École Polytechnique Fédérale de Lausanne

Automatic Code Synthesis for Testing Java Libraries

by Zhaorui Li

Master Thesis

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Francesco Berla
External Expert

Flavio Toffalini
Thesis Supervisor

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

July 14, 2023

Learn and live.
— Unknown

Dedicated to my father, my mother, and all my friends that supported me during those years.

Acknowledgments

I want to thank my advisor, Prof. Mathias Payer, for his continuous support, close guidance, and for all the feedback I received. I enjoyed the time I spent at HexHive working with him and I thank him for pushing me to strive for more. He is a brilliant researcher and I wish him the best in his career. I also want to thank my supervisor, Flavio Toffalini, for his unwavering support, insightful discussions, and valuable feedback throughout my research journey. His guidance has been instrumental in shaping the direction of this project. The HexHive research group, with its vibrant and collaborative environment, has been a source of inspiration and joy. I am thankful to all the members of HexHive who have contributed to making this experience truly remarkable. The lively discussions, coffee breaks, and camaraderie have made writing my master's thesis an enjoyable endeavor.

I would like to extend my gratitude to my friends and family, especially my father, for their constant support and encouragement throughout my academic journey.

Lastly, I want to express my appreciation to all my schoolmates and team members at Polygl0ts for their enthusiasm and shared passion for this field. Their presence has kept my interest and passion alive, even during the most challenging and mundane subjects.

Lausanne, July 14, 2023 Zhaorui Li

Abstract

Fuzzing is widely regarded as a practical approach for detecting security bugs in software. However, its adoption often requires significant effort. In order to conduct fuzzing on a library, a fuzzer relies on a fuzz driver that exercises specific code within the library and accepts inputs from the fuzzer. The formulation of high-quality fuzz drivers with a suitable sequence of APIs is crucial for effective fuzzing, as it allows for comprehensive exploration of program states. However, the process of writing fuzz drivers is predominantly a manual task, which poses a significant obstacle to the widespread adoption of fuzzing techniques.

Efforts have been made to automate the generation of fuzz drivers, whereas all of them focus on synthesizing drivers for C/C++ libraries instead of Java libraries. Java's memory management system abstracts away pointers and provides automatic memory allocation and garbage collection. This eliminates the risk of memory corruption found in C/C++ libraries. Additionally, the differences in object manipulation and memory management between the two languages make the direct application of parameter synthesis techniques from C/C++ to Java libraries unfeasible. The methodologies developed for C/C++ libraries, which involve direct object manipulation and manual memory management, do not readily translate to the automatic memory management system and object handling mechanisms of Java.

To tackle these challenges, we first undertook a comprehensive analysis of existing Java fuzz drivers to understand how the differences between Java and C/C++ impact fuzz drivers. Then based on our analysis we introduce JFuzzDrive, a tool designed to automatically synthesize fuzzers for complex Java libraries. JFuzzDrive utilizes comprehensive system analysis to infer the interface of the targeted library and generates fuzzers tailored specifically for that library. One of the key advantages of JFuzzDrive is its ability to operate without requiring any human interaction, making it applicable to a wide range of libraries. Moreover, the fuzzers generated by JFuzzDrive utilize the power of Jazzer, enabling improved code coverage and the discovery of bugs that may be deeply embedded within the library.

We demonstrate the effectiveness of JFuzzDrive by applying it to 20 open-source project libraries from OSS-Fuzz. Additionally, we executed the generated fuzzers and compared them with manually written drivers. Results show JFuzzDrive can achieve the same coverage in most cases compared to manually written drivers. These results highlight the effectiveness and versatility of JFuzzDrive in identifying vulnerabilities and improving code coverage.

Contents

Chapter 1

Introduction

Modern applications are extremely complex and increasingly large. The extensive scope of these applications leads to an signficantly expansive attack surface. Moreover, since a significant portion of the codebase comprises third-party libraries beyond developers' control, the security risks are further amplified. Given that these applications lack security boundaries and any component's vulnerability can jeopardize the entire application, the resulting scenario is deeply concerning. A single bug within such a vast attack surface poses a significant threat to the overall security of the application.

Due to the extensive volume and complexity of these modern application code, fuzzing provides a straightforward yet powerful approach for identifying unknown vulnerabilities. Automated generational grey-box fuzzing, e.g. based on AFL[1] or its subsequent advancements like AFLfast[6] or AFLGo[5], demonstrates a high level of effectiveness in identifying program bugs. This technique achieves its efficacy by mutating inputs based on execution feedback and exploring new code coverage[21]. By traversing different program paths, fuzzed inputs implicitly generate valid and intricate program states. If an illegal path is encountered, it swiftly results in an error state, which is either gracefully handled by the program or leads to an actual crash. Consequently, code coverage serves as a reliable measure of the fuzzed program's state. Latest progress in coverage-guided fuzzing[24, 30] has expanded the reach of fuzzing to delve into deeper program paths, leading to the discovery of a considerably larger number of bugs.

Despite the impressive achievements of greybox-fuzzing techniques in terms of code coverage and discovering program crashes, their effectiveness does not directly extend to fuzzing libraries. Libraries do not have entry point for receiving inputs and thus need an application program, commonly referred to as fuzz drivers, to provide an entry point to facilitate fuzz testing. These fuzz drivers play a crucial role in exercising the library code.

Creating effective fuzzing drivers can be a time-consuming task that demands a comprehen-

sive understanding of the target library. Developers need to familiarize themselves with the target library's interface and how to utilize it correctly. For instance, when fuzzing an image format library, knowledge of specifying image contents and triggering format parsing operations is essential. Additionally, configuring the library calls in a way that crashes only occur due to implementation bugs is crucial. Crashes resulting from violating preconditions of a function, such as attempting to read from an unopened file, are unproductive distractions. Although these implicit dependencies between library calls are occasionally mentioned in documentation, they are typically not formally specified. Furthermore, fuzz targets should be deterministic and devoid of side effects to ensure reliable reproduction of crashes. Therefore this approach lacks scalability when dealing with numerous diverse libraries.

Efforts have been made to automate the generation of fuzz drivers. Google have introduced their FUDGE framework[4], which offers a semi-automated approach to generate fuzz drivers. FUDGE scans the program's source code to identify vulnerable function calls and then creates fuzz drivers by replacing parameters. While FUDGE demonstrates effectiveness in specific scenarios, it often produces an excessive number of fuzz drivers for larger projects, necessitating manual removal of invalid results. Moreover, attempting all potential drivers becomes infeasible. Another example is FuzzGen[14], which utilizes a comprehensive system analysis to infer the library's interface and synthesize fuzz drivers based on existing test cases. The performance of FuzzGen heavily relies on the quality of the existing test cases. Other works, such as IntelliGen[32] or APICraft[31], target at C/C++ libraries instead of Java libraries.

Java libraries have a lot of difference compared to C/C++ libraries. First of all, C/C++ allows direct memory manipulation and address referencing and provides low-level access to system resources and hardware while Java abstracts away the concept of pointers and provides a higher level of memory management through automatic memory allocation and garbage collection. Therefore the risk of memory corruption like memory leaks, buffer overflows, or dangling pointers does not exist in Java. In addition due to the difference in object manipulation and memory management between C/C++ and Java, the direct application of parameter synthesis techniques used for C/C++ libraries is not viable for Java libraries. The capability of C/C++ to manipulate objects directly, coupled with manual memory management, contrasts with Java's automatic memory management system. Therefore, methodologies developed for parameter synthesis in C/C++ libraries cannot be readily adapted for Java libraries, as the underlying mechanisms and considerations for object handling differ significantly.

In this thesis, we introduce JFuzzDrive, a system specifically designed to expedite the generation of fuzz drivers for Java libraries. By leveraging advanced techniques from the fields of static analysis, JFuzzDrive empowers developers with an automated solution that significantly reduces the time and effort required to create effective fuzz drivers.

The core objective of JFuzzDrive is to ensure that the automatically synthesized fuzz drivers achieve comparable performance to those created manually by expert developers. We believe that

automating the driver synthesis process should not come at the cost of compromised performance or reduced effectiveness in detecting bugs and vulnerabilities. Through rigorous evaluation and experimentation, we substantiate our claim that JFuzzDrive's automatically generated fuzz drivers exhibit similar performance and efficacy as their manually written counterparts in most scenarios.

Before formulating our approach for fuzz driver synthesis, we undertook a comprehensive analysis of existing Java fuzz drivers. Recognizing that fuzzing encompasses more than just memory bugs, we sought to identify the various types of errors these fuzzers aimed to uncover. To achieve this objective, we conducted an exhaustive review of a wide range of fuzz drivers specifically targeting diverse Java libraries. Through this meticulous examination, we gained invaluable insights and developed unique perspectives on the subject matter.

The fundamental principle guiding the design of JFuzzDrive is inspired by Ockham's Razor, a philosophical principle that suggests selecting the simplest explanation or solution when faced with multiple options. In the context of JFuzzDrive, this principle translates into generating fuzz drivers that invoke APIs with minimal effort and complexity.

To achieve this goal, JFuzzDrive follows a systematic process that leverages static analysis techniques and algorithmic methods. The initial step involves performing static analysis on the target libraries. This analysis aims to gather crucial information about the APIs, such as identifying potential API candidates, understanding type dependencies, and capturing exception information. By comprehensively examining the target libraries, JFuzzDrive establishes a solid foundation for subsequent fuzz driver generation.

Building upon the static analysis results, JFuzzDrive employs algorithmic methods to synthesize parameters for the entry points extracted from other existing work. These algorithmic techniques leverage the gathered information to determine suitable values and configurations for the fuzzing process. By automating the parameter synthesis step, JFuzzDrive eliminates the need for manual intervention and significantly reduces the effort and expertise required from developers.

Finally, armed with the synthesized parameters, JFuzzDrive constructs fuzz drivers that are specifically tailored to the target libraries. These fuzz drivers play a crucial role in initializing the fuzzing process, allowing for the systematic exploration and testing of the libraries' functionalities, inputs, and edge cases. By generating these drivers automatically, JFuzzDrive streamlines the process, ensuring efficient and effective fuzzing of the target libraries.

In order to assess its efficacy, we conducted testing of JFuzzDrive on various real-world projects obtained from Google's fuzzer-test-suite. The results demonstrate that JFuzzDrive achieves nearly identical coverage of branches, paths, and bugs in most cases when compared to manually-written drivers by domain experts. In addition the runtime overhead introduced by JFuzzDrive during driver generation is negligible.

During this project we faced many challenges. Unlike types in C, Java types with inheritance and

polymorphism make it non-trivial to identify parameter types accurately and completely. Therefore we need to retrieve all inheritance relationship of the target library, which is not easy as it looks. After identifying correctly the paramter types of target functions, parameter synthesis is another problem. Search space for API call combinations to generate parameters is huge. Which combination should be chosen in a reasonable time is a crucial part in terms of the performance of driver generation.

In the rest of the report, we first give an overview of the backgrond. Then we describe our contributions:

1. We studied all fuzz drivers targeted on 103 Java libraries and categorize them based on what errors they are trying to find.
2. We design a static analysis method on target libraries to gather API candidates, type dependency, and exception information.
3. We design an algorithm that automatically generate API call sequences for target functions and handle all the exceptions.

Subsequently, we assess the prototype of our system, JFuzzDrive, and provide an evaluation of its performance. We delve into the limitations encountered during the evaluation process, explore relevant works in the field, and ultimately present our conclusions.

Chapter 2

Background

In this chapter we provide a summary of basic knowledge required to understand the underlying concepts behind JFuzzDrive and the challenges we faced during development.

2.1 Fuzzing

Fuzzing, commonly referred to as fuzz testing, is an automated software testing technique that plays a crucial role in uncovering vulnerabilities, bugs, and crashes within software systems. Fuzzing involves providing the target system with a stream of random or unexpected inputs, such as malformed data or invalid commands, to provoke abnormal behavior and trigger potential weaknesses. The goal of fuzzing is to identify scenarios where the software does not handle these inputs correctly, potentially leading to security vulnerabilities or functional flaws.

One significant advantage of fuzzing is its ability to operate autonomously once the initial setup is completed. Unlike traditional testing methods that require continuous human intervention, fuzzing can run unattended, freeing up valuable resources and allowing developers to focus on other tasks. This automation aspect makes fuzzing highly efficient and cost-effective, particularly when applied to large software systems or complex environments.

Furthermore, fuzzing is highly parallelizable and can harness the power of modern computing architectures. By utilizing multiple cores or distributed computing resources, fuzzing can execute in a highly parallel manner, enabling the simultaneous generation and execution of numerous test cases. This scalability allows fuzzing to handle massive software systems, libraries, or network protocols efficiently, making it a widely adopted and preferred software testing technique.

2.1.1 Fuzzer

The software used to perform fuzzing is commonly referred to as a fuzzer. Fuzzers can be classified into different categories based on their input generation techniques, crash handling approaches, or information handling mechanisms. Two primary categories of fuzzers are generational fuzzers and mutational fuzzers.

Generational fuzzers, such as PROTOS[28] or PEACH[10], generate inputs by following predefined formats or specifications. These fuzzers are designed to create test cases that conform to specific protocols, file formats, or data structures. Generational fuzzers often leverage knowledge about the expected structure or syntax of the input to generate meaningful and valid test cases. By adhering to these predefined formats, generational fuzzers aim to explore different variations and edge cases within the specified structure, potentially uncovering vulnerabilities or unexpected behaviors.

On the other hand, mutational fuzzers, including AFL[1] and honggfuzz[13], employ a different approach for input synthesis. Mutational fuzzers start with existing inputs, typically referred to as seed inputs, and randomly mutate or modify them to generate new test cases. Mutations may involve flipping bits, inserting or deleting data, or rearranging the input sequence. The purpose of these random mutations is to explore different paths and input variations that the software under test may not have encountered before. By mutating inputs in a randomized manner, mutational fuzzers aim to uncover unforeseen vulnerabilities or crashes that may not be captured by generational fuzzers.

Both generational and mutational fuzzers have their strengths and weaknesses. Generational fuzzers excel in situations where the input format is well-defined and knowledge about the structure or syntax is available. These fuzzers can generate targeted and valid inputs, effectively exploring the specified protocol or file format. On the other hand, mutational fuzzers are valuable when the input format is unknown or when testing for unexpected behaviors. By randomly mutating inputs, mutational fuzzers have the potential to discover unforeseen vulnerabilities or unexplored execution paths.

The choice between generational and mutational fuzzers depends on the specific requirements and characteristics of the target software or system. Often, a combination of both approaches is used to achieve comprehensive coverage and maximize the chances of identifying a wide range of vulnerabilities or defects.

2.1.2 Fuzzer Performance

When evaluating the effectiveness and performance of a fuzzer, code coverage and the number of unique crashes detected are two crucial aspects to consider. Code coverage refers to the percentage of the target codebase that the fuzzer is able to exercise during the testing process. A higher code

coverage indicates a more thorough exploration of the software's functionalities and can increase the likelihood of uncovering potential vulnerabilities.

Similarly, the number of unique crashes found is an essential metric that gauges the fuzzer's ability to identify and trigger unexpected behaviors or flaws in the target software. Each unique crash represents a distinct issue that could potentially be a security vulnerability or a stability problem.

In addition to code coverage and unique crashes, there are other factors that can be used to evaluate a fuzzer's performance. These factors include: fuzzing time, fuzzing speed, number of inputs per crash, false positive rate, etc.

2.1.3 Fuzz Driver

In contrast to fuzzing programs that typically have an explicit entry point designed to accept inputs, fuzzing libraries operate differently. Fuzzing libraries, which lack a dedicated entry point, require an application program to serve as the entry point. This application program, commonly referred to as a fuzz driver or fuzz harness, plays a crucial role in facilitating fuzzing for the target library.

The fuzz driver serves as a crucial bridge between the fuzzer and the target library. It establishes the necessary communication and interaction required for effective fuzzing. Essentially, the fuzz driver determines the direction and scope of the fuzzing process, guiding how the fuzzer interacts with the target library and what inputs are provided.

The design and implementation of a robust fuzz driver are critical to the success of library fuzzing. A well-crafted fuzz driver should be able to handle different input types, support a variety of library configurations, and capture relevant information or outputs for analysis and vulnerability detection.

2.2 Java Basics

We will provide a short summary of what does the *Java* language differ from other languages. We will focus on aspects that is the source of non-trivial challenges we faced during driver generation.

2.2.1 Java Generics

Java has a complex type system, especially its generics. Java generics are a facility of generic programming that was introduced in 2004. It is designed to allow a type or method to operate on objects of various types while providing compile-time type safety. Basically Java generics are parameterized types that is similar to template in C++. These generics are used both in methods

(Generic Method) and classes (Generic Class). In addition to non-generic classes, there are other types can be a valid type for the type arguments for a parameterized type:

1. Type Variable. Type variable is an unqualified identifier and a placeholder for a specific type.
2. Wildcard Type. Java allows using wildcard type optionally with an upper or lower bound serve as the type arguments for a parameterized type.
3. Other generic types including array type that is made up of generic types.

Java generics have been proven to Turing complete[12].

Type Erasure

For backwared compatibility, Java implements type erasure from Java 5. During compilation, Java compiler replace all type parameters in generic types with their bounds or *Object* if the type parameters are unbounded. To preserve type safety and polymorphism in extended generic types, Java compiler insert type casts and generate bridge methods. By type erasure, no new classes are created for parameterized types and thus generics incur no runtime overhead.

Type erasure also introduces some problems. Due to type erasure, type parameters cannot be determined at run-time. To solve this problem, Java generics generate only one compiled version of a generic class or function regardless of the number of parameterizing types used. Furthermore, Java validate the type information at compile-time and is not included in the compiled code.

2.2.2 Java Reflection

Reflection is an unique feature of *Java*. It allows an executing Java program to inspect and/or modify the behavior of fields, methods, classes, and interfaces at runtime. For example, it's possible to obtain the names of all members of a Java class and display them. The ability to examine and manipulate a Java class from within itself seems trivial, but this feature does not in other programming languages. For example, it is impossible in a C, or C++ program to get information like signature or declaring class about the functions defined within that program.

2.2.3 Java Exceptions

Exceptions are unwanted or unexpected events, which occur during the execution of a program, that interrupts the normal workflow of the program. Contrary to *C* which does not support exception handling, *Java* has a complete mechanism to handle exceptions. This mechanism is different from

that in *C++* where all types can be thrown as exceptions and all exceptions are runtime exceptions, *Java* only allows throw *throwable* as exceptions and *Java* has two kinds of exceptions: checked exceptions and unchecked exceptions. Exceptions that are subclasses of *Error* or *RuntimeException* are unchecked exceptions, everything else are checked exceptions.

Checked Exception

Checked Exceptions are exceptions that are checked at compile time. These exceptions usually represents errors outside the control of program. For example, *FileNotFoundException* is a checked exception which occurs when the specified file does not exist. These exceptions are designed to avoid some wrong application level operations and thus *Java* stipulates if some code in a method throws a checked exception, then this method must either specify the exception using the *throws* keywords or handle the exception within the method by try-catch clause.

Unchecked Exception

Unchecked exceptions, on the contrary, are exceptions that are not checked at compile time. Therefore the program does not need to handle or specify these exceptions. It is up to the programmars to be aware of these exceptions and handle them. Unchecked exceptions are usually caused by programming errors, such as out of bounds accessing or attempting to divide by zero, or by some JVM failures like out of memory situations and thus are typically unrecoverable .

Chapter 3

Study of Java Drivers

In this chapter, we delve into our comprehensive study of Java drivers to shed light on the objectives and goals of Java fuzzers in scenarios where memory corruption is not the primary concern. Specifically, our investigation revolves around the following key question: What are Java fuzzers attempting to discover when memory corruption vulnerabilities are not the focus?

To undertake this study, we collected 103 Java libraries that have fuzz drivers available in OSS-Fuzz[25]. With these libraries at our disposal, we conducted a meticulous examination of their associated fuzzer drivers. In total, we identified 207 fuzz drivers associated with these 103 Java libraries. This implies that, on average, each library required approximately two drivers for thorough testing. This substantial number of fuzz drivers ensures extensive coverage and evaluation of the libraries, allowing us to assess their robustness and identify potential vulnerabilities or unexpected behavior. By leveraging this diverse set of fuzzing cases, our study aims to provide valuable insights into the effectiveness and practicality of fuzzing techniques for Java libraries.

We manually examined all these 207 fuzz drivers. In Figure 3.1, we present our analysis results, providing a comprehensive overview of our findings. This analysis encapsulates the collective insights obtained from the manual examination of each fuzzer driver, allowing us to identify common patterns, trends, and objectives pursued by Java fuzzers when memory corruption vulnerabilities are not the primary target.

We categorize Java fuzzers into 3 kinds based on the errors they try to find:

1. Unexpected Exceptions. This category includes all errors and exceptions that are not expected and result in a crash. Fuzzers in this category aim to identify any unexpected behavior or crashes in the target Java application[1]. They typically generate input data that triggers various

[1]There are two ways of identifying unexpected crashes: 1. fuzzer invoke some APIs and treat every crash as unexpected. 2. the programmar specify some exceptions to be caught and consider all the other exceptions as unexpected crashes

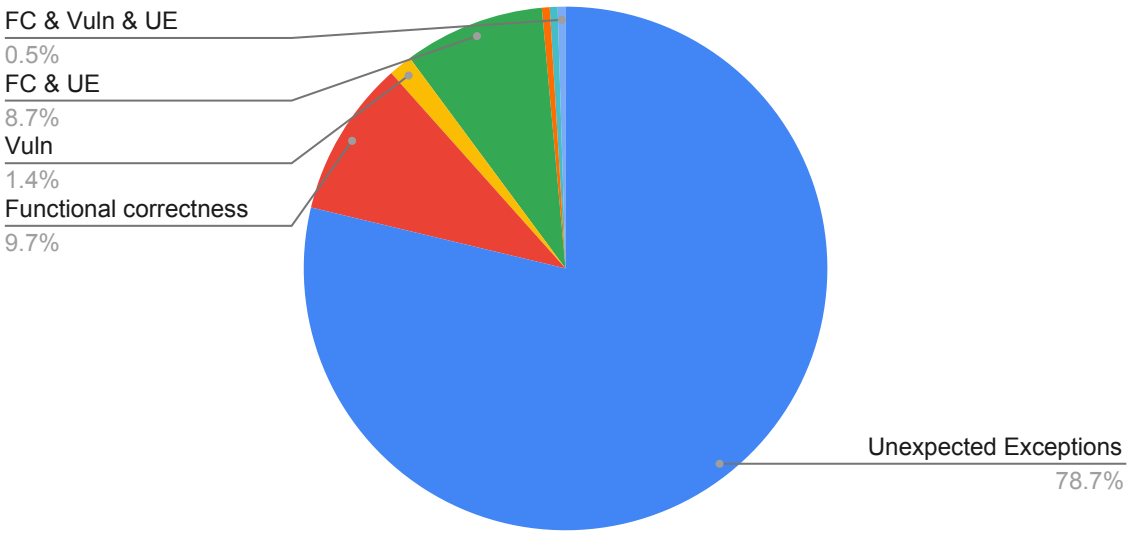

Figure 3.1: Characteristics of Java Fuzz Drivers

code paths and boundary conditions to uncover hidden exceptions or errors.

2. Functional (Logic) Error. Fuzzers in this category focus on detecting functional or logical errors in the target Java application. These errors occur when the behavior of the application deviates from the intended logic or requirements. For example, a fuzzer may check if a string value remains unchanged after undergoing encoding and decoding. If the string value changes, the fuzzer reports a crash, indicating a potential functional error.

3. Vulnerabilites. Although Java doesn't suffer from memory corruption vulnerabilities like C or C++, it can still be susceptible to other types of vulnerabilities. Fuzzers in this category aim to identify vulnerabilities such as SQL injection, path traversal, or denial-of-service (DoS) in Java applications. Each type of vulnerability may require a different approach for detection. For example, a fuzzer targeting SQL injection vulnerabilities may check if illegal characters persist after input filtering, indicating a potential injection vulnerability.

These three categories of fuzzers are not entirely independent of each other. While each category has its primary focus, there is some overlap in the types of errors they can detect. For example, a fuzz driver checks if there are any logic error of the system can also check if there are any inputs that will lead to the crash of system.

From the analysis result we can see most of the fuzzers (78%) prioritize finding unexpected exceptions. A small set of fuzz drivers (around 10%) focus on proving the functional correctness of the libraries. However, only certain types of fuzzers, particularly those targeted at specific libraries or components like JSON parsers or database connectors, are specifically tailored to uncover

vulnerabilities inherent to those particular functionalities.

By conducting this in-depth study of Java drivers, we aim to contribute to the understanding of Java fuzzing practices and shed light on the broader objectives pursued beyond memory corruption vulnerabilities. Through this empirical analysis, we hope to provide valuable insights for the development of more effective and targeted fuzzing techniques, ultimately enhancing the overall security and reliability of Java software systems.

Chapter 4

Design

In this chapter we will go through the designs of JFuzzDrive. We will introduce our design goals and the overview of JFuzzDrive. We will also explain what were the key challenges we had to face and the solutions we adopted to overcome these problems.

4.1 Goals

As mentioned in chapter 3, a Java Fuzz Driver is aimed at finding three kinds of errors in the Java library: unexpected exceptions, logic errors, and vulnerabilities. We do not focus on catching logic errors and vulnerabilities because they are difficult to infer and they account for only a small part of all the Java Fuzz Drivers (less than 25%). Therefore our goal is to provide a framework that automatically generates fuzz drivers for Java libraries that try to find unexpected exceptions in the libraries.

This framework does not need to take care of what APIs the generated driver must contain. But it should identify which APIs inside the library can be used in generated drivers. It should also take care of invoking APIs: that is, it should identify what types the API needs, construct corresponding objects, and parse them as parameters. In addition it should identify and handle all the exceptions thrown by the APIs it used. Finally, the driver generation process should not introduce too much runtime overhead.

4.2 System Overview

The architecture of JFuzzDrive is divided into two main components: a *static analyzer* and a *driver generator*. A Java library jar and a list of API extracted from the library that the generated driver

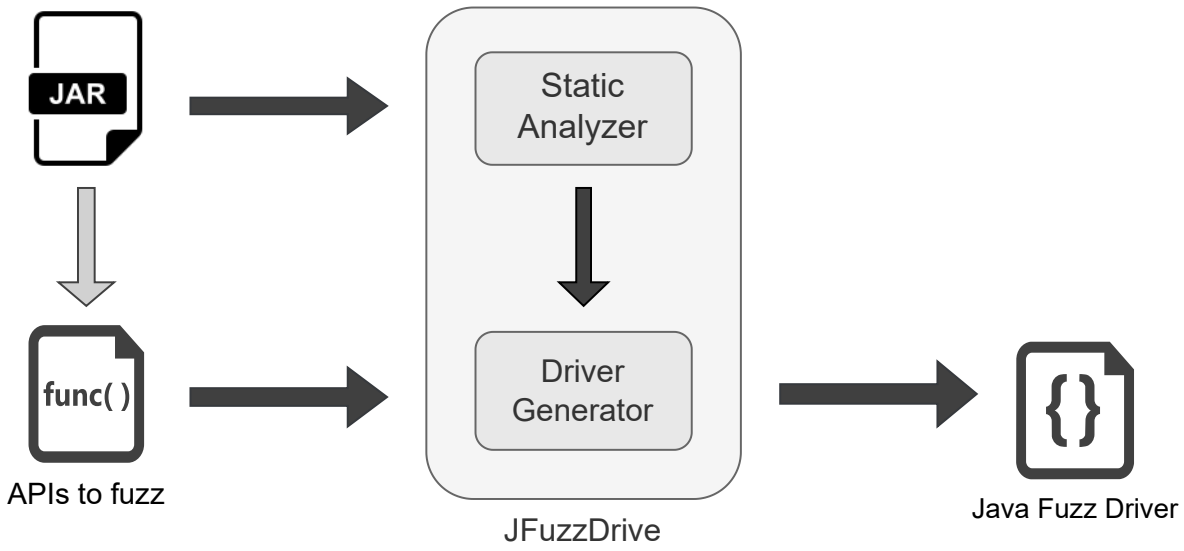

Figure 4.1: Overview of the structure of JFuzzDrive

must contain serve as the input of JFuzzDrive. The *static analyzer* takes the library jar as input and performs static analysis on this library to extract API candidates, exception information, and type dependency relationship. The *driver generator* part generates drivers based on the information extracted by *static analyzer* and the input APIs.

The separation of *static analyzer* and *driver generator* brings a benefit that the *static analyzer* can run offline and thus reduces the runtime overhead of driver generation. Also it makes the driver generation more easy to maintain and update. In the rest of this chapter we will go into detail about how these two components work.

4.3 Static Analyzer

The *static analyzer* performs static analysis on the input Java libarary jar. It extracts all APIs from the library jar that can be used to construct objects as well as their corresponding exceptions. In addition it builds a type dependency graph which records all dependencies among the classes in the Java library. These static analysis can be processed offline so that the *driver generator* can use these results to generate drivers for different input APIs without analyzing the library again.

4.3.1 API Inferring

The initial task for the static analyzer is to extract APIs from the input Java library that can be utilized in generating drivers. As mentioned in section 2.2.1, Java boasts a Turing complete type system,

which presents unique challenges for type checking. Due to the undecidability of type checking in Java, it becomes impossible for a type checker to guarantee both partial correctness and termination simultaneously.

To address this challenge, the static analyzer employs a strategy to narrow down the scope of extracted APIs. Specifically, it excludes APIs that involve parameter types containing wildcards, type variables, or generic arrays. By focusing on APIs with more deterministic and well-defined parameter types, the static analyzer can simplify the analysis process while still capturing a significant portion of relevant APIs.

One advantage of Java's Turing complete type system is that the type information itself often suffices to create objects. In other words, the method signature, which includes complete type information of the parameters and the declaring class, is often adequate for the driver generator's purposes. This approach reduces the reliance on complex type checking algorithms while still ensuring the necessary information is available for generating drivers.

In summary, the static analyzer proceeds by traversing all classes defined within the input jar. During this traversal, it records the method signatures of all methods, including normal methods, static methods, and constructors, defined within each class. By capturing these method signatures, the static analyzer gathers the essential information required for subsequent stages of the driver generation process.

By focusing on method signatures and excluding certain types involving wildcards, type variables, or generic arrays, the static analyzer strikes a balance between extracting relevant APIs and mitigating the challenges posed by Java's Turing complete type system. This approach enables efficient extraction of APIs for the driver generation process, ensuring the subsequent stages have the necessary information to construct effective drivers.

4.3.2 Exceptions

Besides the method signatures of APIs, the *static analyzer* also needs to extract the exception information thrown by APIs. We discuss in section 2.2.3 that in Java has two kinds of exceptions: checked exceptions and unchecked exceptions. The unchecked exceptions are fatal and not recoverable and hence are usually unexpected exceptions that a fuzz driver tries to find out. In addition these unchecked exceptions is non-trivial to identify during static analysis since they are thrown in runtime instead of compilation time. Therefore the *static analyzer* focuses on checked exceptions and thus gather the exception information of API through its signature.

4.3.3 Type Dependency Graph

Determining what types are suitable to parse as the argument is not a trivial thing due to the complicated type system in Java. Using the parameter type declared in the method merely is not enough. Listing 4.1 shows an example of simplified inheritance relationship in the Java library *Guava*. In some cases, the parameter type is even an interface or an abstract class that no methods or constructors return such type.

```
public class TreeBasedTable<R, C, V> extends StandardRowSortedTable<R, C, V>
class StandardRowSortedTable<R, C, V> implements RowSortedTable<R, C, V>
public interface RowSortedTable<R, C, V> extends Table<R, C, V>
public void putAll(Table<R, C, V> table);
```

Listing 4.1: A simplified inheritance relationship in *Guava*. The type *TreeBasedTable* is suitable to parse as the argument of method *putAll*.

To address the aforementioned challenge, the static analyzer incorporates the construction of a type dependency graph. This graph serves as a comprehensive record of all inheritance information within the analyzed Java library. By capturing the relationships between classes and their inheritance hierarchies, the type dependency graph provides valuable insights into the available types and their associations.

During the subsequent driver generation process, the driver generator component leverages the type dependency graph to efficiently search and identify suitable types that can be parsed as parameters. By traversing the graph, the driver generator can navigate through the inheritance relationships and extract relevant type information based on the specific requirements of the driver generation algorithm.

4.4 Driver Generator

The *driver generator* is composed of three modules: object construction, IR generation, and lifting. The object construction module creates objects (through an API call sequence) to parse as the parameters of the input APIs. Then the IR generation module embeds this API call sequence with exception information and transfer them into intermediate representation. Finally these intermediate representations are lifted to Java code and outputed as the final Java fuzz driver.

4.4.1 Object Construction

This module creates objects to parse as the parameters of the input APIs. The output of this moudle is an API call sequence where the last API call is the input API. In addition all the parameters and

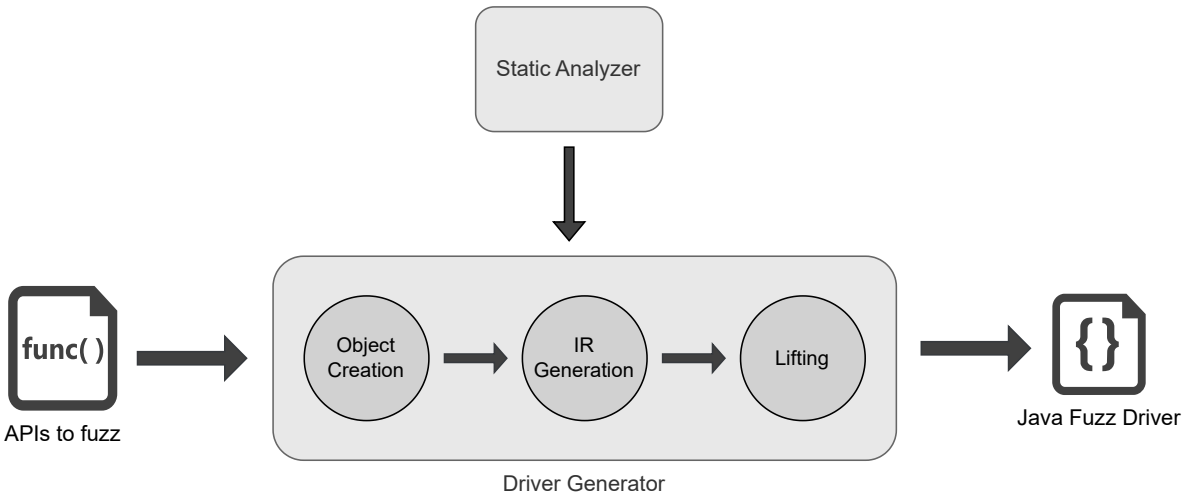

Figure 4.2: Structure of Driver Generator

declaring classes are related to an API call in this call sequence (the variable returned by this API call are parsed as this parameter).

The main challenge of this module is that the search space for API call combinations is very large. For example for library antlr4, which has 9000+ API candidates and 600+ classes, has around 1 million combinations to generate an object for type $LexerGrammar$. Traversing all of these combinations will introduce huge runtime overhead for driver generation. Therefore we follow the Ockham's Razor and choose the API call sequence with the minimum length, which implies the sequence creates the minimum number of new objects.

Algorithm 1 shows a simplified object creation process. For an input API, we will call this object creation algorithm for each parameter type and its declaring class (if it is not a static method). The key idea of this algorithm is a Breadth First Search. Each step we pick an API call sequence from the queue. Then we pick one type that this API call sequence require, add an API to this API call sequence that returns this type, and put all of these API call sequences to the next level queue. If this API call sequence does not need any other type, it is finished. Once a finished API call sequence occurs, we search the rest of the queue for all the other finished API call sequences and pick one randomly from them as the final result. Once the queue is empty, we set it to the next level queue and clear the next level queue. The usage of next level queue ensures that each API call sequence in the same queue has the same length.

4.4.2 IR Generation

The purpose of this module is to seamlessly integrate the API call sequence generated by the object creation module with essential exception information and transform them into a structured

Algorithm 1: Object Creation

Input: Type $r\,t$
Output: An API call sequence

```
Create empty list cand
Create queue Q, Q_next
for Constructors and APIs c return type r t or its subtypes do
    Let reqtype[c] = all parameter types (and declaring class) of c
    Q.append(c)
while True do
    Let u = Q.pop()
    if reqtype[u] is empty then
        /* We finish creating a new type. we check all other API call sequences if they
           are finished                                                              */
        cand.append(u)
        for rest x of Q do
            if reqtype[x] is empty then
                cand.append(x)
        Randomly choose from cand and return
    else
        Let t =reqtype[u].pop()
        for Constructors and APIs c return type t or its subtypes do
            Let u' = u + c
            Let reqtype[u']=reqtype[u]+ all parameter types (and declaring class) of c
            Q_next.append(u')
    if Q is empty then
        Q = Q_next
        clear(Q_next)
```

intermediate representation. By combining the API calls with their corresponding exception details, obtained from the exception dictionary extracted through the *static analyzer*, we create a comprehensive representation of the program's execution flow.

The intermediate representation is categorized into two distinct types: one category represents the creation of objects, while the other category signifies the invocation of methods. Each entry within the intermediate representation corresponds to a specific API call within the original API call sequence, capturing crucial details and context.

The object creation entries depict the creation and initialization of objects, including the associated constructor calls and any relevant parameters. These entries provide a detailed overview of how objects are instantiated and prepared within the program. Conversely, the method invocation

entries represent the execution of methods, encompassing the method name, the object on which the method is invoked (if applicable), and any relevant arguments passed during the invocation. These entries highlight the sequence of method calls and provide insights into the interactions between different objects and components of the program.

A detailed explanation of these intermediate representations will be presented in the next chapter.

4.4.3 Lifting

This module takes care of lifting the intermediate representation generated by the IR generation module to Java code and outputs it as the generated Java Driver. A simplified idea of how lifting works is as following: First we assign a token to each intermediate representation indicating the variable returned by (or constructed by) this intermediate representation; Next for each intermediate representation, we substitute every parameter (and declaring class), which relates to an intermediate representation, with its corresponding assigned tokens; Finally we transfer each intermediate representation to a Java clause. If an intermediate representation has exceptions, we wrap it with try-catch clause.

Chapter 5

Implementation

In this chapter we cover our implementation of JFuzzDrive. We cover the details how we analyze an Java library and how we generate drivers.

5.1 Static Analyzer

The *static analyzer* needs to extract method signatures and exception information and builds a type dependency graph. The implementation is harder than it looks. We will share in this section the challenges we faced during implementation and the solutions we adopted to overcome them.

5.1.1 API Inferring

During this process, two significant challenges arise. The first challenge relates to type erasure, as discussed in section 2.2.1. Due to type erasure in Java, the binary representation of compiled code only contains erased type information, rather than the original, more specific type information. This limitation complicates the extraction of accurate type information during static analysis.

The second challenge involves recording not only the methods defined within a class but also inherited methods from superclasses and superinterfaces. Merely considering the methods declared within a class does not provide a comprehensive picture of the available methods in the class hierarchy.

To address these challenges, Java reflection comes to the rescue. Java reflection provides a powerful mechanism for obtaining information about classes, interfaces, and their associated methods at runtime. It enables access to the complete method signatures, including the original type information, by extracting them from the constant pool section in the class file. This capability

of reflection overcomes the limitation of type erasure and allows for the retrieval of more precise type details.

Furthermore, Java reflection provides the means to retrieve all methods associated with a class, including those declared by the class itself, as well as those inherited from superclasses and superinterfaces. By leveraging reflection, the static analyzer can comprehensively capture the full set of methods available within the class hierarchy, ensuring that no methods are overlooked.

In summary, the *static analyzer* addresses these challenges by employing Java reflection. It utilizes reflection to resolve classes within the Java library jar, bypassing generic classes and interfaces since they cannot be directly instantiated. Subsequently, the static analyzer traverses the methods and constructors of each class, leveraging reflection to access the complete method signatures. It excludes constructors of abstract classes and compiler-generated methods, as these are not intended to be directly invoked.

5.1.2 Exceptions

The problem for exceptions is that Java compiler does not allow two exceptions related by subclassing occurs in one multi catch statement. For example *FileNotFoundException* and *IOException* cannot occur in the same catch statement since *FileNotFoundException* is a subclass of *IOException*. In addition if we do not use multi catch statment, the order of catch blocks is restricted. Java compiler does not allow catching a child exception after catching a parent exception. Therefore when retrieving the exception information from method signatures, we ignore those exceptions that is the subclass of one recorded exceptions. We also keep this order during the driver generation part.

5.1.3 Type Dependency Graph

We build type dependency graph to record all inheritance relationship. This process is not trivial especially when a subclass parses normal classes only for part of the type arguments. Listing 5.1 shows an example of inheritance with partially parsed type arguments.

```
public class Foo extends Bar<Double>
public class Bar<T> implements Qux<String, Boolean, Integer, T>
public interface Qux<A, B, C, D> extends Bax<A>
```

Listing 5.1: Example of inheritance with partially parsed type arguments. We need to records class *Foo* as a subclass of *Bar<Double>*, *Qux<String, Boolean, Integer, Double>* and *Bax<String>*

Our solution is to keep the type variables as variables until a subclass parses a normal class to it. In the example above, we record *Qux<A, B, C, D>* as a subclass of *Bax<A>*. When we start retrieving the inheritance of *Bar*, we substitute variable *A* with *String* and thus record *Bar<T>* is a subclass of

Bax<String>. Similarly we record *Foo* as a subclass of *Qux<String, Boolean, Integer, Double>* after we substitute varialbe *T* with *Double*.

5.2 Driver Generator

In this section we share our implementation of the *driver generator*. We will discuss how we reuse a variable when constructing objects, what IR we use, and what Java clause we use for lifting.

5.2.1 Object Construction

We found out during object construction, built-in types in Java need special care compared to customized class defined in the library. The first problem is how to create a built-in type does not specify in any library. To solve this problem, we manually add constructors for built-in types that the library might use.

Another problem is when constructing objects, whether to reuse a created variable relates to the final performance of the generated driver. We found out usually a fuzz driver tends to invoke multiple methods for the same varaible rather than recreating them. In addition the fuzz driver prefers creating a new instance for Java built-in types like *int* or *double*. Listing 5.2 shows an example of variable reuse in real fuzz driver for Java library *javassist*. Therefore our strategy for variable reuse is we set a moderate probability for Java built-in types and a high probability for class defined in the library.

```
cc.getSuperclass();
cc.getNestedClasses();
cc.getClassFile();
cc.getInterfaces();
cc.getDeclaringClass();
cc.getComponentType();
```

Listing 5.2: Code snippet of fuzz driver for Java library *javassist*

5.2.2 IR Generation

We discuss in section 4.4.2 that each intermediate representation represents an API call. Therefore we define two kinds of intermediate representation: one representing an API call and the other representing creating an object. For creating objects, we found creating array objects is different from creating other objects. We only need to call constructors to create a normal object while array objects do not have constructors. In addition array objects have length and dimension which a

normal object does not. Hence for array objects, we create a new kind of intermediate representation and set the length of the object to a fixed default length.

5.2.3 Lifting

Lifting module takes care of transfering intermediate representations to Java clauses and wrap them with try-catch clause. It also needs to take care of necessary import statement. Whereas Java compiler supports using full qualified name (package name followed by class name separated by a period) of a class without importing it. Therefore we use full qualified name for all classes for convenience.

Also this module needs to connect the driver with the fuzzer. The fuzzer provide several APIs that can generate primitives (e.g. *int*, *double*, etc.). Therefore we transfer all intermediate representations for constructing primitives to the corresponding fuzzer input APIs. The problem is for type *String* and *byte[]* which does not have a fixed length, the fuzzer provide two APIs: one taking an integer as parameter and the other taking all data left in the input. Therefore we sort all intermediate representation for constructing primitives and transfer the last one to the API that take all data left in the input. The other intermediate representation we parse a fixed integer as the parameter.

Chapter 6

Evaluation

In this section, we aim to validate the claims made earlier by conducting a series of experiments. Our objective is to provide empirical evidence supporting the effectiveness and efficiency of JFuzzDrive in generating drivers with comparable performance to manually-written drivers. Furthermore, we investigate whether JFuzzDrive introduces any runtime overhead during the driver generation process.

To achieve these goals, we formulate the following research questions that we seek to answer:

RQ1 How long does it take to generate a driver using JFuzzDrive? Can JFuzzDrive scale effectively to handle large libraries?

RQ2 How do the drivers generated by JFuzzDrive compare to manually-written drivers in terms of coverage?

RQ3 Does JFuzzDrive successfully identify crashes that are also detected by manually-written drivers?

To address these research questions, we conduct a comprehensive evaluation. We start by measuring the average time required for driver generation using JFuzzDrive. This analysis provides insights into the efficiency of JFuzzDrive and its ability to handle various Java libraries, including large-scale ones.

Next, we evaluate the fuzzing coverage achieved by the drivers generated by JFuzzDrive. By comparing this coverage with that of manually-written drivers for real-world Java libraries, we can determine the extent to which JFuzzDrive effectively captures the relevant program behavior.

Finally, we assess whether JFuzzDrive successfully identifies crashes that are also detected by manually-written drivers. This analysis validates the reliability and effectiveness of the generated

drivers in uncovering potential vulnerabilities and error scenarios.

6.1 Setup and Hardware

We run our fuzzing campaigns with Jazzer [17] version 0.17.1, which was obtained from the git release with the specific commit hash b12d1ea863b336b120e192700ac11c9744af6cfd. Jazzer is a widely recognized coverage-guided, in-process fuzzer based on libFuzzer[23] that targets the JVM platform. Renowned for its effectiveness, Jazzer has successfully identified numerous vulnerabilities in software systems. Jazzer relies on Jacoco[16] for edge coverage instrumentation for JVM bytecode. Jacoco calculates branch coverage through all **if** and **switch** statements.

We extracted 20 Java libraries from the OSS-Fuzz platform to serve as our benchmark. OSS-Fuzz, a free fuzzing platform designed for the open-source community, hosts numerous manually written drivers for each library, which we utilized in our study. To ensure consistency and accuracy, we retrieved the latest version of each library from the Maven repository. In cases where a library was not available in the Maven repository, we followed the instructions provided by OSS-Fuzz to compile the binary ourselves.

For each library, we focused on extracting the APIs present in the manually written fuzz drivers as our input APIs. These APIs serve as the foundation for generating our own drivers for the fuzzing process. In total, we generated four additional drivers for each library, resulting in a total of five fuzz drivers for each library: four generated drivers and one driver obtained from OSS-Fuzz, which serves as the ground truth or reference driver.

By gathering these libraries, extracting their respective APIs, and generating multiple drivers for each library, we established a comprehensive benchmark to evaluate the performance and effectiveness of our approach. This benchmark enables us to compare the performance of the generated drivers against the existing manual drivers, providing valuable insights into the capabilities and limitations of our automated fuzzing process.

The experimentation was carried out on a machine equipped with an Intel Xeon Gold 5218 CPU, featuring a clock speed of 2.30GHz and a total of 16 cores and 32 threads. The machine was equipped with 64 GB of RAM and operated on the Ubuntu 20.04 operating system with a Linux kernel version of 5.4.0. The software stack included OpenJDK 17.0.7 as both the Java Development Kit (JDK) and the Java Runtime Environment (JRE), and Python version 3.9.5.

Considering the limited timeframe available for the experiments, the fuzzer campaigns were executed in parallel to optimize efficiency. To achieve this, dockers were built for each fuzz driver, and each docker was assigned to a specific physical core of the machine. This parallelization allowed for the simultaneous execution of up to 16 fuzzers, maximizing the utilization of available computational resources.

To ensure a controlled and consistent experiment setup, each fuzz driver was allocated a maximum of 5 hours for execution, with a limit of 10 crashes. This means that the fuzzer would stop running either after reaching the 5-hour time limit or upon discovering 10 crashes, whichever occurred first. To obtain reliable and statistically significant results, each fuzzer was executed three times, and the average result was recorded.

6.2 Performance

We evaluate JFuzzDrive based on three aspects: how long does it take to generate a driver, how many crashes does our generated drivers find compared to the ground truth, and what is the coverage of our generated drivers.

6.2.1 Generation Performance

Table 6.1 shows the scale of our benchmark. For every library, JFuzzDrive spends less than 0.5 seconds to generate one driver. Therefore the runtime overhead of driver generation is negligible.

Library	**Total APIs**	**Classes**	**API to fuzz**	**Library**	**Total APIs**	**Classes**	**API to fuzz**
angus-mail	2665	205	3	janino	3611	661	2
antlr4	9305	2567	6	javassist	4272	426	8
apache-compress	5562	562	5	jersey	2250	1855	9
apache-imaging	5673	817	3	jettison	773	47	1
apache-math	9329	1357	5	joda-time	5215	247	1
bc-java	30523	7867	2	json-smart	188	106	1
cbor	378	63	1	osgi	346	203	6
dom4j	5348	184	1	xmlbeans	45840	1029	2
greenmail	1324	593	3	xstream	3938	498	3
h2database	23410	1028	2	zxing	1475	291	5

Table 6.1: Statics of benchmark we use. **Total APIs** means the total amount of APIs exposed by this library. **Classes** means the number of classes defined in the library. **API to fuzz** means the number of APIs extracted as input that the generated driver must contain.

6.2.2 Coverage

We compare both the branch coverage and the line coverage of JFuzzDrive generated drivers with the OSS-Fuzz drivers. In the case of the *antlr4* and *xstream* libraries, all the fuzzers encountered the same timeout, resulting in the inability to gather any coverage information for these two libraries.

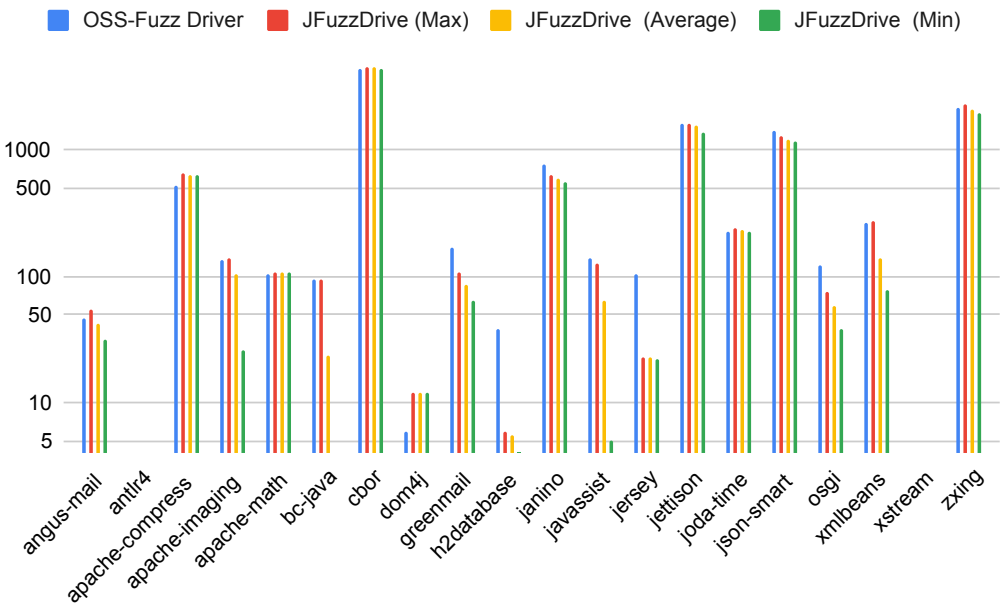

Figure 6.1: Branch Coverage for drivers. We take the logarithm of the original branch coverage multiplied by 10000 as the plot value.

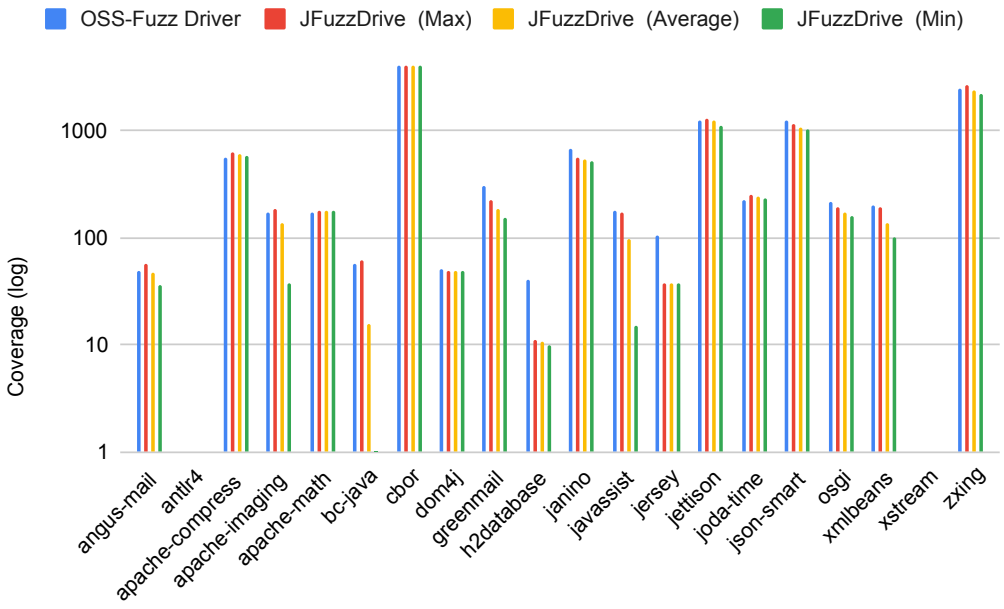

Figure 6.2: Line Coverage for drivers. We take the logarithm of the original line coverage multiplied by 10000 as the plot value.

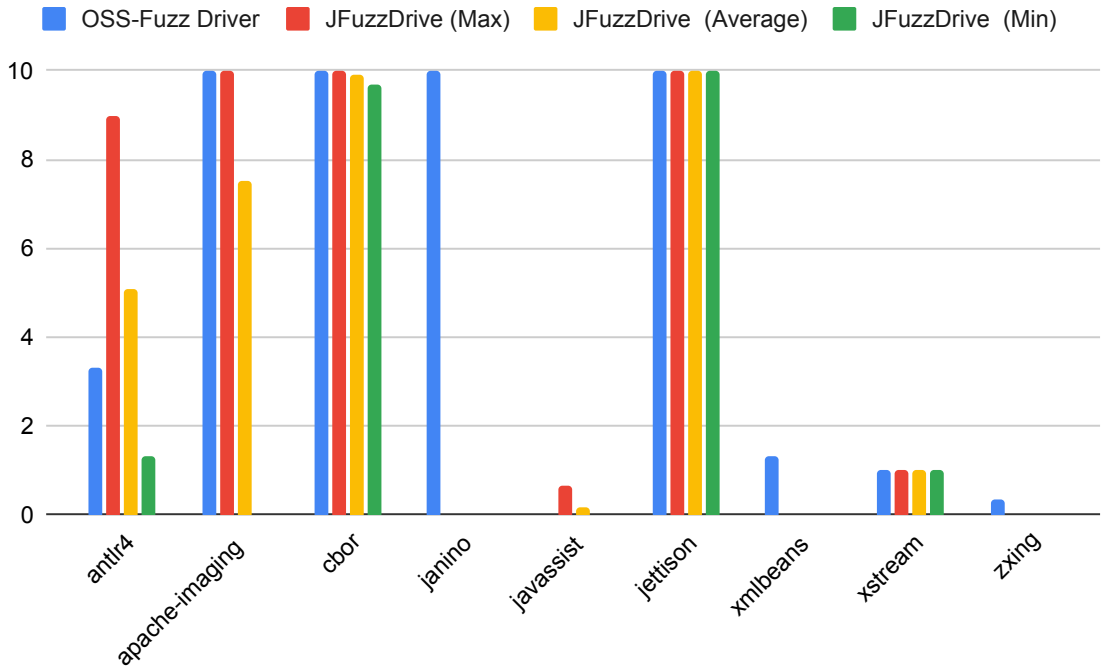

Figure 6.3: Crashes found by drivers. We omit those libraries that none of the drivers find any crashes

Figure 6.1 and 6.2 shows the coverage result. We can see that, for the majority of cases, the drivers generated by JFuzzDrive achieve coverage levels that are comparable to those of expert-written drivers. Additionally, the results indicate a stable performance of JFuzzDrive, with minimal variation observed between the best and worst coverage outcomes.

However, there are instances where JFuzzDrive's coverage falls slightly below that of manually written drivers. Likewise, there are cases where the coverage levels among the generated drivers exhibit some fluctuation. These specific scenarios will be explored and discussed in detail in Section 6.3.

6.2.3 Crashes

Another important aspect of assessing the performance of fuzz drivers is to analyze the number of crashes discovered by the fuzzer. During our evaluation, we observed that JFuzzDrive identified certain crashes that were not classified as crashes by the manually written drivers. These particular findings are categorized as false positives. In Figure 6.3, we present the results of the crash findings after filtering out these false positives.

Upon examination, we found out that, in the majority of cases, the number of crashes detected

by JFuzzDrive is comparable to or even exceeds the number of crashes identified by the manually written drivers. However, it is worth noting that false positive crashes did occur, and their occurrence will be thoroughly investigated in the subsequent section. Additionally, the reasons behind cases where JFuzzDrive did not identify crashes that were detected by the manually written drivers will also be discussed.

6.3 Discussion

In this section we discuss the factor that affects JFuzzDrive differs from manually written drivers. These factor might lead to both a lower coverage and a higher coverage.

6.3.1 Unchecked Exceptions

As we mentioned in section 4.3.2, JFuzzDrive treats all unchecked exceptions as unexpected exceptions that must be caught. However in some cases, unchecked exceptions are expected due to the randomness of fuzzer's input. Listing 6.1 shows an example of catching unchecked but expected exceptions. These exceptions are usually caused by the format of fuzzer's input does not match the format required by the API. These format mismatching crashes are false positives mentioned in section 6.2.3.

The presence of uncaught exceptions can be a contributing factor to the lower coverage achieved by JFuzzDrive. In our experimental setup, JFuzzDrive terminates the fuzzing process once a sufficient number of crashes have been encountered, regardless of whether they are false positives or genuine crashes. As a result, in certain cases, JFuzzDrive may terminate prematurely due to the occurrence of false positive crashes. Consequently, this early termination leads to a lower coverage in terms of covered lines compared to the manually written drivers.

However, it is important to note that for those cases where the false positive crashes do not cause JFuzzDrive to halt prematurely, the coverage is not impacted. In these scenarios, JFuzzDrive continues the fuzzing process and achieves coverage levels comparable to the manually written drivers.

```
try {
    Version v = new Version(fuzzedDataProvider.consumeRemainingAsString());
} catch(IllegalArgumentException ex) {
    return;
}
```

Listing 6.1: Code snippet of fuzz driver for Java library *osgi*. The version string should be in format like 1.0.1-rc whereas fuzzer input are just random bytes.

6.3.2 Relationship among arguments

JFuzzDrive employs a strategy where it constructs objects by invoking the object creation algorithm (Algorithm 1) individually for each type required by an API. However, this method overlooks the inherent relationships among arguments within an API and across multiple APIs. An example of such argument relationships is illustrated in Listing 6.2. As a consequence of the reuse strategy adopted by JFuzzDrive, there is a possibility that the same variable is used for these two arguments and therefore result in lower coverage.

```
String regExStr = data.consumeString(100);
String parse = data.consumeRemainingAsString();
try {
    RegularExpression regex = new RegularExpression(regExStr);
    regex.matches(parse);
}
catch(Exception e){}
```

Listing 6.2: Code snippet of fuzz driver for Java library *xmlbeans*. Parameter *regExStr* and *parse* should be different. Otherwise the regex match does not make much sense.

Addressing this limitation will be a focus of future work, as improving the understanding and utilization of argument relationships within APIs could enhance the coverage achieved by JFuzzDrive. By incorporating more sophisticated strategies that consider argument dependencies, JFuzzDrive could further improve its effectiveness in generating comprehensive and high-coverage fuzz drivers.

6.3.3 Unintentional API Usage

JFuzzDrive only extracts method signatures for API used to construct objects. These method signatures have no type state information and thus generated API call sequences could have issues. Listing 6.3 shows an intended way to create a *Document*. Whereas JFuzzDrive only uses its constructor and creates an empty document.

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder;
Document doc;
try {
    builder = factory.newDocumentBuilder();
} catch (ParserConfigurationException e) {
    return;
}
try{
    doc = builder.parse(
```

```
        new ByteArrayInputStream(data.consumeRemainingAsBytes()));
}
catch (SAXException | IOException e){
    return;
}
```

Listing 6.3: Code snippet of fuzz driver for Java library *dom4j*. This is a typical way of creating a valid and meaningful document

But for this library JFuzzDrive achieve a higher coverage than the manually written driver. The reason is the manually written driver spent all the time on building a correct document and thus does not cover the document handling part. On the contrary, JFuzzDrive only creates an empty document which successfully cover the document handling part.

6.3.4 *Null* type

JFuzzDrive does not use *Null* type in driver generation for two reasons: 1. everything in Java can be assigned to *Null* except primitive types or types decorated by *NotNull* annotation; 2. Whether *Null* type should be avoided is controversial. This strategy could have issues when the library has special treatment of *Null* type. Listing 6.4 shows an example. Since JFuzzDrive does not use *Null*, it can never cover the function *readHeader* and thus have low coverage.

```
public ResultSet read(Reader reader, String[] colNames) {
    return readResultSet(colNames);
}
private ResultSet readResultSet(String[] colNames) {
    this.colNames = colNames;
    initRead();
}
private void initRead() {
    if (colNames == null) {
        readHeader();
    }
}
```

Listing 6.4: Code snippet of Java library *h2databse*. Only when the parameter *colNames* is set to *Null*, the function *readHeader* can be executed.

Chapter 7

Related Work

JFuzzDrive is closely aligned with recent studies in the field of fuzzing. A number of state-of-the-art fuzzers have been proposed in the literature to detect bugs effectively and efficiently. AFL[1] is a prominent example of a feedback-guided, grey-box fuzzer. AFL measures the achieved edge coverage through instrumentation to obtain feedback. This feedback is then utilized to assess the quality of an input, scoring each input and reusing the best ones for subsequent runs. The selected inputs are further mutated based on a genetic algorithm. This iterative process has proven to be highly effective, leading to various improvements[3, 6, 7, 9, 22, 27] in seed scheduling, input mutation, and feedback mechanisms.

However, it is important to note that the majority of these improvements have been primarily focused on fuzzing programs that take files or commands as input. While these advancements have made significant contributions to the field of fuzzing, JFuzzDrive addresses a different aspect by targeting fuzzing at the API level, specifically in the context of Java libraries. By focusing on this particular domain, JFuzzDrive aims to generate comprehensive and high-quality fuzz drivers that effectively explore the API functionalities and uncover potential vulnerabilities.

7.1 Library Fuzzing

In contrast to end-to-end fuzzing approaches, library fuzzing requires a deep understanding of the target library, including correct API usage and prerequisites for API calls, when implementing fuzz drivers. While initiatives like OSS-Fuzz[25], utilizing libFuzzer[23], incentivize open source maintainers to implement fuzz drivers, the manual effort required for fuzz driver development has hindered the widespread adoption of fuzzing in many open source projects. FuzzBuilder[15] was proposed as a tool to partially address this issue by assisting in transforming unit tests (UT) into fuzz drivers. However, FuzzBuilder's approach still relies on manual configuration to specify test

functions and target API parameters, resulting in fuzz drivers whose quality heavily depends on the extent of manual work involved.

In contrast, JFuzzDrive offers an automatic and reliable solution for generating fuzz drivers by analyzing unit tests. By leveraging this automated approach, JFuzzDrive aims to accelerate the adoption of library fuzzing at a larger scale, removing the barriers posed by manual efforts. The focus of JFuzzDrive is to enable the seamless and efficient generation of fuzz drivers, ensuring the effectiveness and quality of the resulting drivers while alleviating the burden on developers.

7.2 Fuzz Driver Generation

The field of automatic fuzz driver generation is an area of active research, with recent advancements in this domain. Notable techniques and approaches have been proposed to address this challenge. FUDGE[4], for instance, focuses on automatically synthesizing fuzz drivers for open-source libraries. It extracts potential fuzz drivers from consumer programs that utilize the target library and relies on human experts to make decisions regarding the most suitable drivers for fuzzing. On the other hand, FUZZGEN[14] takes a source-code-based approach, leveraging consumer programs' source code to learn the correct usage of library functions. By constructing an Abstract API Dependency Graph (A^2DG) using this learned knowledge, FUZZGEN can generate fuzz drivers by traversing the A^2DG.

Besides these two works, Intelligen[32] employs a methodology of selecting functions that involve potentially risky operations, such as memory or pointer access, in order to synthesize fuzz drivers. However, due to its lack of consideration for API relationships, the resulting generated fuzz drivers may be incomplete in terms of functionality and coverage. Recently UTOPIA[18] directly utilizes the meticulously crafted API call sequence from existed unit tests, which is specifically designed for testing purposes. By leveraging this well-constructed API call sequence, UTOPIA reduces the potential inaccuracies or incorrectness that may arise when synthesizing fuzz drivers. However all of these works target at generating fuzz drivers for C/C++ libraries instead of Java libraries. To our knowledge, JFuzzDrive is the first work that synthesize drivers for Java libraries.

In contrast to these static approaches that analyze source code, there are two projects that adopt dynamic methods to infer API relationships. WINNIE[19] focuses on fuzzing closed-source libraries on the Windows platform through automatic fuzz driver generation and efficient cloning of Windows processes. APICraft[31], on the other hand, targets closed-source libraries of the MacOS SDK. Both projects dynamically trace API sequences during runtime by executing the target programs, such as through manual dry runs. They leverage the observed API call sequences to generate new fuzz drivers. However, it is worth noting that these dynamic approaches are not well-suited for large-scale adoption and push-button automation. While they provide valuable insights by dynamically capturing API behaviors, their reliance on runtime execution and manual dry runs make them less suitable for widespread automated usage.

7.3 Unit Test Generation

While the research area of fuzz test generation is relatively new, there exists a closely related field of automated unit test generation. Randoop[26] is an example of a pure random test generation tool for Java. It utilizes feedback information as guidance to generate random method calls. In contrast, Palulu[2] employs a dynamic-random approach by inferring a call sequence model from a sample execution and generating random tests based on that model. However, it lacks information about arguments and does not extract additional information from the source code through static analysis. On the other hand, RecGen[34] follows a static-random approach, relying solely on static analysis to guide random test generation. While it may struggle to create valid call sequences for complex interfaces, it does not include a dynamic analysis phase. Palus[33], developed at Google, improves upon these tools by combining both static and dynamic approaches.

Another approach to generating unit tests involves carving them from existing tests, such as extracting unit tests from execution traces of system tests[8]. Basilisk[20] utilizes a similar concept, carving parameterized unit tests from system tests for C programs. By employing this method, a fuzz target for a particular function can be precisely extracted based on a specific system test execution that calls the function.

Certain automated testing tools are capable of generating a test driver when the target of testing is a single top-level function. For instance, the symbolic execution tool DART[29] can extract the interface of a given function and automatically create a driver for it. Micro execution[11] can execute a function without requiring a user-provided driver by automatically identifying its input/output interface. The focus of this form of test driver generation is to enable the execution of a single function without the need to write setup or teardown driver code.

These techniques primarily concentrate on directing the entire suite of unit tests towards predefined objectives rather than optimizing specific unit tests. JFuzzDrive, on the one hand, differs from these unit test generation techniques in various aspects, including goals and approaches. On the other hand, some concepts employed in these unit test generation techniques have influenced the design of JFuzzDrive. For instance, the utilization of evolutionary algorithms for test generation is an idea that has inspired the development of JFuzzDrive.

Chapter 8

Conclusion

8.1 Future Work

Support for other errors JFuzzDrive only focus on finding checked exceptions while logic correctness, vulnerabilities, and some unchecked exceptions are also crucial for libraries. In future work, we would like to make JFuzzDrive generating drivers that catch all kinds of errors.

Improved Static Analysis JFuzzDrive only extracts method signatures for driver synthesis but the information stored in method signatures is limited. Improving static analysis on libraries, for example analyzing type state information or API dependency information can definitely improve the performance of generated drivers.

Improved Driver Synthesis Algorithm JFuzzDrive generate drivers based on the Ockham's Razor. This principle is proved to have some issues for some libraries. Improving the driver synthesis algorithm can make JFuzzDrive more scalable and targeting at more libraries.

8.2 Conclusion

In this paper, we make an comprehensive analysis on Java fuzzers and introduce JFuzzDrive, an automated framework for synthesizing fuzz drivers for Java libraries. JFuzzDrive utilizes static analysis on libraries to gather information about library consumers and synthesize parameters for APIs automatically. We evaluate JFuzzDrive on 20 real world libraries and the results show JFuzzDrive achieve the same performance as manully written drivers in most cases. These results highlight the capabilities of JFuzzDrive in effectively fuzzing libraries and uncovering critical security issues.

Bibliography

[1] *American Fuzzy Loop*. https://lcamtuf.coredump.cx/afl/.

[2] Shay Artzi, Michael D Ernst, Adam Kiezun, Carlos Pacheco, and Jeff H Perkins. "Finding the needles in the haystack: Generating legal test inputs for object-oriented programs". In: (2006).

[3] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. "REDQUEEN: Fuzzing with Input-to-State Correspondence". In: *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[4] Domagoj Babic, Stefan Bucur, Yaohui Chen, Franjo Ivancic, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. "FUDGE: Fuzz Driver Generation at Scale". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019.

[5] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. "Directed Greybox Fuzzing". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2329–2344.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-Based Greybox Fuzzing as Markov Chain". In: *IEEE Transactions on Software Engineering* 45.5 (2019), pp. 489–506.

[7] Peng Chen and Hao Chen. "Angora: Efficient Fuzzing by Principled Search". In: *CoRR* abs/1803.01307 (2018). arXiv: 1803.01307.

[8] Sebastian Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Matthew Jorde. "Carving and Replaying Differential Unit Test Cases from System Test Cases". In: *IEEE Transactions on Software Engineering* 35.1 (2009), pp. 29–45.

[9] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. "CollAFL: Path Sensitive Fuzzing". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 679–696.

[10] *GitLab Protocol Fuzzer*. https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce.

[11] Patrice Godefroid. “Micro Execution”. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 539–549.

[12] Radu Grigore. “Java Generics are Turing Complete”. In: *CoRR* abs/1605.05274 (2016). arXiv: 1605.05274.

[13] *Honggfuzz*. https://android.googlesource.com/platform/external/honggfuzz/.

[14] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. “FuzzGen: Automatic Fuzzer Generation”. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2271–2287.

[15] Joonun Jang and Huy Kang Kim. “FuzzBuilder: Automated Building Greybox Fuzzing Environment for C/C++ Library”. In: *Proceedings of the 35th Annual Computer Security Applications Conference*. ACSAC ’19. San Juan, Puerto Rico, USA: Association for Computing Machinery, 2019, pp. 627–637.

[16] *Java Code Coverage Library*. https://www.jacoco.org/jacoco/trunk/index.html.

[17] *Jazzer*. https://github.com/CodeIntelligenceTesting/jazzer.

[18] Bokdeuk Jeong, Joonun Jang, Hayoon Yi, Jiin Moon, Junsik Kim, Intae Jeon, Taesoo Kim, WooChul Shim, and Yong Ho Hwang. “UTOPIA: Automatic Generation of Fuzz Driver using Unit Tests”. In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society. 2022, pp. 746–762.

[19] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. “Winnie: Fuzzing windows applications with harness synthesis and fast cloning”. In: *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS 2021)*. 2021.

[20] Alexander Kampmann and Andreas Zeller. “Carving Parameterized Unit Tests”. In: *CoRR* abs/1812.07932 (2018). arXiv: 1812.07932.

[21] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. “Evaluating Fuzz Testing”. In: *CoRR* abs/1808.09700 (2018). arXiv: 1808.09700.

[22] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. “Steelix: Program-State Based Binary Fuzzing”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017, pp. 627–637.

[23] *LibFuzzer - a library for coverage-guided fuzz testing*. https://llvm.org/docs/LibFuzzer.html.

[24] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. “Fuzzing: Art, Science, and Engineering”. In: *CoRR* abs/1812.00140 (2018). arXiv: 1812.00140.

[25] *OSS-Fuzz*. https://github.com/google/oss-fuzz.

[26] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. “Feedback-Directed Random Test Generation”. In: *29th International Conference on Software Engineering (ICSE'07)*. 2007, pp. 75–84.

[27] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. “Optimizing seed selection for fuzzing”. In: *23rd USENIX Security Symposium (USENIX Security 14)*. 2014, pp. 861–875.

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

[29] Koushik Sen. “DART: Directed Automated Random Testing”. In: *Hardware and Software: Verification and Testing*. Ed. by Kedar Namjoshi, Andreas Zeller, and Avi Ziv. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 4–4.

[30] Hyrum K. Wright, Daniel Jasper, Manuel Klimek, Chandler Carruth, and Zhanyong Wan. “Large-Scale Automated Refactoring Using ClangMR”. In: *2013 IEEE International Conference on Software Maintenance*. 2013, pp. 548–551.

[31] Cen Zhang, Xingwei Lin, Yuekang Li, Yinxing Xue, Jundong Xie, Hongxu Chen, Xinlei Ying, Jiashui Wang, and Yang Liu. “APICraft: Fuzz Driver Generation for Closed-source SDK Libraries”. In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2811–2828.

[32] Mingrui Zhang, Jianzhong Liu, Fuchen Ma, Huafeng Zhang, and Yu Jiang. “IntelliGen: Automatic Driver Synthesis for Fuzz Testing”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 2021, pp. 318–327.

[33] Sai Zhang, David Saff, Yingyi Bu, and Michael D. Ernst. “Combined Static and Dynamic Automated Test Generation”. In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ISSTA '11. Toronto, Ontario, Canada: Association for Computing Machinery, 2011, pp. 353–363.

[34] Wujie Zheng, Qirun Zhang, Michael Lyu, and Tao Xie. “Random Unit-Test Generation with MUT-Aware Sequence Recommendation”. In: *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*. ASE '10. Antwerp, Belgium: Association for Computing Machinery, 2010, pp. 293–296.