



École Polytechnique Fédérale de Lausanne

Framework for Evaluating Synthetic Drivers for Library Fuzzing

by Zurab Tsinadze

Master Project Report

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Dr. sc. Flavio Toffalini
Thesis Supervisor

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

June 9, 2023

Abstract

The software contains bugs or defects that can potentially allow unauthorized access to our data or enable attackers to escalate their privileges, such as by installing malware. Various testing strategies exist to help developers identify these bugs before they can cause any harm. One effective strategy is fuzzing, which involves executing a program in a controlled test environment with random inputs to detect crashes.

While fuzzing binaries, such as CLI programs, can be easily done with a single click using out-of-the-box solutions, library fuzzing requires testers to tackle an additional level of complexity: it is essential to have drivers that make use of the APIs exposed by the library. However, developing drivers that are semantically correct and have high coverage is a time-consuming and a technically challenging task that typically requires a deep understanding of the library.

Several research efforts have focused on automating driver generation. The objective of this project is to investigate the current state of the art in this field, and create methodology and tooling for evaluating automatically generated drivers.

Contents

Abstract	2
1 Introduction	4
2 Background	6
2.1 Fuzzing	6
2.2 Sanitization	7
2.3 Library Testing	9
2.3.1 Challenges with the Library Fuzzing	9
2.3.2 LibFuzzer	11
2.4 Related Works	12
2.4.1 FUDGE	13
2.4.2 FuzzGen	13
2.4.3 UTopia	14
2.4.4 RUBICK	14
3 Methodology	16
3.1 High Level description	16
3.2 Design of the fuzzing campaigns	17
3.3 Crash clustering	19
3.4 Coverage metrics	20
3.5 Driver Complexity	21
4 Evaluation	22
4.1 Target Libraries	22
4.2 Crashes	23
4.3 Coverage	23
4.4 Driver Complexity	26
5 Conclusion	32
Bibliography	33

Chapter 1

Introduction

Complex software systems are susceptible to various types of input-based bugs. Some of these bugs might be vulnerabilities that can be exploited by the adversary to gain unauthorized access, execute arbitrary code, or mount a *denial-of-service* (DoS) attack.

Fuzzing is a software testing technique that involves executing the program with random inputs to catch vulnerabilities and bugs. The idea behind the technique is simple, it stresses the program with a large number of diverse inputs. These inputs could be random (black box fuzzing) or could be generated by mutating previous "interesting" (according to some metric, such as code coverage) inputs. Coverage-guided mutational fuzzing is considered to be state-of-art.

Fuzzing has been on the rise in the last couple of years and is currently commonly used across the industry and many open-source projects, including libraries.

Fuzzing binaries is considered much easier compared to libraries due to several factors. When fuzzing binary, the tester focuses on the executable file itself, they can directly target the file, which usually has a well-defined entry point, such as the main function, which is the ideal point to inject inputs for fuzzing. Binaries usually have a predictable input format: often accept CLI arguments, files, network packets, or some other well-structured input. Moreover, binaries generally do not have state management complexity.

On the other hand, fuzzing libraries requires additional considerations. Libraries are developed to be reusable, serving different purposes for different applications. Testers need to manually define and implement entry points, or so-called drivers. The drivers act as the interface between the fuzzing engine and the library. They need to be carefully designed, taking into account multiple different factors that we will discuss in the next chapter.

While it may involve more effort and consideration compared to fuzzing binaries, thoroughly testing libraries is extremely important, since popular libraries are used across different software

applications, making any vulnerability in them highly impactful and widespread, giving attackers an opening to exploit all consumer software (think of Log4j vulnerability [1]). By fuzzing libraries, potential security risks can be identified and addressed early in the software development life-cycle, reducing the likelihood of vulnerabilities being exploited in production.

Google operates numerous online services and, just like the whole industry, heavily relies on various open-source software components. With the goal of improving the overall security and stability of open-source projects, they started the OSS-Fuzz project [5]. They provide automated fuzzing infrastructure, tools, and even computational resources to allow "important" projects (open to interpretation, but as a rule-of-thumb, it needs to be critical for the global IT world or have a large user base) to integrate fuzzing into their CI pipeline. As of June 2023, there are 1087 projects in OSS-Fuzz and the project has helped identify and fix over 8,900 vulnerabilities and 28,000 bugs [6].

While OSS-Fuzz undoubtedly reduces the burden on library developers by providing all the tooling, there is still room for improvement. An ideal scenario would be to automate the process of writing drivers, reducing or even completely removing the effort required from developers. Automating the driver generation has immense potential in enhancing the efficiency of library fuzzing. By removing this step, developers can save time and resources and instead focus on rolling-out new features or analyzing and fixing bugs and vulnerabilities. An automated approach to driver generation would involve leveraging program analysis techniques to analyze APIs and codebase in general. This analysis could identify key entry points, function signatures, input requirements, inter-dependencies of APIs, and expected behaviors, allowing the automated tool to generate semantically correct and effective fuzzing drivers. By automating the driver generation process, library developers could seamlessly integrate fuzzing into their development workflows, promoting continuous security testing and proactive vulnerability discovery. This automation would not only reduce the burden on library developers but also ensure a more systematic and comprehensive approach to library fuzzing, ultimately leading to improved software security and stability.

The topic of automating the generation of drivers for library fuzzing has attracted attention in recent research works. Several papers have been proposed with various techniques, such as program analysis, analysis of how consumers use the libraries, and large language models.

The goal of this project is to create a framework and methodology for evaluating the effectiveness of these tools and comparing them with drivers manually written by developers of the libraries. This evaluation would allow us to understand the strengths and limitations of these tools. Furthermore, comparing automated and manual drivers shows their relative performance and effectiveness in terms of code coverage achieved, bug discovery rate, and time and effort required. It helps determine whether automated drivers can provide comparable results to their manual counterparts and identify areas for improvement or optimization, the insight that could be used while designing similar tools.

Chapter 2

Background

This section aims to provide the necessary context and knowledge of the technologies used in this project, discuss the challenges associated with library fuzzing, and present related work in synthetic driver generation.

2.1 Fuzzing

The intuition behind fuzzing lies in mutating or generating inputs that are then fed into the program being tested. By subjecting a program to a wide range of inputs, fuzzing helps identify and address potential security flaws, stability issues, and unexpected behavior. Mutation-based fuzzing involves taking initial inputs (also known as corpus) and applying various modifications, such as bit flips, random insertions, deletions, or replacements, to create a mutant. By exploring the program's behavior with these inputs, fuzzing aims to trigger crashes. These crashes are saved for human analysis. While "interesting" inputs are added to the corpus.

Due to its randomness, the core goal of the fuzzing is to execute the program with different inputs as many times as possible considering a limited amount of resources, both time and CPU. However, there is always a trade-off between the speed and efficiency of fuzzing. Black box fuzzers treat the program as a black box, apply random mutations to the inputs, and collect as many crashes as possible. Coverage-guided grey box fuzzing, on the other hand, uses feedback obtained from code coverage analysis during the fuzzing process. This technique aims to guide the fuzzing process toward uncovering deeper program states. It instruments the program under test to collect coverage information during execution and uses this feedback to guide the generation of new inputs. By utilizing coverage information, coverage-guided grey box fuzzing tends to be more effective at finding complex vulnerabilities and exploring deeper program states. Figure 2.1 and Figure 2.2 demonstrate the difference between the two techniques.

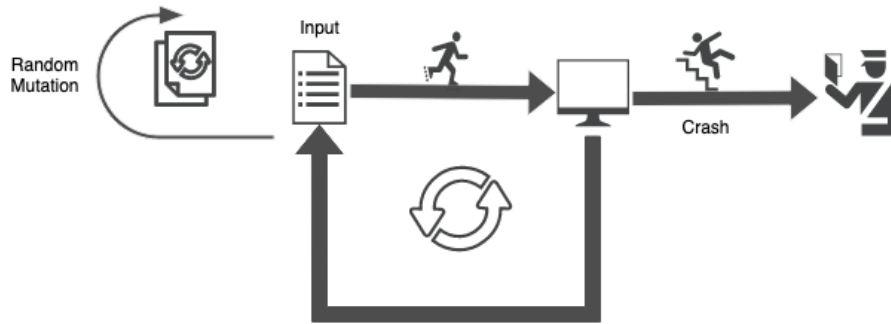


Figure 2.1: Black Box fuzzing

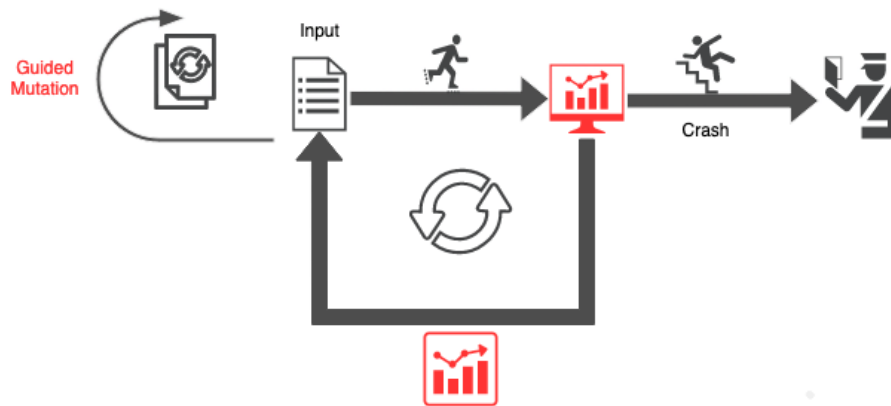


Figure 2.2: Coverage-guided Greybox fuzzing

Fuzzing engines focus on capturing inputs that cause program crashes, but they may miss inputs that trigger bugs but do not result in a crash. For instance, scenarios like out-of-bounds reads or writes may not cause a crash. This is where sanitizers play a crucial role.

2.2 Sanitization

Sanitizers are oracles that mark a bug. They instrument programs compile-time by adding different checks to enforce policies during run-time. Sanitization helps identify issues such as memory errors or leaks (ASan), uninitialized memory reads (MSan), undefined behavior (UBSan), and data races (TSan) that can be potential security vulnerabilities. Violation of these policies results in a crash that will be saved by the fuzzing engine, sanitizers make faults detectable. By integrating sanitizers into the fuzzing process, the fuzzer can detect and report errors more effectively. However, Careful consideration is required when selecting the appropriate sanitizer for fuzzing, as the core principle of fuzzing revolves around speed and efficiency. Sanitizers, while immensely valuable for

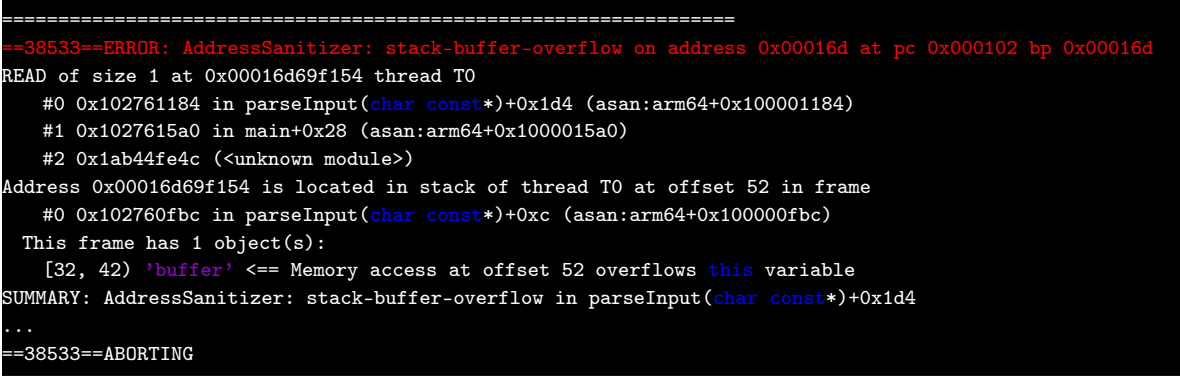
detecting memory-related bugs, come with significant run-time costs, often resulting in performance slowdowns ranging from 2 to 10 times. Additionally, compatibility issues arise when multiple sanitizers are combined, leading to an exponential increase in slowdowns for each added sanitizer. Given these constraints, it becomes crucial to prioritize the selection of sanitizers that align with the specific goals of the fuzzing process. Since a substantial portion of bugs typically involve memory-related issues [3] [11], it often makes sense to utilize AddressSanitizer (ASan) due to its effectiveness in detecting memory errors. To see ASan in action consider the following simple code snippet:

```
#include <iostream>

void parseInput(const char *input) {
    char buffer[10];
    strncpy(buffer, input, sizeof(buffer) - 1);
    buffer[sizeof(buffer) - 1] = '\0';
    std::cout << "Value at index 20: " << buffer[20] << std::endl;
}

int main() {
    const char *input = "This is long input string";
    parseInput(input);
    return 0;
}
```

If we compile this code with no sanitizer flags and run it, we would get an output that looks something like this: Value at index 20: ?, where ? is some garbage value. However, compiling it with `-fsanitize=address` flag will result in a crash:



```
=====
==38533==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x00016d at pc 0x000102 bp 0x00016d
READ of size 1 at 0x00016d69f154 thread T0
    #0 0x102761184 in parseInput(char const*)+0x1d4 (asan:arm64+0x100001184)
    #1 0x1027615a0 in main+0x28 (asan:arm64+0x1000015a0)
    #2 0x1ab44fe4c (<unknown module>)
Address 0x00016d69f154 is located in stack of thread T0 at offset 52 in frame
    #0 0x102760fbc in parseInput(char const*)+0xc (asan:arm64+0x100000fbc)
This frame has 1 object(s):
    [32, 42) 'buffer' <== Memory access at offset 52 overflows this variable
SUMMARY: AddressSanitizer: stack-buffer-overflow in parseInput(char const*)+0x1d4
...
==38533==ABORTING
```


2.3 Library Testing

2.3.1 Challenges with the Library Fuzzing

As previously mentioned, libraries are designed for the reusability across various applications and for serving different purposes. Unlike binaries, they do not have a well-defined entry point, posing a challenge for testers. To bridge the gap between the fuzzing engine and the library, testers must manually create drivers—code snippets that facilitate interaction with the library. These drivers serve as the interface through which the fuzzing engine interacts with the library’s functionality. There are multiple challenges associated with writing such drivers.

1. **Understanding of the library:** Testers should have good knowledge of intended usage, APIs, and functionality. This is crucial for identifying key entry points and relevant target functions for fuzzing.
2. **Input Semantics:** While the fuzzing engine will handle generating semi-random inputs, testers should split this data into different parts to fuzz different parameters. These chunks should adhere to API’s expected formats.
3. **Starting Seeds:** Testers need to select initial inputs, known as seeds, to effectively jumpstart the fuzzing process. The quality and diversity of these seeds play a crucial role in the effectiveness of the fuzzing campaign. However, selecting seeds for libraries can be more challenging compared to binaries.
4. **Callback Handling:** Libraries oftentimes have callbacks that require testers to implement appropriate handling in the drivers.
5. **Error Handling:** Drivers should have error handling mechanisms to gracefully handle exceptions, crashes, or any other unexpected behavior during fuzzing. Drivers should not crash.
6. **State Management:** Libraries have stateful behavior and testers need to manage internal state appropriately. This involves initializing the library, setting necessary configurations, and ensuring the state is reset on each fuzz input run. This is extremely important for in-process fuzzing engines.

For synthetic fuzz drivers to be considered on par with manually written ones, they need to effectively address the majority of these challenges. To understand the complexity of this requirement, let us have a look at the following code:

Listing 2.1: Open tiff file, read metadata, read image data, close the file

```
#include <stdio.h>
#include <tiffio.h>

void errorHandlingCallback(const char* module, const char* format, va_list args) {
    vfprintf(stderr, format, args);
}

void processImageData(TIFF* tiff, uint32_t width, uint32_t height) {
    uint16_t bitsPerPixel, samplesPerPixel;
    TIFFGetField(tiff, TIFFTAG_BITSPERSAMPLE, &bitsPerPixel);
    TIFFGetField(tiff, TIFFTAG_SAMPLESPERPIXEL, &samplesPerPixel);
    uint32_t imageSize = width * height * (bitsPerPixel / 8) * samplesPerPixel;
    unsigned char* imageData = (unsigned char*)malloc(imageSize);
    if (imageData) {
        TIFFReadRGBAImage(tiff, width, height, (uint32_t*)imageData, 0);
        // Process the image data as needed
        free(imageData);
    }
}

void processTIFFFile(const char* filename) {
    TIFF* tiff = TIFFOpen(filename, "r");
    if (tiff) {
        uint32_t width, height;
        // Retrieve image dimensions
        if (TIFFGetField(tiff, TIFFTAG_IMAGEWIDTH, &width) &&
            TIFFGetField(tiff, TIFFTAG_IMAGELENGTH, &height)) {
            // Process the image data
            processImageData(tiff, width, height);
        } else {
            fprintf(stderr, "Failed to retrieve image dimensions\n");
        }
        TIFFClose(tiff);
    } else {
        fprintf(stderr, "Failed to open TIFF file: %s\n", filename);
    }
}

int main() {
    TIFFSetErrorHandler(errorHandlingCallback);
    const char* filename = "example.tif";
    processTIFFFile(filename);
    return 0;
}
```

Even for arguably simple usage of the libtiff library, the provided code snippet demonstrates the complexity of creating the driver. The process of reading TIFF image files requires careful consideration of various factors and dependencies within the library. The correct usage requires opening the TIFF file using the `TIFFOpen()` function and subsequently retrieving image metadata information with multiple `TIFFGetField()` calls. The `TIFFReadRGBAImage()` function, responsible for reading the image data, depends on the successful execution of both `TIFFOpen()` and `TIFFGetField()` calls. The code also has important error-handling measures, such as checking for file opening and memory deallocation to prevent leaks. Moreover, the `TIFFSetErrorHandler()` function expects a callback as the argument. This example shows the importance of careful handling and adherence to library-specific requirements.

2.3.2 LibFuzzer

LibFuzzer [4] is an **in-process**, coverage-guided, evolutionary fuzzing engine developed by Google that specializes in library testing.

LibFuzzer is a coverage-guided evolutionary fuzzing engine designed for in-process fuzzing. It is integrated with the library being tested and operates by feeding fuzzed inputs through a specific fuzzing entry point, also known as the "target function." As the fuzzing process unfolds, LibFuzzer tracks the code coverage achieved by the inputs and generates mutations on the existing corpus to maximize the coverage. The code coverage information utilized by LibFuzzer is provided by LLVM's SanitizerCoverage instrumentation.

To use LibFuzzer on a library, the first step is to implement a fuzz target function (or as we called it before - the driver). This function receives an array of bytes as input and performs interesting operations on the data using the library's API. The fuzzing engine executes the fuzz target multiple times with different inputs within **the same process**. The fuzz target should be tolerant of various input types, including empty, large, or malformed inputs. It should not `exit()` on any input and ideally joins all threads within the function. The fuzz target should also strive for speed, avoiding complex operations, excessive memory consumption, and global state modifications. Narrower targets are preferred, with each target focused on a specific aspect or format.

Coverage-guided fuzzing heavily relies on a corpus of sample inputs. This corpus should be initialized with a diverse collection of valid and invalid inputs relevant to the code under test. For instance, in the case of a graphics library, the initial corpus might contain a variety of small PNG or JPG files. LibFuzzer generates random mutations based on these sample inputs, exploring new code paths in the process. If a mutation triggers the execution of a previously uncovered path, it is saved to the corpus for further mutations and added to the queue. The corpus also serves as a sanity and regression check, ensuring that the fuzzing entry point remains functional and that the sample inputs successfully pass through the code under test without encountering issues.

As you see, LibFuzzer’s requirements for its target function largely reassembles the challenges that we defined in the previous subsection.

Listing 2.2: LibFuzzer fuzz target signature

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {  
    DoSomethingWithData(Data, Size);  
    return 0;  
}
```

Custom mutators and dictionaries are important features supported by LibFuzzer, enhancing the efficiency of fuzzing.

Custom mutators allow testers to define their own mutation functions tailored to the specific characteristics of the target library or input format. This enables the generation of more targeted and meaningful mutations, increasing the likelihood of discovering vulnerabilities or triggering specific code paths. Custom mutators can be designed to manipulate specific data structures, and modify certain fields or properties. Custom mutators can significantly improve the quality of the generated test cases.

Dictionaries, on the other hand, serve as a valuable resource for guiding the fuzzing process toward relevant and interesting areas of the code. A dictionary is a list of input values or keywords that have significance within the library’s API or expected input format. By providing a dictionary to LibFuzzer, testers can bias the generation of test cases towards these predefined values, increasing the chances of triggering specific functionalities or exercising critical code paths. Dictionaries can be especially useful in situations where certain inputs or parameter values are known to have a significant impact on the behavior of the library. By focusing the fuzzing efforts on these important values, testers can efficiently explore the space of inputs and increase the likelihood of finding vulnerabilities or uncovering unexpected behaviors. For instance, if we are fuzzing library that handles PNG files, triggering some parts of the codebase might depend on the input file having so-called critical chunks: **IHDR**, **PLTE**, **IDAT**, and **IEND**. A dictionary with these keywords would give us massive speedup as random bitflips will take a long time to produce them.

2.4 Related Works

The field of automatic driver generation for library fuzzing has seen significant advancements in recent years. Several notable works have emerged in top-tier conferences, including FuzzGen [7], UTopia [8], FUDGE [2], and RUBICK [13]. These tools aim to automate the process of generating drivers that interact with libraries, enabling effective fuzzing campaigns. In order to evaluate the effectiveness of these automatic approaches, it is essential to compare them with manually written drivers, which have traditionally been used for library fuzzing. For this purpose, we consider

manually written drivers from the OSS-Fuzz project. By examining the strengths and weaknesses of both automatic and manual driver generation techniques, we can gain valuable insights into the advancements made in automatic driver generation. This chapter presents the core ideas of related works.

2.4.1 FUDGE

FUDGE is a system designed to speed up the creation of fuzz drivers. While the drivers it generates still require some input and fixes from the tester, the tool's main concept is that it can automatically synthesize fuzz drivers that are semantically correct by analyzing how the target library is used in the existing codebase. FUDGE processes the entire Google codebase, including third-party libraries, searching for instances where the target library is utilized, extracting relevant information, and using it to create drivers. These generated fuzzers are then executed using LibFuzzer, and FUDGE filters out uninteresting drivers based on their runtime behavior. The tool presents various metrics to the developer, such as code coverage, size of the generated driver, and observed crashes, through a UI. Developers can then adapt and modify the fuzz targets based on this feedback.

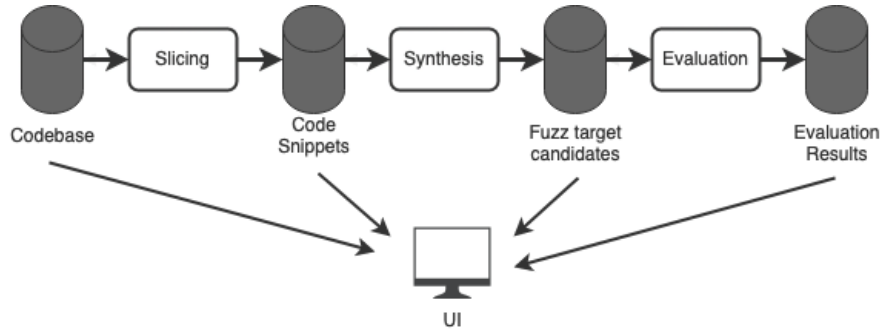


Figure 2.3: High-level overview of the FUDGE.

2.4.2 FuzzGen

FuzzGen analyzes unit tests inside library and library consumers, i.e., programs or other libraries on a system (Android, Debian, etc.) that use the target library. The tool consists of three parts: an API inference, an *Abstract API Dependence Graph* (A^2DG) construction mechanism, and a driver generator that uses A^2DG .

For the API inference, FuzzGen discovers all declared functions in the library and all functions used in the system. The intersection of these two sets gives us which functions of the library are actually being used, everything else can be ignored. The A^2DG is a graph that records all API interactions, including parameter value range and possible interactions. Next, generic A^2DG is constructed by merging different consumers. The driver generator component leverages this A^2DG

to automatically synthesize fuzzers capable of building complex states. FuzzGen then uses LibFuzzer to fuzz individual API components.

FuzzGen was developed concurrently and independently from FUDGE. While the core ideas behind these two projects might be similar, there are a few differences. Firstly, FUDGE extracts code snippets from a single consumer, and created drivers are then tested dynamically. In contrast, FuzzGen iterates CFG, i.e., minimizing consumers. Secondly, FUDGE creates multiple small fuzz drivers from snippets, while FuzzGen merges multiple consumers into single A^2DG , allowing the synthesis of arbitrary length drivers (any random walk through this graph is a possible driver). FuzzGen drivers are larger and more generic, allowing to create complex API sequences.

2.4.3 UTopia

UTopia claims that both FuzzGen and FUDGE have fundamental limitation in their approach by relying on consumer code and instead proposes to infer API dependencies by utilizing unit tests.

The high-level idea of UTopia is to take the unit test, which has an API calls with parameters in it, and convert it to fuzz driver by injecting fuzz inputs into some of the API parameters. UTopia runs a so-called root definition analysis to identify where it can inject inputs without affecting valid API usage semantics.

This approach has several advantages.

1. Large number of popular projects already have well-written unit tests.
2. Unit tests are written by library developers, people who have knowledge of intended usage.
3. Testers may use constant values as API parameters inside unit tests. These values are extracted by UTopia and form an initial corpus.
4. Usually unit tests have the so-called `tearDown` function that is run after each execution of the test. This function may be used to ensure that the state, or at least part of it, is reset before each fuzz run.

However, the majority of UTopia crashes are still spurious. Unit test code is used for testing, developers do not intend to use it for executing it multiple times in-process. They will hard code non-essential parameters and neglect error handling since the code is not intended for production.

2.4.4 RUBICK

To solve the challenges faced by previous related works, RUBICK introduces an automata-guided control-flow-sensitive approach for fuzz driver generation. It represents API usage as deterministic

finite automata and utilizes an active automata learning algorithm to distill the API usage patterns. This information is then used to synthesize a single automata-guided fuzz driver, which provides a scheduling interface for the fuzzer. The fuzzer can use this interface to test independent sets of API usage during the fuzzing process. While the abstract of the paper suggests a promising idea and presents short evaluation results, it is important to note that the paper is currently under embargo until the 32nd USENIX Security Symposium, and further details and findings are not yet publicly available.

Chapter 3

Methodology

In this chapter, we will introduce a semi-automatic framework and best practices for evaluating the quality of automatically generated drivers. The objective of this chapter is to provide a comprehensive overview of the framework, highlighting key components. By employing our framework, researchers can gain valuable insights into the performance, effectiveness, and potential limitations of the automatically generated drivers.

3.1 High Level description

Our analysis has two parts: library analysis, which focuses on static information obtained from the target library source code, and driver analysis, which involves dynamic metrics gathered during the execution of the generated drivers.

The library analysis phase is relatively straightforward. It involves extracting information such as the number of exposed APIs, which provides insight into the area of the library that can be fuzzed. Additionally, we identify the APIs that expect callback parameters, the APIs that utilize variable argument lists, and the APIs that expect objects as parameters. These metrics help us understand the characteristics and requirements of the target library, as well as the complexity of generating synthetic drivers. The driver analysis phase focuses on dynamic metrics gathered during the execution of the generated drivers. One key metric is the number of crashes encountered during the fuzzing campaigns. We take note of the total number of crashes, the number of unique crashes, and the number of non-reproducible crashes. Unfortunately, human involvement is still required to triage clustered crashes to identify false positives, bugs, or vulnerabilities. However, these metrics help us assess the stability of the driver. Another important metric we consider is the coverage achieved by a single driver, which includes information about basic blocks, functions, lines covered. This metric provides insight into how well the driver explores the codebase of the target library and

help us identify areas that require further fuzzing. We also calculate the aggregate coverage of all drivers for the project, providing an overall measure. Furthermore, we analyze the number of API calls made inside the driver itself, which gives us insights into the complexity of interactions of the driver with the library. Additionally, we explore the depths of call stacks for each function call within the driver. This data can be parsed in various ways to extract metrics such as maximum depth or average depths that drivers reach. By using the above-mentioned metrics, we aim to assess the quality of the synthetic drivers.

In the next sections, we will discuss the key parts of the framework in more detail.

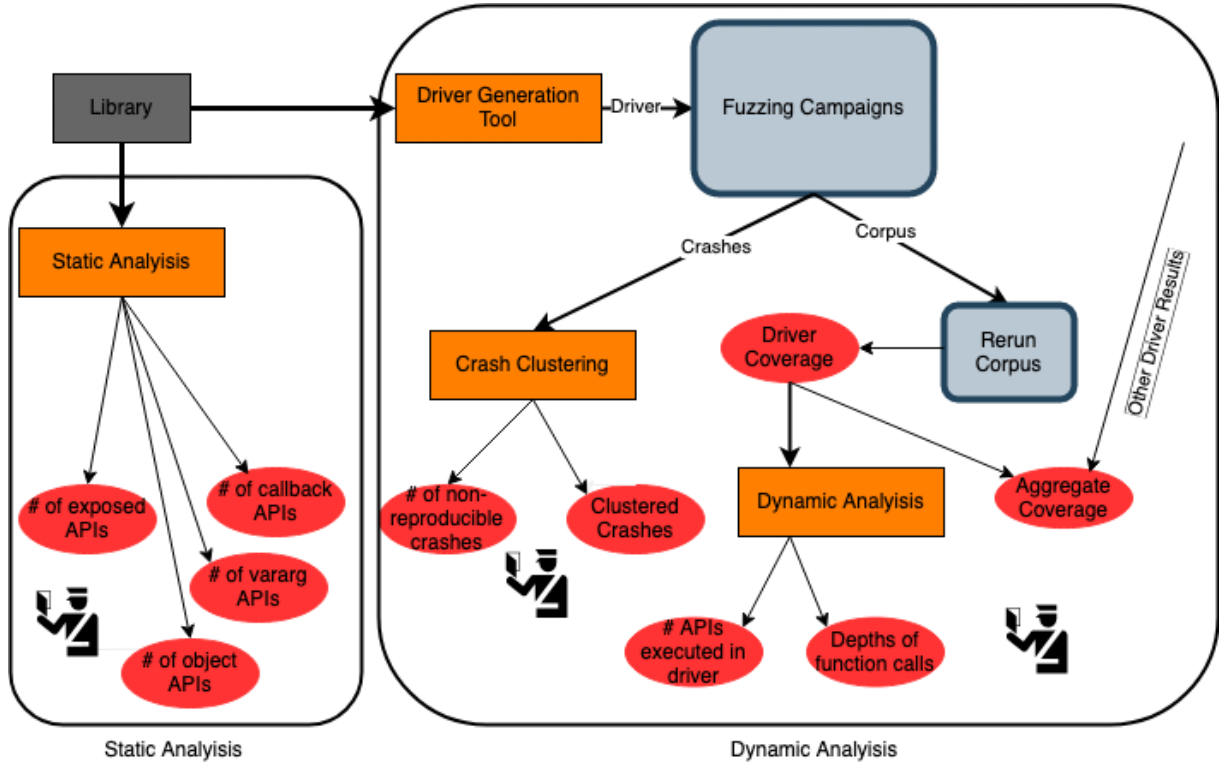


Figure 3.1: High-level overview

3.2 Design of the fuzzing campaigns

Once the drivers have been compiled with the necessary instrumentations, the fuzzing process can be initiated. However, before starting the fuzzing, there are two crucial parameters that need to be considered: time and computational resources. These parameters play a significant role in determining the efficiency and effectiveness of the fuzzing campaign.

Firstly, it is important to decide how many fuzzers should be run in parallel. It is recommended

to choose the number of physical cores available minus one and assign each core to a specific fuzzer. This allows for optimal utilization of computational resources and ensures that the fuzzers can run concurrently without interference with each other.

Next, the duration for which the fuzzers should run needs to be determined. Time is a limited and valuable resource, and users need to set constraints based on their specific requirements. It is essential to maintain consistency across different fuzzers generated from the same tool. When comparing different tools or manually written drivers, one approach is to define the total available time resource and allocate it proportionally among the fuzzers. For example, if a budget of 100 hours is defined, and a tool like UTopia generates 50 synthetic drivers while there are 10 manually written drivers in OSS-Fuzz, each UTopia fuzzer would run for 2 hours, and each OSS-Fuzz fuzzer would run for 10 hours. This ensures a fair comparison and allows for an equal allocation of time resources among different fuzzers.

By carefully considering the number of parallelly running fuzzers and the allocated fuzzing time, users can optimize their fuzzing process to make the most efficient use of available computational resources and ensure consistent evaluation of different fuzzers or driver generation tools.

Once the decisions regarding the number of parallel fuzzers and the allocated fuzzing time have been made, the fuzzing process can be initiated using LibFuzzer within isolated and constrained Docker containers.

Listing 3.1: Full command for running LibFuzzer inside Docker container

```
docker run \  
--rm \  
--cpuset-cpus $CPU_ID \  
--shm-size=2g \  
-d \  
--name $fuzz_target \  
-v $(pwd):/$target_tool \  
-t $CONTAINER_NAME \  
timeout $TIMEOUT $FUZZ_BINARY $CORPUS -fork=1 -ignore_timeouts=1 -ignore_crashes=1  
-ignore_ooms=1 -artifact_prefix=$CRASHES
```

Upon completion of the fuzzing campaign, two relevant folders will be obtained: "crashes" and "corpus". The "crashes" folder contains all the inputs that caused the fuzzer to crash during the fuzzing process. These inputs are particularly interesting as they may indicate potential vulnerabilities or bugs within the library. On the other hand, the "corpus" folder contains all the inputs that the fuzzing engine found "interesting" based on coverage metrics. These inputs have contributed to achieving higher code coverage within the targeted library and may have triggered unique execution paths or uncovered previously unknown behaviors.

It is important to acknowledge that fuzzing is a probabilistic process, and the results obtained

from a single fuzzing campaign may not be representative of the overall performance of a fuzzer. To ensure statistically confident comparisons between any two given fuzzers based on specific metrics, it is recommended to run fuzzing campaigns multiple times, at least five.

3.3 Crash clustering

After fuzzing is over typically get a significant number of crashing inputs. However, it is important to note that not all of these inputs necessarily represent unique bugs within the target library. To address this issue, we use CASR [12]. CASR is a tool for determining the similarity between crashes by analyzing their respective call stacks. CASR can effectively identify and eliminate duplicate crashes. The tool first focuses on removing duplicate crashes, ensuring that each unique crash is represented only once in the final set. Next, CASR uses a hierarchical clustering method to group the remaining crashes based on their similarities, see Figure 3.2.

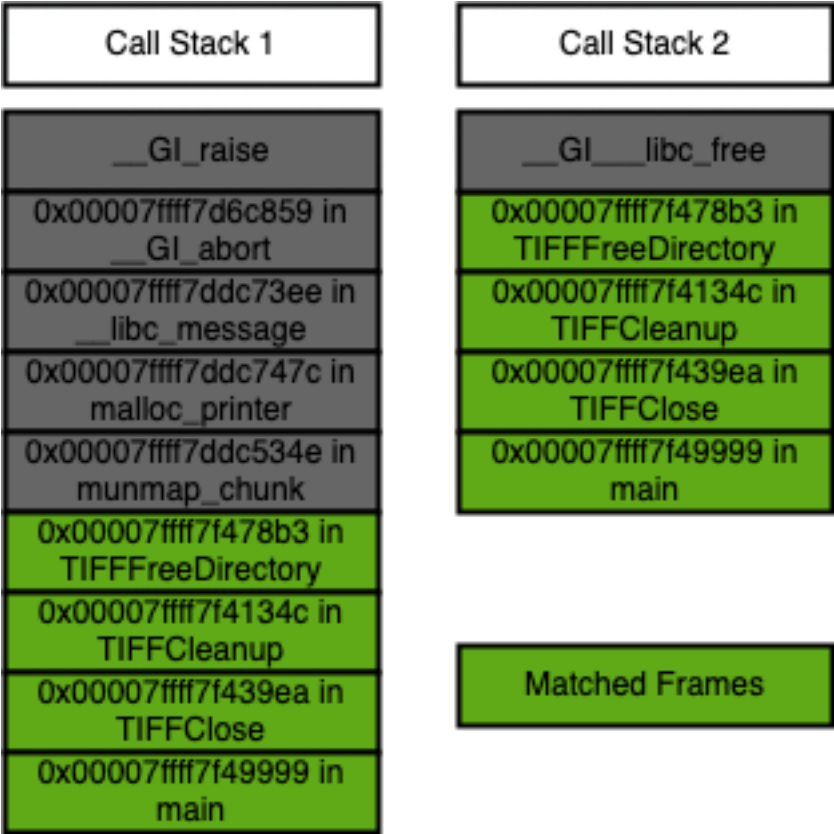


Figure 3.2: Two crashes from the same cluster

One metric that we aim to extract from the crash analysis step is the number of unreproducible crashes. These crashes occur when a driver fails to handle state resetting correctly, leading to

inconsistent or unpredictable behavior during the fuzzing process. As LibFuzzer is an in- process fuzzing engine, the state of the library is carried over between fuzzing iterations. If the driver does not reset its internal state properly after each fuzz input, it can result in unreproducible crashes. Hence we can use this number to assess the quality of the driver as it shows how well the driver handles the state.

The most straightforward way to get this number is to rerun the fuzzer with crashing inputs and see how many of them succeed. However, since CASR already reruns these inputs, we might just parse the log file to find such misleading inputs.

After deduplication and clustering are done, we are left with a number of unique crashes. Unfortunately, a human analyst needs to go through these inputs to determine whether it is a bug in a library or in a driver itself (false positive) and assess if it is an exploitable vulnerability.

3.4 Coverage metrics

The first step to getting coverage metrics is corpora minimization. Corpora minimization involves reducing the size of the initial corpus of interesting inputs generated during the fuzzing process. The purpose of corpora minimization is to remove redundant or unnecessary inputs while still preserving the coverage achieved by the remaining inputs. By eliminating redundant inputs, corpora minimization allows for more efficient analysis and exploration of the remaining inputs.

To minimize corpora, we will run the same LibFuzzer fuzzer with special `-merge=1` flag:

```
mkdir corp_min && ./fuzzer -merge=1 ./corp_min ./corp_original
```

Next, we need to compile the fuzz driver with coverage instrumentation flags and run this coverage-instrumented fuzzer (also known as profile) over the minimized corpora:

```
./fuzzer_cov_instrumented -runs=0 ./corp_min
```

Once the fuzzer has completed iterating through all the inputs, it generates a `.profrac` file that contains highly accurate code coverage information. To extract the coverage metrics, the next step is to convert this `.profrac` file into a profile data file using the `llvm-profdata` tool [10]. This tool processes the raw profile data and prepares it for further analysis. After obtaining the profile data file, we will use the `llvm-cov` tool to extract various metrics of our fuzzing campaign. `llvm-cov` [9] provides a wide range of functionalities to analyze profile data files, including generating reports, identifying uncovered code regions, and calculating coverage percentages at different granularities such as basic blocks, functions, and lines. Scripts can be further adapted to the user's concrete use case, for example, we can use the list of "challenging" APIs that we inferred during static analysis, to see what percentage of them have been fuzzed.

3.5 Driver Complexity

To gain insights into how deeply fuzzer explores the library, we use GDB. From the previous step we already have coverage information, which contains all the executed functions, we strategically place the breakpoints at these function calls. Then we rerun the fuzzer and when breakpoints are hit, we log the backtrace using gdb. By collecting stack trace dumps at various breakpoints, we obtain a nice view of the functions triggered by the fuzzer. This trace dump can be further processed to extract different metrics of interest.

One important metric is the maximum depth reached by the fuzzer, which indicates the furthest level of function calls from "LLVMFuzzOneInput". This metric provides an understanding of the fuzzer's ability to explore deeper levels of the library. Additionally, the average of max depths of each function provides a measure of the overall depth of function calls. We can also combine these metrics with per-function coverage to see how being deep in the library correlates with how much the function is fuzzed.

Furthermore, by analyzing the stack trace dump, we can identify the APIs directly called within the driver code. This number will be dynamic and might not represent the true value that can be obtained statically, however, we argue that this is a more important metric, since for example tools like FuzzGen are able to generate drivers with an arbitrary number of API calls, which makes static information useless for comparison. We currently do not infer this number statically, but it is planned to add that feature, since such information would be useful for more flexible driver generation tools to create drivers that are similar to ones created by other tools for direct comparison.

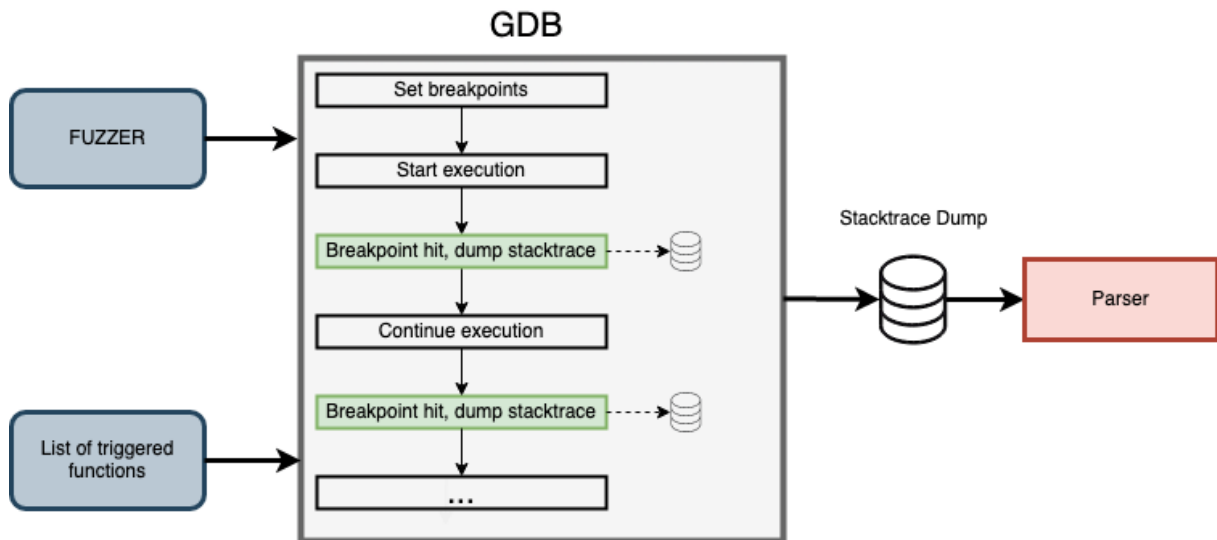


Figure 3.3: Stacktrace analysis with GDB

Chapter 4

Evaluation

We have evaluated automatically generated drivers from UTopia and manually written ones from OSS-Fuzz. Experiments were run on Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz with 16GiB memory and 6 physical cores.

Each driver was compiled with ASan and fuzzer ran 8 hours in an isolated docker container with assigned CPU for 5 iterations.

For fairness to OSS-Fuzz we used existing initial corpora when available, while UTopia drivers started fuzzing with a small corpus that the tool extracts from unit tests.

In this chapter, we present our findings.

4.1 Target Libraries

In our experiments, we focused on evaluating five target libraries: `assimp`, `libaom`, `libhttp`, `libvpx`, and `wabt`. These libraries were chosen to provide a diverse range of characteristics, including the number of lines of code and programming languages used (C and C++). Furthermore, both UTopia and OSS-Fuzz have found vulnerabilities in these libraries.

To gather information about the libraries, we performed a static analysis. In Table 4.1, "Exposed APIs" refers to the total number of APIs exposed by the library, and Callback/Complex/Vararg provides the breakdown of APIs that expect callback, complex, or vararg parameters.

The analysis of the libraries' exposed functions provides valuable insights into their API characteristics. From the metrics presented in the table, it is evident that only a small number of exposed functions expect vararg parameters. Furthermore, we have manually verified that most of these vararg APIs are used for logging purposes. This indicates that vararg parameters are not extensively

used within the libraries' APIs.

Additionally, the analysis reveals that the libraries have limited usage scenarios where callback functions need to be provided by the users as API parameters. It is possible that the libraries follow a more straightforward design approach.

On the other hand, a significant proportion of the exposed APIs expect complex data types as parameters. This indicates that libraries often require structured or composite data to perform their intended functionality.

Table 4.1: LoC = Lines of Code, PL = Programming Language, U = UTopia, O = OSS-Fuzz

Library	LoC	PL	U drivers	O drivers	Exposed APIs	Callbacks	Complex Types	Varargs
assimp	356k	C++	7	1	4269	0	253	2
libaom	363k	C	6	1	5065	0	2132	2
libhttp	20k	C	3	2	386	0	341	1
libvpx	248k	C	4	2	1467	0	631	2
wabt	47k	C++	5	5	1034	0	697	5

4.2 Crashes

While UTopia-generated drivers resulted in significantly higher number of reports compared to manually written drivers, it is important to note that a large portion of these reports are false positives. We managed to reproduce 7 bugs that were already reported by UTopia team to the maintainers of these libraries.

When analyzing the non-reproducible reports for the UTopia-generated drivers, it is observed that the numbers are relatively high, see Figure 4.1. This can be attributed to the inherent challenges of in-process fuzzing engines, where factors such as incorrect state resetting sporadic crashes. On the other hand, the manually written drivers used in OSS-Fuzz did not produce any non-reproducible reports, indicating a higher level of stability in handling the state and reliability in terms of crash reproduction.

4.3 Coverage

In this section, we present the coverage metrics obtained from the UTopia and OSS-Fuzz drivers, highlighting the differences in coverage between synthetic and manually written drivers.

Table 4.4 and Table 4.5 provide the coverage metrics of UTopia and OSS-Fuzz drivers, respectively, at different granularities such as regions, functions, lines, and branches.

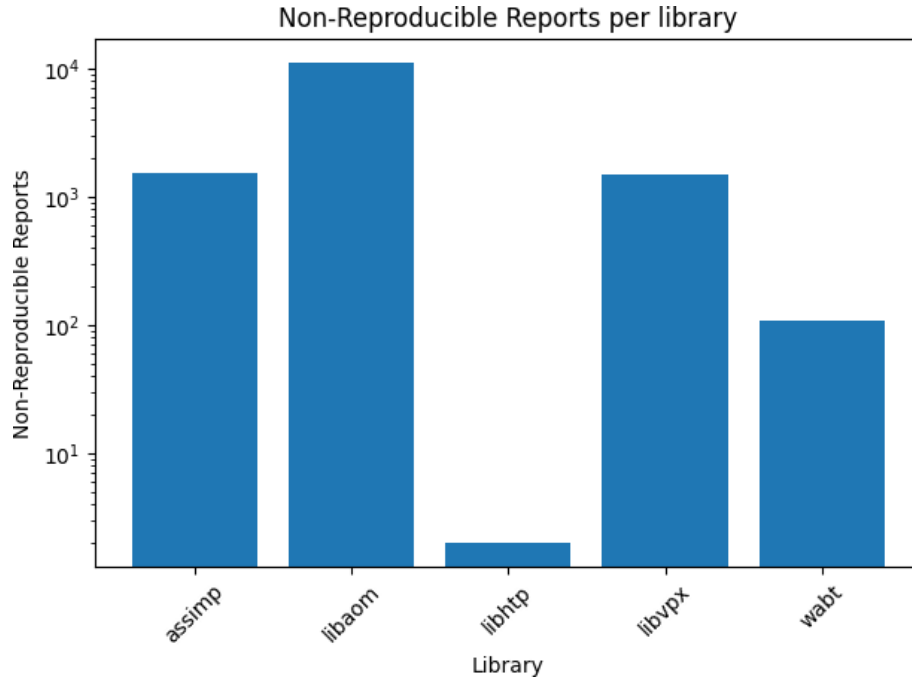


Figure 4.1: UTopia non-reproducible reports

These metrics provide valuable insights into the effectiveness of the drivers in exploring the library code. Higher coverage indicates a more thorough exploration of the library, potentially leading to the discovery of bugs and vulnerabilities.

While the coverage achieved by automatically generated drivers may be much lower, it is important to consider the context in which they are used. While we mostly focused on comparing single manual and automatic drivers, arguably the primary goal of UTopia is to generate a large number of diverse drivers automatically, leveraging their collective power to uncover potential issues in the target library. In contrast, manually written drivers are crafted with specific knowledge and expertise, which often results in higher coverage but with a limited number of drivers.

Figure 4.2 compares the coverage of specific functions by UTopia and OSS-Fuzz. Each function is associated with a color representing its region coverage percentage. The functions are ordered the same way, and 1D heatmaps are plotted for both UTopia and OSS-Fuzz.

In addition to the comparison between UTopia and OSS-Fuzz drivers, we conducted another experiment to evaluate the impact of using a constant-based starting corpus on the coverage achieved by UTopia fuzzers. UTopia extracts constant values from unit tests to create a simple starting corpus for fuzzing. The hypothesis behind this approach is that using meaningful constant values as initial inputs may lead to a more effective exploration of the library code.

To validate this hypothesis, we ran UTopia fuzzers for 1 hour with and without the starting

Table 4.2: Summary of Fuzzing Results for UTopia. TR = Total Reports, UR = Unique Reports, TO/OOM = Time out & out of memory

Library	Driver	Metrics					
		TR	UR	Clusters	Crashes	Leaks	TO/OOM
assimp	1	8256	77	26	73	3	216
	2	0	0	0	0	0	0
	3	0	0	0	0	0	0
	4	0	0	0	0	0	0
	5	28237	2	2	1	1	0
	6	30371	1	1	1	0	0
	7	0	0	0	0	0	0
libaom	1	0	0	0	0	0	0
	2	27	4	2	1	3	15
	3	11242	4	2	0	4	0
	4	0	0	0	0	0	0
	5	0	0	0	0	0	0
	6	0	0	0	0	0	0
libhttp	1	33293	3	1	3	0	0
	2	12057	4	3	3	1	0
	3	46097	1	1	1	0	0
libvpx	1	1451	1	1	0	1	843
	2	26	1	1	0	1	1
	3	615	8	2	8	0	0
	4	1479	0	0	0	0	0
wabt	1	51566	4	1	4	0	2
	2	0	0	0	0	0	0
	3	0	0	0	0	0	0
	4	0	0	0	0	0	0
	5	1	1	1	1	0	0

corpus. The goal was to check whether these initial inputs would result in significant differences in the achieved coverage. Table 4.6 presents the coverage metrics obtained from this experiment.

Interestingly, the results indicate that the starting corpus did not give a substantial difference in the coverage. This finding can be explained by the simplistic nature of the constant values extracted from unit tests. In many cases, testers do not invest significant effort in generating complex or meaningful inputs for unit tests, leading to relatively basic constants in the starting corpus that can also easily be generated randomly.

Table 4.3: Summary of Fuzzing Results for OSS-Fuzz. TR = Total Reports, UR = Unique Reports, TO/OOM = Time out & out of memory

Library	Driver	Metrics					
		TR	UR	Clusters	Crashes	Leaks	TO/OOM
assimp	assimp_fuzzer	3241	18	12	8	10	0
libaom	av1_dec_fuzzer	0	0	0	0	0	0
libhttp	fuzz_htp	0	0	0	0	0	0
	fuzz_diff	0	0	0	0	0	0
libvpx	vpx_dec_fuzzer_v9	0	0	0	0	0	0
	vpx_dec_fuzzer_v8	0	0	0	0	0	0
wabt	read_binary_interp	745	4	3	4	0	0
	read_binary_ir	246	6	3	6	0	0
	wasm2wat	1	1	1	1	0	0
	wat2wasm	0	0	0	0	0	0
	wasm_objdump	13537	6	4	6	0	0

4.4 Driver Complexity

In this chapter, we examine the complexity of the drivers used in our evaluation. Table 4.7 and Table 4.8 present the driver complexity metrics for each driver: number of APIs and number of unique APIs. We can observe that the complexity of manually written drivers, with the exception of libhttp, is generally not too high. These drivers have a similar level of complexity as UTopia drivers. However, as we saw in terms of coverage, it becomes evident that OSSFuzz drivers outperform UTopia drivers. This finding supports our hypothesis that unit tests, although important for functional testing, may often focus on a specific part of the codebase, resulting in limited coverage. On the other hand, library developers, with their comprehensive understanding of the library, can strategically select a small number of API calls that can achieve high coverage across various code paths.

Figure 4.3 shows the maximum depths at which each function was called for both UTopia and OSS-Fuzz. It is evident that manually written drivers in OSS-Fuzz demonstrate better reach.



(a) Color scale

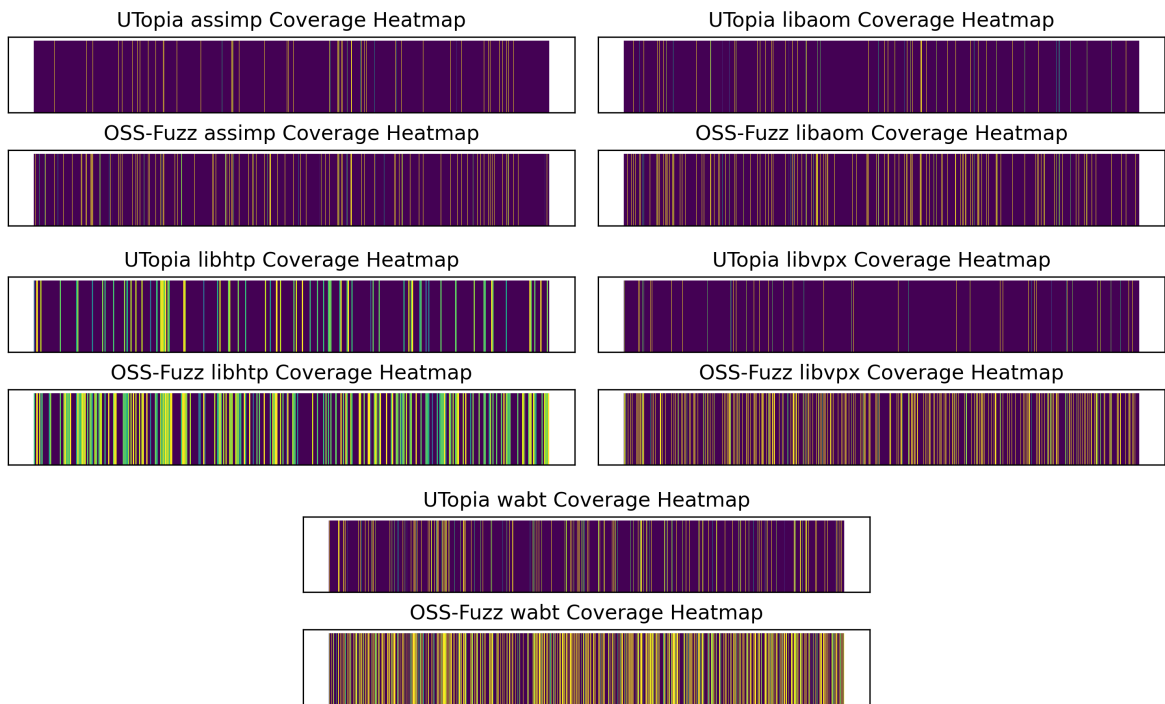


Figure 4.2: Comparison of per-function coverage for UTopia and OSS-Fuzz

Table 4.4: Summary of UTopia Coverage

Library	Driver	Metrics in %			
		Region Cover	Function Cover	Line Cover	Branch Cover
assimp	1	1.32	3.51	1.66	0.82
	2	0.14	0.44	0.17	0.05
	3	1.47	4.21	1.76	0.92
	4	2.47	5.06	2.60	1.77
	5	2.48	5.06	2.62	1.78
	6	1.14	3.78	1.33	0.66
	7	0.14	0.58	0.20	0.06
	TOTAL	3.77	6.75	4.03	2.85
libaom	1	0.48	0.99	0.23	0.25
	2	22.63	24.04	18.10	17.87
	3	6.66	3.57	3.31	3.80
	4	0.14	0.65	0.13	0.06
	5	0.17	0.72	0.15	0.10
	6	0.19	0.86	0.17	0.10
	TOTAL	22.89	24.39	18.47	18.12
libhtp	1	0.48	4.37	1.51	0.15
	2	4.19	13.59	8.51	3.77
	3	0.67	5.44	1.92	0.24
	TOTAL	5.04	16.22	9.30	4.17
libvpx	1	1.17	3.97	0.83	0.55
	2	0.21	1.39	0.24	0.09
	3	7.17	9.66	5.45	4.21
	4	0.92	3.09	0.61	0.37
	TOTAL	8.19	12.35	6.13	4.66
wabt	1	7.67	8.54	8.46	14.35
	2	0.25	1.03	0.89	0.13
	3	0.59	1.93	1.24	0.23
	4	0.75	1.80	1.15	0.26
	5	3.20	4.85	11.51	2.76
	TOTAL	9.98	13.17	19.17	15.86

Table 4.5: Summary of OSS-Fuzz Coverage

Library	Driver	Metrics in %			
		Region Cover	Function Cover	Line Cover	Branch Cover
assimp	1	9.15	14.49	7.39	6.65
	TOTAL	9.15	14.49	7.39	6.65
libaom	1	62.66	56.35	55.96	59.85
	TOTAL	62.66	56.35	55.96	59.85
libhttp	1	32.56	47.26	38.41	34.78
	2	12.55	10.38	12.37	11.43
	TOTAL	38.73	52.40	44.69	40.76
libvpx	1	41.20	19.91	21.30	41.54
	2	61.39	59.73	58.48	63.70
	TOTAL	63.07	61.03	59.76	63.67
wabt	1	53.27	34.59	40.98	45.61
	2	72.07	45.50	60.51	74.84
	3	71.99	57.41	63.52	73.58
	4	58.28	38.40	50.94	65.90
	5	14.24	22.57	30.29	20.70
	TOTAL	77.06	72.45	78.66	86.22

Table 4.6: Coverage with and without Constant-based Starting Corpus after 5 iterations

Library	With Starting Corpus		Without Starting Corpus	
	Coverage	Stdev	Coverage	Stdev
assimp	4.07%	0.256	3.77%	0.312
libaom	23.2%	0.831	22.89%	0.456
libhttp	5.76%	0.352	5.04%	0.882
libvpx	8.16%	0.628	8.19%	0.773
wabt	10.2%	0.772	9.98%	0.762

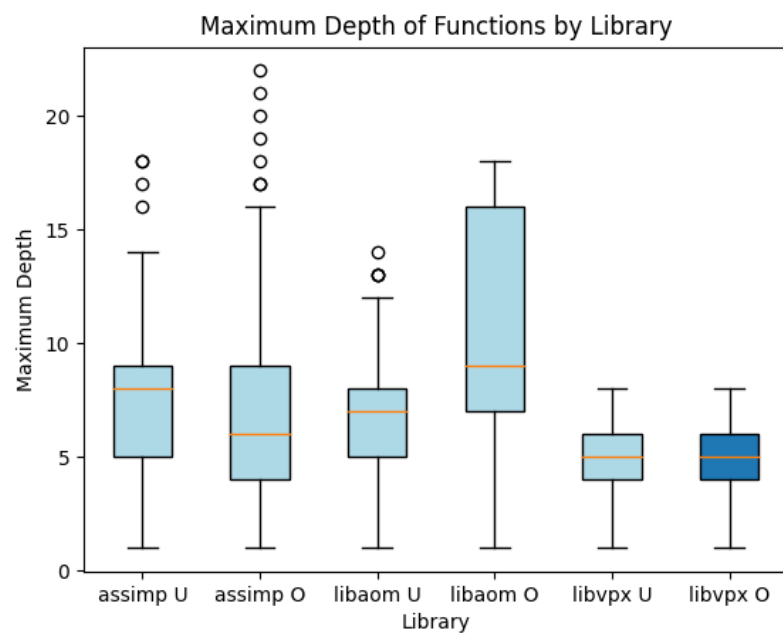


Figure 4.3: Maximum depths of each function. U = UTopia, O = OSS-Fuzz

Table 4.7: UTopia driver complexity

Library	Driver	Metrics	
		Number of APIs	Number of unique APIs
assimp	1	1	1
	2	1	1
	3	1	1
	4	1	1
	5	1	1
	6	5	4
	7	3	2
libaom	1	3	3
	2	4	4
	3	8	4
	4	3	3
	5	1	1
	6	1	1
libhttp	1	1	1
	2	10	5
	3	4	4
libvpx	1	14	7
	2	5	5
	3	4	3
	4	2	2
wabt	1	4	1
	2	2	2
	3	3	3
	4	4	4
	5	3	3

Table 4.8: OSS-Fuzz driver complexity

Library	Driver	Metrics	
		Number of APIs	Number of unique APIs
assimp	assimp_fuzzer	4	4
libaom	av1_dec_fuzzer	6	6
libhttp	fuzz_htp	33	25
	fuzz_diff	47	27
libvpx	vpx_dec_fuzzer_v9	7	7
	vpx_dec_fuzzer_v8	7	7
wabt	read_binary_interp	2	2
	read_binary_ir	2	2
	wasm2wat	1	1
	wat2wasm	3	3
	wasm_objdump	5	1

Chapter 5

Conclusion

We have explored the state-of-the-art in automatic driver generation for library fuzzing, focusing on developing a methodology and framework for evaluating synthetic drivers. We began by discussing the complexities of library fuzzing, including the need for a deep understanding of the library APIs, seed selection, callback handling, error handling, and state management. These challenges highlight the importance of developing automated solutions and represent the main issues these tools need to tackle. While synthetic drivers may not match the reach and efficiency of manually written ones, the advantage of tools like UTopia is their ability to generate a large number of drivers. Despite the individual drivers', the cumulative exploration power of a vast number of drivers can rival that of a few manually written ones.

In conclusion, our project has demonstrated the value and potential of automatic driver generation for library fuzzing. We believe that our methodology and framework contribute to the advancement of software security practices, providing a more efficient and effective approach to library fuzzing.

Bibliography

- [1] Apache. *Log4j Vulnerability*. <https://logging.apache.org/log4j/2.x/security.html>. Accessed: 08.06.2023.
- [2] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, 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*. ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, 2019, pp. 975–985.
- [3] *Chrome: 70% of all security bugs are memory safety issues*. <https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues/>. Accessed: 08.06.2023.
- [4] Google. *LibFuzzer*. <https://www.llvm.org/docs/LibFuzzer.html>. Accessed: 08.06.2023.
- [5] Google. *OSS-Fuzz Project*. <https://github.com/google/oss-fuzz>. Accessed: 08.06.2023.
- [6] Google. *OSS-Fuzz Trophies*. <https://google.github.io/oss-fuzz/#trophies>. Accessed: 08.06.2023.
- [7] 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. ISBN: 978-1-939133-17-5. URL: <https://www.usenix.org/conference/usenixsecurity20/presentation/ispoglou>.
- [8] Bokdeuk Jeong, Joonun Jang, Hayoon Yi, Jiin Moon, Junsik Kim, Intae Jeon, Taesoo Kim, WooChul Shim, and Yong Ho Hwang. “UTOPIA: Automatic Fuzz Driver Generation using Unit Tests”. In: *Proceedings of the 44th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, May 2023.
- [9] LLVM. *llvm-cov - emit coverage information*. <https://www.llvm.org/docs/CommandGuide/llvm-cov.html>. Accessed: 08.06.2023.
- [10] LLVM. *llvm-profdata - Profile data tool*. <https://www.llvm.org/docs/CommandGuide/llvm-profdata.html>. Accessed: 08.06.2023.

- [11] *Microsoft: 70 percent of all security bugs are memory safety issues*. <https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/>. Accessed: 08.06.2023.
- [12] Georgy Savidov and Andrey Fedotov. “Casr-Cluster: Crash Clustering for Linux Applications”. In: *2021 Ivannikov ISPRAS Open Conference (ISPRAS)*. IEEE. 2021, pp. 47–51. DOI: 10.1109/ISPRAS53967.2021.00012.
- [13] Cen Zhang, Yuekang Li, Hao Zhou, Xiaohan Zhang, Yaowen Zheng, Xian Zhan, Xiaofei Xie and Xiapu Luo, Xinghua Li, Yang Liu, and Sheikh Mahbub Habib. *Automata-Guided Control-Flow-Sensitive Fuzz Driver Generation*. <https://www.usenix.org/conference/usenixsecurity23/presentation/zhangcen>. Accessed: 08.06.2023.