (visited on 02/25/2023).

[2] URL: https://junit.org.

[3] URL: https://jupyter.org.

[4] Apache Ant. 2018. URL: https://ant.apache.org (visited on 02/25/2023).

[5] Niyazi Ari and Makhamadsulton Ustazhanov. "Matplotlib in python". In: *2014 11th International Conference on Electronics, Computer and Computation (ICECCO)*. IEEE. 2014, pp. 1–6.

[6] Cornelius Aschermann et al. "REDQUEEN: Fuzzing with Input-to-State Correspondence." In: *NDSS*. Vol. 19. 2019, pp. 1–15.

[7] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-based greybox fuzzing as markov chain". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 1032–1043.

[8] Eric Bruneton. "ASM 4.0 A Java bytecode engineering library". In: (2011). URL: https://asm.ow2.io/asm4-guide.pdf.

[9] 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.

[10] Koen Claessen and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 2000, pp. 268–279.

[11] Vijay Ganesh, Tim Leek, and Martin Rinard. "Taint-based directed whitebox fuzzing". In: *2009 IEEE 31st International Conference on Software Engineering*. 2009, pp. 474–484. DOI: 10.1109/ICSE.2009.5070546.

[12] Charles R Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (2020), pp. 357–362.

[13] Paul Holser. *junit-quickcheck: Property-based testing, JUnit-style*. 2014. URL: https://pholser.github.io/junit-quickcheck (visited on 10/14/2022).

[14] James Kukucka et al. "Confetti: Amplifying concolic guidance for fuzzers". In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 438–450.

[15] laf-intel. *Circumventing fuzzing roadblocks with compiler transformations*. 2016. URL: https://lafintel.wordpress.com (visited on 01/24/2023).

[16] Caroline Lemieux et al. "Perffuzz: Automatically generating pathological inputs". In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2018, pp. 254–265.

[17] Nicholas Marriott. *tmux 3.3a*. 2022. URL: https://github.com/tmux/tmux/wiki (visited on 01/24/2023).

[18] Apache Maven. 2018. URL: https://maven.apache.org (visited on 02/25/2023).

[19] Wes McKinney et al. "pandas: a foundational Python library for data analysis and statistics". In: *Python for high performance and scientific computing* 14.9 (2011), pp. 1–9.

[20] Hoang Lam Nguyen and Lars Grunske. "BeDivFuzz: integrating behavioral diversity into generator-based fuzzing". In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 249–261.

[21] Augustus Odena et al. "Tensorfuzz: Debugging neural networks with coverage-guided fuzzing". In: *International Conference on Machine Learning*. PMLR. 2019, pp. 4901–4911.

[22] Rohan Padhye, Caroline Lemieux, and Koushik Sen. "JQF: Coverage-guided property-based testing in Java". In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2019, pp. 398–401.

[23] Rohan Padhye et al. "Semantic fuzzing with zest". In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2019, pp. 329–340.

[24] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. "T-Fuzz: Fuzzing by Program Transformation". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 697–710. DOI: 10.1109/SP.2018.00056.

[25] Nick Stephens et al. "Driller: Augmenting fuzzing through selective symbolic execution." In: *NDSS*. Vol. 16. 2016. 2016, pp. 1–16.

[26] Apache Tomcat. 2018. URL: https://tomcat.apache.org (visited on 02/25/2023).

[27] Michael L Waskom. "Seaborn: statistical data visualization". In: *Journal of Open Source Software* 6.60 (2021), p. 3021.

[28] Michal Zalewski. 2017. URL: http://lcamtuf.coredump.cx/afl/ (visited on 01/24/2023).

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den February 27, 2023