Fuzzing Programs with Structure Aware Fuzzers

A Mini Project Report

submitted to the APJ Abdul Kalam Technological University

in partial fulfillment of the requirements for the award of degree

Bachelor of Technology

in

Computer Science and Engineering

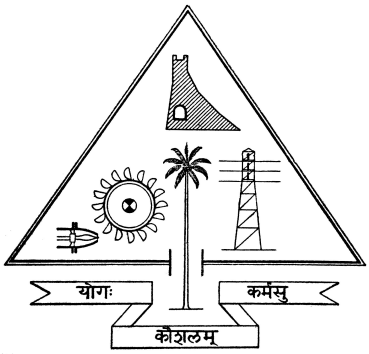
by

Darshana Das K(TCR21CS020)

Krishnahari P(TCR21CS032)

Libna Kuriakose T(TCR21CS033)

Parvathy C M(TCR21CS049)



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

GOVERNMENT ENGINEERING COLLEGE, THRISSUR

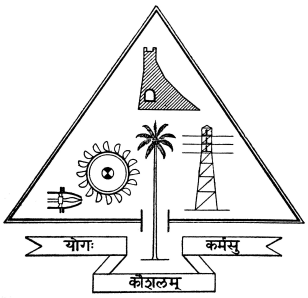
KERALA

JUNE 2024

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

GOVERNMENT ENGINEERING COLLEGE

THRISSUR - 680009



CERTIFICATE

This is to certify that the mini project report entitled **Fuzzing Programs with Structure Aware Fuzzers** submitted by **Darshana Das K** (TCR21CS020), **Krishnahari P** (TCR21CS032), **Libna Kuriakose T** (TCR21CS033) & **Parvathy C M** (TCR21CS049) to the APJ Abdul Kalam Technological University in partial fulfillment of the B.Tech. degree in Computer Science and Engineering is a bonafide record of the project work carried out by him/her under our guidance and supervision. This report in any form has not been submitted to any other University or Institute for any purpose.

Dr. Ezudheen P
(Project Guide)
Assistant Professor
Department of CSE
GEC Thrissur

Dr. Ezudheen P
(Project Coordinator)
Assistant Professor
Department of CSE
GEC Thrissur

Dr. Ajay James
Associate Professor and HoD
Department of CSE
GEC Thrissur
Thrissur

DECLARATION

We hereby declare that the project report **Fuzzing Programs with Structure Aware Fuzzers** , submitted for partial fulfillment of the requirements for the award of degree of Bachelor of Technology of the APJ Abdul Kalam Technological University, Kerala is a bonafide work done by us under supervision of **Dr. Ezudheen P**.

This submission represents our ideas in our own words and where ideas or words of others have been included, we have adequately and accurately cited and referenced the original sources.

We also declare that I have adhered to ethics of academic honesty and integrity and have not misrepresented or fabricated any data or idea or fact or source in our submission. We understand that any violation of the above will be a cause for disciplinary action by the institute and/or the University and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been obtained. This report has not been previously formed the basis for the award of any degree, diploma or similar title of any other University.

Darshana Das K

Krishnahari P

Libna Kuriakose T

Parvathy C M

Thrissur

April 24, 2024

Abstract

This project is under the guidance of Dr. Rahul Gopinath, Lecturer at University of Sydney. Fuzzing is an automated software testing technique where invalid, unexpected, or random data is given as input into a program in the hope of uncovering bugs, crashes, or vulnerabilities. In recent years, structure aware fuzzers have gained attention due to their ability to better understand the input structure of programs and generate more effective test cases. Fuzzing programs that handle structured binary inputs presents a difficult challenge due to the specific input format these programs require. While current fuzzers are versatile in their approach, they often lack effectiveness when it comes to targeting a particular format. This work aims to develop a structure aware fuzzer to improve the efficiency of fuzzing techniques.

In this work, we introduce a generation-based structure-aware fuzzer, which is utilizing the format defined in Kaitai Struct. Additionally, we enhance the structure-aware fuzzing in libFuzzer by integrating Kaitai Struct. Kaitai Struct's capability to define complex data structures in a declarative manner makes it well-suited for use as input to our tool. This work addresses the limitations of traditional fuzzing methods by leveraging knowledge of the program's structure to guide the generation of input data.

Keywords: structure aware fuzzing, Kaitai Struct, libFuzzer

Acknowledgement

We take this opportunity to express our deepest sense of gratitude and sincere thanks to everyone who helped us to complete this work successfully. We express our sincere thanks to **Dr. Ajay James**, Head of Department, Computer Science and Engineering, Government Engineering College, Thrissur for providing us with all the necessary facilities and support.

We would like to place on record our sincere gratitude to our project guide **Dr. Ezudheen P**, Assistant Professor, Computer Science and Engineering, Government Engineering College, Thrissur and our external guide **Dr. Rahul Gopinath**, Lecturer, University of Sydney, for the guidance and mentorship throughout this work.

Finally we thank our family, and friends who contributed to the successful fulfilment of this project work.

Darshana Das K

Krishnahari P

Libna Kuriakose T

Parvathy C M

Contents

List of Figures

List of Tables

ABBREVIATIONS

KS Kaitai Struct

KSY Kaitai Struct YAML

AFL American Fuzzy Lop

List of Symbols

Chapter 1

Introduction

1.1 Introduction

Software testing is the process of running an application with the intent of finding software bugs. It leads to minimizing errors and cutting down software costs [1]. As attacks on vulnerabilities is one of the primary methods of cybersecurity breaches, it's crucial to detect and resolve these weaknesses. For instance, the WannaCry ransomware attack exploited a vulnerability in the Server Message Block (SMB) protocol. This attack quickly spread, infecting more than 230,000 computers across 150 countries within a single day, resulting in significant losses [11]. Fuzzing, an essential software testing technique, has gained prominence for its ability to uncover vulnerabilities in software programs. This concept was first developed by Barton Miller in 1988 [13].

The process of fuzzing begins by generating inputs for the target program. It aims to detect vulnerabilities by feeding these generated inputs to the target applications and monitoring the execution states [11]. Hence, Fuzzing is "a highly automated testing technique that covers numerous boundary cases using invalid data (from files, network protocols, API calls, and other targets) as application input to better ensure the absence of exploitable vulnerabilities" [12].

A fuzzer can be categorized as either generation-based or mutation-based, depending on how it generates test inputs. In generation-based fuzzing, test inputs are created from scratch based on the known format or grammar of the input. Conversely,

mutation-based fuzzing involves modifying existing, well-formed test inputs to create new ones [3].

Traditionally, fuzzing uses either random inputs or random mutations of valid inputs. Since it relies on randomness, fuzzing may miss security violations that rely on unique corner-case scenarios. To address this limitation, there has been recent work on "smart" input generation for fuzzing, based on domain-specific knowledge of the target system [2]. By incorporating an understanding of the data's structure into the fuzzing process, these advanced techniques can generate test cases that effectively enhances code coverage. This approach has the potential to uncover a broader spectrum of bugs and vulnerabilities.

Kaitai Struct is a declarative language used to describe various binary data structures, laid out in files or in memory: i.e. binary file formats, network stream packet formats, etc [4]. In addition to Kaitai Struct, another pivotal component in the realm of fuzzing is libFuzzer— a library tailored for in-process fuzzing. LibFuzzer is adept at conducting fuzz testing on individual functions or modules within a program, allowing for granular examination of code segments.

1.2 Problem Statement

Despite advancements in fuzz testing techniques, effectively fuzzing programs that handle structured binary inputs remains a challenge. Programs processing binary data, such as multimedia files, demand very specific input formats due to the complex nature of their structures [14]. While coverage-guided and feedback-directed fuzzing tools like libFuzzer and AFL have demonstrated significant success in detecting vulnerabilities, they often lack structure awareness, making them less effective for testing programs with specific input requirements.

There are libraries like libprotobuf mutator that can be integrated with libFuzzer to enable structure-aware fuzzing [16]. However, it's important to note that this approach relies on Protocol Buffers (protobuf), a serialization format primarily designed for data interchange rather than comprehensive structure representation [15]. As a result, using protobuf may not provide sufficient structure awareness for complex binary file formats.

Kaitai Struct offers a unique advantage by providing a declarative language for describing binary file formats, enabling precise and comprehensive modeling of complex structures. However, despite its potential benefits, there is currently no relevant tool that seamlessly integrates Kaitai Struct with fuzzing, leaving a gap in structure-aware fuzz testing capabilities.

Thus the problem addressed by this project is the absence of structure-aware fuzz testing tools for software that deals with complex binary inputs. Current fuzzing tools, while effective for general testing, often fail to accurately handle intricate binary file formats, resulting in limited vulnerability detection.

1.3 Objectives

Our research centers on employing structure-aware fuzzers to test programs. Specifically, we develop a Python-based generation fuzzer that utilises the descriptive power of Kaitai Struct (KSY) files to accurately model binary data structures and generate tailored test cases. Additionally, the project provides a novel approach that enhances libFuzzer's capabilities to effectively fuzz test individual code segments, by integrating the structure awareness brought by Kaitai Struct. This integration enables comprehensive testing of programs with complex data structures. Thus the project seeks to provide security researchers and developers with a robust framework for structured fuzz testing, facilitating the discovery of security vulnerabilities in software applications. The objectives are:

- Develop a Python-based generation fuzzer that incorporates structure awareness by utilizing Kaitai Struct as input.
- Ensure that the developed fuzzer can generate test cases that achieve at least 90 percent coverage when parsed by open-source parsers for that file format.
- Develop a library for handling Kaitai Struct integration with libFuzzer to enable structure awareness.
- Ultimately, aim to create a tool for security researchers and developers to conduct structure-aware fuzzing.

1.4 Novelty of Idea and Implementation Steps

This research introduces a novel approach to fuzz testing by utilising Kaitai Struct, a declarative language for describing binary file formats. The novelty of this idea lies in its development of a generation fuzzer tailored to generate test cases specific to a given file format, along with a library for enhancing structure-aware fuzzing using libFuzzer.

While there are tools available that offer structure aware fuzzing, none of them harness the descriptive power of Kaitai Struct and there exist no libraries that seamlessly integrate Kaitai Struct with libFuzzer, thereby limiting the fuzzing capabilities of libFuzzer in the context of structured binary inputs. By incorporating Kaitai Struct into the fuzzing process, this research aims to provide a more precise and comprehensive modeling of complex data structures within binary files. This innovative approach has the potential to significantly advance fuzz testing techniques and enhance the detection of vulnerabilities in software applications.

The implementation steps are

- Library for File Formats in Kaitai Struct: In addition to the existing format library of Kaitai Struct, we are expanding the library to include descriptions for more file formats. This enhancement will facilitate researchers' access to a wider range of file formats, thereby enabling them to utilize Kaitai Struct more effectively.

- Generation Fuzzer: The generation fuzzer generates random bits that are assigned to each field specified in the Kaitai Struct. This process ensures the creation of diverse and comprehensive test cases, covering various scenarios and data inputs for thorough fuzz testing.

- Integration with libFuzzer: The library will be designed to seamlessly integrate Kaitai Struct descriptions with the fuzzing process within the libFuzzer framework.

1.5 Societal and Industrial Relevance

Cybersecurity is of critical importance in today's world. Organizations have expended enormous amounts of resources to secure themselves. However, despite those efforts, data breaches — in which hackers steal personal data — continue to increase year-on-year: there was a 20% increase in data breaches from 2022 to 2023 [17].

In this context, fuzzing emerges as a crucial tool for software developers to strengthen cybersecurity defenses. It helps detect coding errors and security vulnerabilities in software, operating systems, or networks. Fuzzing systems excel at uncovering various vulnerabilities, including buffer overflow, denial of service (DoS), cross-site scripting, and code injection [18].

The importance of fuzzing in both societal and industrial contexts cannot be overstated. In society, the prevalence of cyber threats underscores the need for robust cybersecurity measures to protect personal and sensitive data. In industry, the integration of fuzzing into software development processes is essential for ensuring the security and reliability of software products.

Furthermore, the advent of structure-aware fuzzing introduces a new framework in fuzz testing. By utilising descriptive languages like Kaitai Struct, structure-aware fuzzing enhances the precision and effectiveness of fuzz testing, particularly for programs with complex data structures. This innovation holds significant promise for industries reliant on software applications, as it enables more thorough testing and detection of vulnerabilities.

Thus our project provides developers with an essential framework to enhance structure-aware fuzzing, crucial for strengthening cybersecurity defenses, safeguarding sensitive information, and reducing costs. Furthermore, using Kaitai Struct and libFuzzer encourages collaboration among security researchers and developers. They can easily share and reuse fuzzing definitions and test cases, speeding up the process of finding and fixing vulnerabilities in different software projects. This collaborative approach follows the principles of open-source development and information security, promoting innovation and collective progress in software security.

Chapter 2

Literature Review

2.1 Topic

Literature Review: Fuzzing Programs with Structure Aware Fuzzers

2.2 Introduction

The purpose of this literature review is to delve into existing research and relevant literature regarding the field of fuzzing programs with structure-aware fuzzers. The project proposes a novel approach to software testing, utilizing structure-aware fuzzing techniques to enhance vulnerability detection in software applications by utilising descriptive languages like Kaitai Struct to accurately model complex data structures.

This review will explore various studies and methodologies related to fuzz testing, focusing on the integration of structure awareness into fuzzing tools. Key areas of investigation will include the algorithms and methodologies employed in structure-aware fuzzing, such as the generation of tailored test cases based on file format specifications and the integration of structure-awareness with existing fuzzing frameworks like libFuzzer. Additionally, the review will examine the potential benefits and limitations of structure-aware fuzzing compared to traditional fuzz testing methods.

2.2.1 Fuzzing

The seminal paper authored by Miller, Fredriksen, and So in 1990 marked a significant milestone in software testing, particularly in the development of fuzzing techniques. Titled ”An empirical study of the reliability of UNIX utilities,” this paper laid the groundwork for fuzzing as a method for evaluating the robustness of software applications.

The study details a project aimed at rigorously testing various Unix utilities by subjecting them to random input strings. The objective was to detect bugs and potential security vulnerabilities within the system. Tools developed for this purpose included a fuzz generator for generating random character streams and ptyjig for testing interactive utilities. These tools played a pivotal role in conducting systematic tests on the Unix utilities, using varied input strings to stress-test the programs and uncover any underlying bugs or vulnerabilities. The results of these tests were meticulously analyzed to identify and classify program bugs that led to crashes or other unexpected behavior.

A key insight gleaned from the study is the recognition of the limitations of formal verification when applied to large, complex systems like Unix utilities. Instead, the use of randomized input strings proved effective in detecting errors that may have eluded traditional testing methods.

LibFuzzer and AFL (American Fuzzy Lop) are two prominent fuzzing tools widely used in the field of software testing and vulnerability discovery. LibFuzzer, developed by Google, is renowned for its effectiveness in discovering vulnerabilities [9]. It has successfully uncovered numerous security flaws in various C and C++ codebases, contributing significantly to improving software reliability and security.

American Fuzzy Lop (AFL) is a renowned coverage-guided fuzzing tool acclaimed for its effectiveness in uncovering vulnerabilities. Developed by Michal Zalewski, AFL utilizes innovative feedback-driven techniques to guide the mutation process, maximizing code coverage and enhancing bug detection [22]. AFL has been instrumental in identifying security flaws across various software projects, demonstrating its significant impact on improving software reliability and security.

2.2.2 Structure Aware Fuzzing

In recent years, there have been notable advancements in the field of fuzzing programs with structure-aware techniques, aimed at enhancing vulnerability discovery in software systems. These developments have introduced various sophisticated methodologies.

One significant contribution is FormatFuzzer, developed by Rafael Dutra, Rahul Gopinath, and Andreas Zeller from CISPA Helmholtz Center for Information Security [19]. FormatFuzzer introduces a format-specific approach to fuzzing, utilizing binary templates to generate C++ code for parsing, mutating, and generating inputs. It effectively handles complex formats such as MP4 or ZIP files and integrates with format-agnostic fuzzers like AFL for intelligent mutations and seed evolution, thereby enhancing fuzzing outcomes.

AFLSmart, as introduced in the paper ”AFLSmart: Smart Greybox Fuzzing” [23] revolutionizes coverage-based greybox fuzzing (CGF) by leveraging high-level structural representations of seed files. Unlike traditional CGF methods, AFLSmart employs innovative mutation operators operating on the virtual file structure to explore new input domains while maintaining file validity. AFLSmart achieves significantly more branch coverage than baseline AFL, discovering 42 zero-day vulnerabilities in widely-used tools and libraries, demonstrating its effectiveness in enhancing software security.

Another notable advancement is presented in the study ”Superion: Grammar-Aware Greybox Fuzzing for Structured Inputs” by Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu [20]. Superion enhances traditional coverage-based greybox fuzzing by incorporating grammar awareness, facilitating more effective trimming and mutation of test inputs while maintaining their validity and structure. Experimental studies on XML and JavaScript engines demonstrate Superion’s improved code coverage and bug-finding capabilities compared to AFL, uncovering 31 new bugs, including 21 vulnerabilities with 16 CVEs assigned.

In their paper titled “Fast Format-Aware Fuzzing for Structured Input Applications” the authors propose an efficient approach for vulnerability discovery in applications with structured input [21]. They propose a fast format-aware fuzzing

approach to recognize dependencies from the specified input to the corresponding comparison instruction. Their developed prototype FFAFuzz significantly reduces time overhead, outperforming existing techniques like Redqueen and WEIZZ, making it a valuable contribution to enhancing security testing for structured input applications.

Another notable development in the field of fuzz testing is Fuzz4All, as presented in the paper ”Fuzz4All: Universal Fuzzing with Large Language Models.” This groundbreaking approach to fuzz testing leverages large language models (LLMs) to achieve universal fuzzing capabilities across various input languages and features. Evaluation across nine systems under test utilizing six different languages, including C, C++, Go, SMT2, Java, and Python, demonstrates that Fuzz4All achieves higher coverage compared to existing, language-specific fuzzers. Moreover, Fuzz4All has successfully identified 98 bugs in widely used systems, such as GCC, Clang, Z3, CVC5, OpenJDK, and the Qiskit quantum computing platform, with 64 bugs confirmed by developers as previously unknown.

Another notable development is the libprotobuf-mutator [16], designed to randomly mutate protocol buffers. This library, intended for use with guided fuzzing engines like libFuzzer, significantly enhances fuzz testing for applications utilizing protocol buffers, contributing to improved software reliability and security.

2.3 Kaitai Struct in Fuzzing

The thesis ”Fuzzing Self-Described Structures” by Kathleen Abols [24] discusses the proactive use of fuzzing to identify errors and vulnerabilities in legacy data formats, with a specific focus on maritime cyber security and S-57 naval charts. The research presents an approach for parsing and generating data for self-defining data formats by leveraging a mixed data-type file format. The research project delves into the parsing and manipulation of data within the ISO/IEC 8211 format, utilizing tools like Kaitai Struct to define the grammar and automatically generate parsers for the self-describing data format. By developing the ParseENC parsing framework, which extends Kaitai’s capabilities to handle the context-sensitive nature of ISO 8211, the study aims to create grammar-conscientious mutated charts for intelligent fuzzing activities. Through experimentation and analysis, the thesis highlights the importance

of understanding and working with self-describing data structures for effective security testing and vulnerability assessment in complex file formats.

2.4 Conclusions and Gap Analysis

The literature review highlights significant advancements in the field of fuzzing programs with structure-aware fuzzers, showcasing the effectiveness of various techniques in enhancing vulnerability detection and improving software reliability and security. From seminal works like Miller et al.'s empirical study of UNIX utilities to recent developments such as AFLSmart and Superion, researchers have continuously pushed the boundaries of fuzz testing methodologies. Moreover, the integration of Kaitai Struct, as highlighted in the ParseENC framework, exemplifies the importance of leveraging descriptive languages to accurately model and manipulate self-describing data structures for intelligent fuzzing activities. However, despite these advancements, there still exists a gap in the availability of a unified framework that seamlessly integrates the descriptive power of tools like Kaitai Struct with the fuzzing capabilities of widely-used frameworks like libFuzzer. While AFLSmart and libprotobuf mutator excel in certain aspects, they may lack the flexibility or compatibility required for broader adoption across different software testing scenarios.

2.5 Summary

The project title, "Fuzzing Programs with Structure Aware Fuzzers," and its objectives are directly influenced by the findings of the literature review. The review underscores the importance of structure awareness in fuzzing techniques and the need for a unified framework that combines descriptive power with fuzzing capabilities. As a result, the project aims to develop a Python-based generation fuzzer that harnesses the descriptive capabilities of Kaitai Struct. Additionally, it provides an approach to develop a library for integration with libFuzzer. This approach aligns with the identified gap in existing solutions and seeks to address it.

Chapter 3

Background

3.1 Fuzzing

Fuzzing is an effective and widely used technique for finding security bugs and vulnerabilities in software. It inputs irregular test data into a target program to try to trigger a vulnerable condition in the program execution [25]. Rather than relying solely on predefined test cases, fuzzing generates inputs automatically, aiming to explore as much of the software's behavior space as possible. This exploration often reveals hidden bugs and security flaws that may have gone unnoticed during traditional testing methods.

Fuzzing can be applied to various types of software, including applications, libraries, and protocols, making it a versatile tool in the quest for software reliability and security. Its effectiveness lies in its ability to simulate real-world scenarios and stress-test software under conditions that developers may not have anticipated. Despite its power, fuzzing faces challenges such as handling complex input formats, achieving comprehensive code coverage, and managing performance overhead. Overcoming these challenges requires continual innovation and collaboration within the research and development community. Overall, fuzzing plays a crucial role in enhancing software quality and security, helping to create safer and more robust digital environments for users worldwide.

3.2 Binary File Formats

In a binary file format specification, fields are named and have as attributes a data type, length and sometimes a constant value. Fields are offset at addresses relative to the beginning of a file [26]. Unlike text files, binary files are not human-readable and may contain a wide variety of data, including images, videos, audio, executables, archives, databases, document files (such as PDFs and Word documents) and more. They are typically viewed as sequences of bytes.

Some binary files contain headers, which are blocks of metadata used by programs to interpret the file's data. Headers often include a signature or magic number to identify the format; for instance, a GIF file's header may contain text like GIF87a or GIF89a. If a binary file lacks headers, it may be termed a flat binary file.

Effective fuzzing of programs handling structured binary inputs, such as multimedia files, poses a challenge due to their reliance on specific input formats [7].

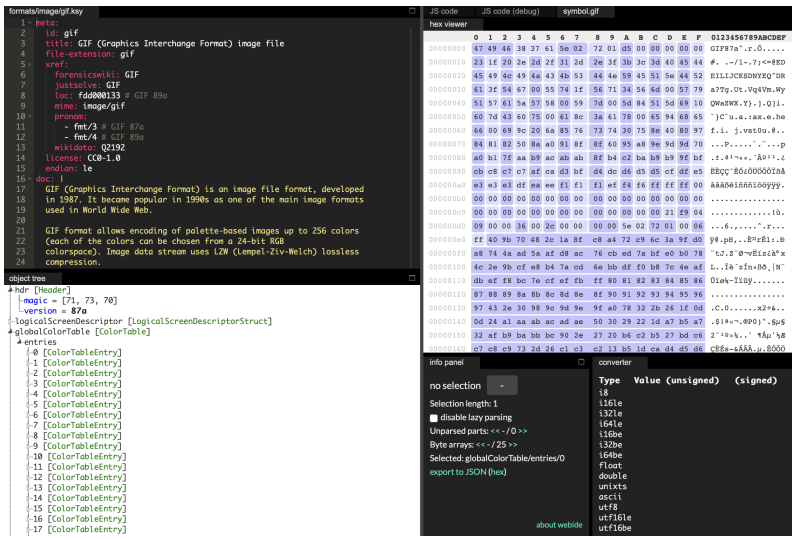
Binary file formats with a similar file structure are referred to as a family of file formats. There are two families of binary file formats that are readily distinguished: the chunk-based and the directory-based file formats. These two families do not exhaust the possible file structures. For instance, there are executable binary file formats and file header-body file formats that have different file structures [26].

```
00000000  47 49 46 38 37 61 5e 02  72 01 d5 00 00 00 00 00  |GIF87a^.r.......|
00000010  23 1f 20 2e 2d 2f 31 2d  2e 3f 3b 3c 3d 40 45 44  |#. .-/1-.?;<=@ED|
00000020  45 49 4c 49 4a 43 4b 53  44 4e 59 45 51 5e 44 52  |EILIJCKSDNYEQ^DR|
00000030  61 3f 54 67 00 55 74 1f  56 71 34 56 6d 00 57 79  |a?Tg.Ut.Vq4Vm.Wy|
00000040  51 57 61 5a 57 58 00 59  7d 00 5d 84 51 5d 69 10  |QWaZWX.Y}.].Q]i.|
00000050  60 7d 43 60 75 00 61 8c  3a 61 78 00 65 94 68 65  |`}C`u.a.:ax.e.he|
00000060  66 00 69 9c 20 6a 85 76  73 74 30 75 8e 40 80 97  |f.i. j.vst0u.@..|
00000070  84 81 82 50 8a a0 91 8f  8f 60 95 a8 9e 9d 9d 70  |...P.....`.....p|
00000080  a0 b1 7f aa b9 ac ab ab  8f b4 c2 ba b9 b9 9f bf  |................|
00000090  cb c8 c7 c7 af ca d3 bf  d4 dc d6 d5 d5 cf df e5  |................|
000000a0  e3 e3 e3 df ea ee f1 f1  f1 ef f4 f6 ff ff ff 00  |................|
000000b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000c0  00 00 00 00 00 00 00 00  00 00 00 00 00 21 f9 04  |.............!..|
000000d0  09 00 00 36 00 2c 00 00  00 00 5e 02 72 01 00 06  |...6.,....^.r...|
000000e0  ff 40 9b 70 48 2c 1a 8f  c8 a4 72 c9 6c 3a 9f d0  |.@.pH,....r.l:..|
*
00001980  6d d5 d5 57 61 55 28 08  00 3b                    |m..WaU(..;|
0000198a
```

Figure 3.1: Binary representation of a GIF image displayed through hexdump, * represents rest of the bytes.

3.3 Kaitai Struct

Kaitai Struct is a declarative language used to describe various binary data structures, laid out in files or in memory: i.e. binary file formats, network stream packet formats, etc. The basic idea of Kaitai Struct is that a particular format can be described using Kaitai Struct language (in a .ksy file), which then can be compiled using kaitai-struct-compiler into source files in one of the supported programming languages. These modules will include a generated code for a parser that can read described data structure from a file / stream and provide access to its contents in a nice, easy-to-comprehend API [4]. For a long time, you could only use Kaitai Struct for parsing, not serialization (writing data to file). However, serialization support has been added to Kaitai Struct [27].



(a) Kaitai Struct Web IDE

```
meta:
  id: gif
  file-extension: gif
  endian: le
seq:
  - id: header
    type: header
  - id: logical_screen
    type: logical_screen
types:
  header:
    seq:
      - id: magic
        contents: 'GIF'
      - id: version
        size: 3
  logical_screen:
    seq:
      - id: image_width
        type: u2
      - id: image_height
        type: u2
      - id: flags
        type: u1
      - id: bg_color_index
        type: u1
      - id: pixel_aspect_ratio
        type: u1
```

(b) KSY File of GIF

Figure 3.2: Kaitai Struct

3.4 Generation based fuzzing

Generation-based fuzzing, starts from a specification, which describes the file format or network protocol, and constructs test cases from these documents. The key to making effective test cases is to make each case differ from valid data so as to hopefully cause a problem in the application, but not to make the data too invalid, or else the target application may quickly discard the input as invalid. Generation-based

fuzzing requires a significant amount of up-front work to study the specification and manually generate test cases. Sometimes manually generated test cases become too similar to the specification and do not differ in the unpredictable ways that benefit generation-based fuzzing. Regardless, intuition says that the extra knowledge gained by understanding the format should result in higher quality test cases. The comparison between mutation-based and generation-based fuzzing techniques, as illustrated in Miller and Peterson's 2007 analysis on PNG file format, indicates that generation-based fuzzing achieves significantly higher code coverage compared to mutation-based fuzzing, particularly in scenarios involving uncommon inputs [5].

3.5 Integration with existing fuzzer

While traditional generation-based fuzzers like csmith or Peach excel at generating inputs according to predefined grammars, they are typically limited to targeting a single input type. On the other hand, coverage-guided mutation-based fuzzers such as libFuzzer or AFL offer greater flexibility by not being restricted to a single input type and not requiring explicit grammar definitions. This makes them easier to set up and use. [10]

Integrating Kaitai Struct with existing fuzzers enhances their capabilities by providing structure-awareness during the fuzzing process. By incorporating Kaitai Struct definitions into fuzzing frameworks like AFL (American Fuzzy Lop) or libFuzzer, the fuzzers gain the ability to generate and mutate test inputs according to the specified binary data format's structure. This integration enables more intelligent and targeted fuzzing, as the fuzzers can generate inputs that adhere to the expected structure of the binary format, increasing the likelihood of triggering meaningful or critical code paths within the target software.

3.5.1 AFLSmart Evaluation

AFLSmart is a robust yet efficient and easy-to-use smart greybox fuzzer based on AFL, a popular and very successful CGF. It leverages a high-level structural representation of the seed files to generate new files. It uses higher-order mutation operators that

work on the virtual file structure rather than on the bit level which allows AFLSmart to explore completely new input domains while maintaining file validity. It uses a novel validity-based power schedule that enables AFLSmart to spend more time generating files that are more likely to pass the parsing stage of the program, which can expose vulnerabilities much deeper in the processing logic. [8]

3.5.2 LibFuzzer Evaluation

LibFuzzer is an in-process, coverage-guided, evolutionary fuzzing engine. Google's libFuzzer was a part of the LLVM project and is widely used for automated software testing through fuzzing. It follows a coverage-guided approach that explores new code paths based on feedback.It uses a feedback-driven approach, which means it can continuously mutate the input data based on the feedback from the program's execution, thus helping uncover a wide range of bugs and crashes, including memory corruption issues and security vulnerabilities in C/C++ code. LibFuzzer is similar in concept to American Fuzzy Lop (AFL), but it performs all of its fuzzing inside a single process. This in-process fuzzing can be more restrictive and fragile, but is potentially much faster as there is no overhead for process start-up.

The fuzzer is linked with the library under test, and feeds fuzzed inputs to the library via a specific fuzzing entrypoint. The fuzzer then tracks which areas of the code are reached, and generates mutations on the corpus of input data in order to maximize the code coverage. The code coverage information for libFuzzer is provided by LLVM's SanitizerCoverage instrumentation. Each libFuzzer process is single-threaded, unless the library under test starts its own threads. However, it is possible to run multiple libFuzzer processes in parallel with a shared corpus directory; this has the advantage that any new inputs found by one fuzzer process will be available to the other fuzzer processes [9].

3.5.3 Conclusion

LibFuzzer's ease of integration, deterministic execution, in-process fuzzing capabilities, code coverage feedback, support for sanitizers, and compatibility with multiple programming languages make it a versatile and powerful choice for fuzz

testing. LibFuzzer is designed to be easily integrated into existing build systems and workflows, especially those based on LLVM. Its library-based approach allows developers to fuzz-test individual functions or libraries within their programs without significant modifications to the build process. Furthermore, LibFuzzer is widely preferred by industry practitioners due to its robustness and effectiveness. Additionally, LibFuzzer is compatible with all major operating systems, enhancing its applicability across different development environments.

3.6 LibFuzzer

3.6.1 Structure Aware Fuzzing using libFuzzer

Mutation-based fuzzers may encounter challenges when dealing with complex input types. For instance, indiscriminate mutations like bit flipping can often lead to invalid inputs that are swiftly rejected by the target application during parsing.

Despite these challenges, libFuzzer can be enhanced to support structure-aware fuzzing for specific input types with minimal additional effort [10].

3.6.2 Custom Mutations

Custom mutations in LibFuzzer allow developers to extend the default mutation strategies provided by the fuzzer. While LibFuzzer includes built-in mutation algorithms such as bit flips, byte flips, arithmetic mutations, and dictionary-based mutations, custom mutations enable developers to tailor the fuzzing process to the specific requirements of their target program. To implement custom mutations in LibFuzzer, developers can define their mutation functions that manipulate the input data in ways that are meaningful for the target program [10]. These custom mutation functions can range from simple transformations to more complex operations depending on the characteristics of the input data and the desired fuzzing strategy.

3.6.3 LibProtobuf-mutator

Libprotobuf-mutator is a library specifically designed to apply random mutations to protocol buffers, which are widely used for describing and serializing structured data. It could be used together with guided fuzzing engines, such as libFuzzer. This mutator library is particularly useful for fuzz testing applications that utilize protocol buffers for data interchange.

3.6.4 Protobuf vs Kaitai Struct

Protobuf and Kaitai Struct are both tools used for describing and serializing structured data, but they have different approaches and purposes.

Protobuf (Protocol Buffers) is a language-agnostic mechanism developed by Google for serializing structured data. It uses a language-specific interface description language (IDL) to define the structure of the data, which is then compiled into code in various programming languages. Protobuf focuses on efficiency, compactness, and language interoperability, making it suitable for use in distributed systems and communication protocols.

On the other hand, Kaitai Struct is a declarative language for describing binary file formats. It allows developers to define the structure of binary data using a human-readable format, which can then be compiled into parsers in multiple programming languages. Kaitai Struct emphasizes simplicity, flexibility, and ease of use, enabling developers to quickly create parsers for a wide range of binary file formats.

In our analysis, we found that only around 20 percent of the features provided by Kaitai Struct could be effectively translated to Protobuf. This limited conversion efficiency suggests that attempting to convert Kaitai Struct definitions to Protobuf so as to use Kaitai Struct with libprotobuf-mutator may not be the most efficient approach. While Protobuf offers advantages in terms of language interoperability and efficiency, its suitability for representing complex binary file formats may be limited compared to Kaitai Struct.

Chapter 4

Feasibility Study and Requirements Analysis

4.1 Feasibility

Feasibility refers to the assessment of whether a project is technically, economically,and operationally viable or achievable. Feasibility studies are conducted to evaluate the potential success and practicality of a project before committing significant resources to its implementation.

4.1.1 Technical Feasibility

Technical feasibility corresponds to determination of whether it's technically feasible to develop the software. Here those tools are considered, which will be required for developing the project. The tools, which are available, and tools, which will be required, are taken into account.

1.Development of Generation-Based Fuzzer with Kaitai Struct Integration:
Developing a generation-based fuzzer that derives structure awareness from Kaitai Struct definitions is technically feasible. Python offers robust libraries and tools for working with Kaitai Struct, enabling efficient parsing and manipulation of binary data structures.

2.Integration of Kaitai Struct with libFuzzer:
Integrating Kaitai Struct with libFuzzer through custom mutation logic is techni-

cally feasible. Custom mutation logic can be applied to Kaitai Struct definitions to generate a library, which can then be seamlessly integrated with libFuzzer for in-depth testing.

4.1.2 Economic Feasibility

Economic Feasibility is about the total cost incurred for the system. The software resource requirement of the proposed system is python,kaitai struct and c++.

4.1.3 Time Feasibility

Time feasibility corresponds to whether sufficient time is available to complete the project. Time feasibility is a measure of how reasonable the project timetable is.Given our technical expertise, are the project deadlines reasonable? Some projects are initiated with specific deadlines. It is necessary to determine whether the deadlines are mandatory or desirable.

Parameters considered are:

- Schedule of the project
- Time by which the project has to be completed
- Reporting period
- Requirement gathering
- Open source documentation about libFuzzer and LLVM
- Human resource available

Considering all the above factors the proposed project of developing a generation-based fuzzer with Kaitai Struct integration is technically feasible within the allotted time frame of 3 months. While integration with libFuzzer may require additional time beyond the initial period, careful planning and prioritization can ensure successful implementation and maintenance of the framework within the established timeline.

4.1.4 Legal Feasibility

From a legal perspective, our project, which involves developing a generation-based fuzzer utilizing Kaitai Struct and integrating it with libFuzzer, necessitates thorough consideration of legal implications. We are dedicated to ensuring compliance with pertinent laws and regulations governing software development, data management, and privacy protection. Specifically, we must address legal requirements concerning the integration and utilization of specialized technologies such as libFuzzer, ensuring adherence to relevant legal frameworks and regulations. By emphasizing legal feasibility in our project report, we underscore our commitment to conducting the project in accordance with applicable laws and regulations. This approach aims to minimize legal risks while fostering acceptance and support within the legal framework governing software development and data management practices.

4.1.5 Operational Feasibility

The proposed fuzzer project, which involves developing a generation-based fuzzer using Kaitai Struct and integrating Kaitai Struct with libFuzzer, demonstrates strong operational feasibility within our organization. Firstly, the project aligns seamlessly with our strategic objectives, aiming to enhance software testing capabilities and bolster security measures. We have assessed our resource availability and found that we possess the necessary expertise in Python programming and experience with Kaitai Struct to initiate the development process effectively. Additionally, our existing technical infrastructure can support the implementation and operation of the project, minimizing the need for significant upgrades. Concerning user acceptance and training, we anticipate smooth adoption among our team members, given their familiarity with similar tools and technologies. While potential risks exist, such as compatibility challenges with existing processes and the need for thorough testing, we have devised mitigation strategies to address these concerns proactively. Overall, the operational feasibility of the project is high, with clear alignment with organizational objectives, adequate resource availability, and effective risk management strategies in place.

4.2 Project Requirements

Project requirements refer to the specific functionalities, features, constraints, and expectations that define what the project must deliver to meet its objectives. These requirements serve as the foundation for the project design and guide the development process. They are typically derived from the project's goals, stakeholder needs, and any relevant regulations or standard.

4.2.1 Implementation Requirements

The proposed fuzzer project, which involves developing a generation-based fuzzer using Kaitai Struct and integrating Kaitai Struct with libFuzzer, has the following requirements:

- Programming Languages: Kaitai Struct, Python (for the generation-based fuzzer) and C++ (for libFuzzer).
- Existing Technologies: Understanding of the LLVM project and familiarity with the libFuzzer is essential for implementation.
- Operating System: Working in a Linux operating system environment is preferable.

4.2.2 Deployment Requirements

The deployment of our fuzzer project is specifically tailored for Dr. Rahul Gopinath at the University of Sydney in collaboration with his research, and it has the following requirements:

- Efficiency: The system should efficiently fuzz binary file formats based on the provided KSY files to generate test cases.
- Integration with libFuzzer: The deployment requires modification of libFuzzer, developed by Google, to seamlessly integrate Kaitai Struct with it.
- Input Generation: KSY files for various binary formats need to be developed to serve as input for fuzzing.

Chapter 5

Project Design

Project design involves planning and organizing a project's activities, resources, timelines, and deliverables. It's about strategically outlining how to achieve the project's goals, ensuring smooth coordination and successful execution.

5.1 Introduction

Structure-aware fuzzers are the type of software testing tool designed to efficiently and effectively test programs by leveraging knowledge about the structure of input data. In particular, a generation-based fuzzer gains structure awareness from Kaitai Struct definitions. Additionally, the capabilities of structure-aware fuzzing in libFuzzer are enhanced by integrating Kaitai Struct seamlessly.

5.1.1 Purpose

The purpose of structure-aware fuzzers is to identify and address software bugs and security vulnerabilities more comprehensively and accurately than traditional fuzzing techniques. By understanding the format, layout, and semantics of the data being processed by a program, structure-aware fuzzers can generate or mutate test inputs that are more likely to uncover vulnerabilities, trigger edge cases, and improve code coverage.These fuzzers are particularly valuable for testing programs that handle complex or structured data, such as network protocols, file formats, and data serialization libraries.

5.1.2 Scope

The scope of this system includes the following functionalities:

- designing and refining the fuzzer's algorithm,
- creating a mechanism for utilizing Kaitai Struct definitions
- integrating it into the libFuzzer framework

5.2 Python-based Generation Fuzzer

We are developing a generation fuzzer that is a software testing tool capable of automatically generating test cases for binary file formats. This fuzzer leverages structured representations, such as Kaitai Struct files, to define the format's layout and characteristics. By encoding the structure of different file formats in KSY files, the fuzzer gains the ability to automatically generate valid and semantically meaningful inputs that conform to the specified format. By utilizing KSY files, we gain the ability to specify the structure of different file formats, including their metadata, headers, and data segments.

The primary goal of our method is to automate the generation of diverse and representative test cases for binary file formats. This process precedes the design of a workflow that begins with a testcase database that we curated by studying various file formats. The database comprises approximately 200 Kaitai Struct files (ksy) representing various file formats. Additionally, we've identified open-source parsers for these file formats to evaluate the working of our fuzzer. The KSY file serves as the blueprint for our generation-based fuzzer, providing the necessary information about the layout and organization of the binary data.

5.2.1 Algorithm

The metadata from the YAML data is initially read, where the endianness and file_extension variables are initialized. This process involves iterating through each key-value pair in the metadata. If the key corresponds to 'file-extension', its value is assigned to the file_extension variable; similarly, if the key represents 'endian', its

value is assigned to the endianness variable. Finally, the endianness and file_extension are returned for further processing.

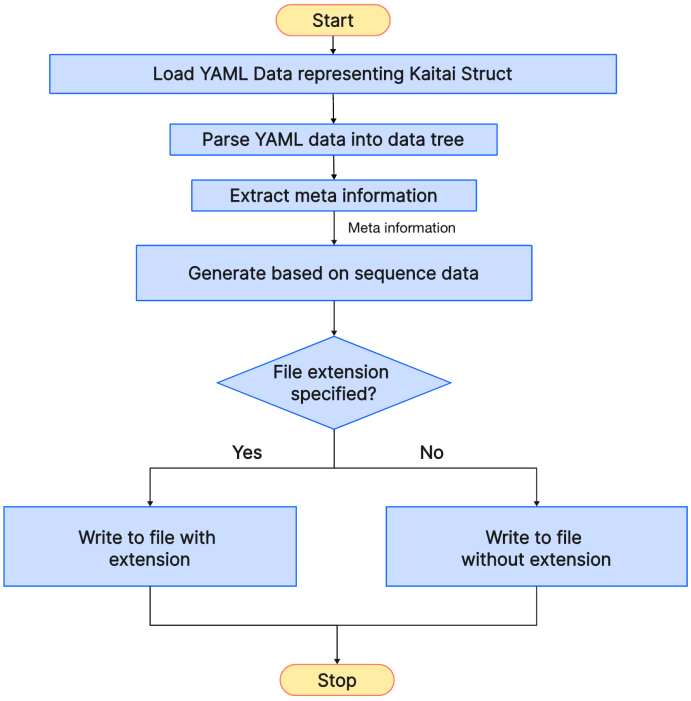


Figure 5.1: High Level representation of python based generation fuzzer

The handling of sequence data begins by initializing the expansion variable to an empty byte string. Subsequently, each item in the sequence undergoes individual processing. This involves initializing variables such as type, size, encoding, endianness, and valid_endian. For each item, specific actions are performed based on its keys. If the key corresponds to 'contents', the pack_list function is invoked to pack the values into binary data. Similarly, if the key represents 'size', its value is assigned to the size variable, and so forth. If the type is None, random binary data is generated based on the specified size and endi- anness using the random_based_on_size function. Conversely, if the type corresponds to one of the predefined types such as 'u2', 'u4', 's2', or 'str', random binary data is generated based on the specified type, size, and endianness using the random_based_on_type function. In cases where the type represents a user-defined type, the handle_type function is recursively called to manage the user-defined type. Ultimately, the processed binary data is returned as the expansion.

The handling of user-defined types involves initializing the expansion variable to an empty byte string. Each key-value pair in the types dictionary is iterated through, and if the key matches the user_defined_type, the handle_seq function is recursively called to manage the sequence defined for the user-defined type. Finally, the processed binary data is returned as the expansion.

5.3 Integration with LibFuzzer

The integration of Kaitai Struct with LibFuzzer enhances the fuzzing process by incorporating structured-awareness into test case generation. By utilizing Kaitai Struct to define the structure of binary file formats, LibFuzzer gains the ability to generate and mutate test inputs that adhere to the specified format. This structured approach to fuzzing allows LibFuzzer to explore deeper into the parsing logic of the target program, increasing the likelihood of uncovering vulnerabilities that may exist in complex data processing stages.

5.3.1 Approach

We propose the development of a comprehensive library tailored for Kaitai Struct to seamlessly integrate with libFuzzer. Our approach involves the implementation of specialized mutation logic specifically designed for Kaitai Struct, enabling effective fuzz testing of complex file formats and data structures. Through this integration, we seek to empower developers with a powerful toolset for uncovering vulnerabilities in software systems handling structured data.

The proposed integration utilises the structured data definitions provided by Kaitai Struct to drive targeted and context-aware mutations. The library parses the Kaitai Struct schema (.ksy file) to comprehend the definition of the data structure, extracting crucial details such as field names, data types (including integers, strings, booleans, etc.), nested structures, arrays, and any specified relationships or constraints articulated within the schema. The core of the library is the mutation engine responsible for generating targeted mutations on the parsed Kaitai Struct objects. This engine will utilize the knowledge extracted from the schema to perform context-aware

modifications.

The mutation engine operates through the following steps:

- Field Selection: Based on the schema, the engine identifies relevant fields for mutation. This can involve prioritizing specific fields based on their criticality or potential vulnerability points within the target software.

- Mutation Techniques: Depending on the data type of the chosen field, the engine applies appropriate mutation techniques. For integers: bit flips, value modifications within a defined range, attempts to overflow the value. For strings: character insertions/deletions, modifying specific characters, manipulating string length based on constraints defined in the schema.

- Context-Aware Mutations: The engine can consider field relationships and nested structures. For instance, if a field represents a URL, the engine might specifically mutate the domain or path components while leaving the protocol intact.

- Validation: After mutation, the engine can validate the modified Kaitai Struct object to ensure it adheres to the schema constraints. This avoids generating nonsensical data that wouldn't be processed by the target software.

The library will provide a mechanism to integrate with libFuzzer. This might involve a custom function that takes the original Kaitai Struct data as input and utilizes the mutation engine to generate a set of mutated versions. The mutated Kaitai Struct objects are then serialized back into a raw byte stream format compatible with libFuzzer for fuzz testing.

5.3.2 WorkFlow

The workflow for integrating Kaitai Struct with libFuzzer involves key steps to facilitate structured fuzz testing of software systems handling complex data formats:

- Schema Parsing: We need to develop a parser that can interpret the Kaitai Struct schema file. This parser will extract the crucial details. By parsing the schema, we establish a blueprint essential for the subsequent fuzz testing process. This

ensures that mutations are applied within the framework of the defined data structure, enabling effective fuzz testing.

- Mutation Engine: The next step in our workflow involves crafting a mutation engine. Following the parsing of the schema, engine identifies relevant fields within the data structure based on the parsed schema and applies targeted mutations accordingly.

- Validation of Mutated Objects: After mutation, our workflow incorporates a validation step for the mutated Kaitai Struct objects. This validation process ensures that the mutated objects conform to the constraints outlined within the schema.

- Serialization of Mutated Objects: After validation, our workflow proceeds with the serialization of the mutated Kaitai Struct objects. This crucial step involves converting the structured data back into raw byte streams. By doing so, the mutated data is prepared for ingestion into libFuzzer, ensuring compatibility with the fuzzing framework.

- Fuzz Testing with libFuzzer: Finally, the serialized byte streams with the mutated data are passed to libFuzzer for fuzz testing the target software. LibFuzzer runs the test cases repeatedly to uncover possible vulnerabilities.

5.4 Data Design

Kaitai Struct is a powerful declarative language and framework for describing various binary file formats. When developing structure-aware fuzzers, the main data used for understanding the structure of input files typically comes from Kaitai Struct YAML (KSY) files.

5.4.1 Data descrption

Structure-aware fuzzers developed using Kaitai Struct can be applied to a wide range of binary file formats. These formats can vary significantly depending on the application domain, but some common examples include:

- Image Formats: Formats like JPEG, PNG, GIF, BMP, and TIFF are used for storing digital images. These formats have complex structures defining image data, metadata, compression methods, color spaces, and more.

- Document Formats: Formats such as PDF, DOCX, XLSX, and PPTX are used for storing text documents, spreadsheets, presentations, and other types of documents. These formats often include metadata, text, images, embedded objects, and formatting information

- Archive Formats: Formats like ZIP, TAR, RAR, and 7z are used for compressing and archiving multiple files and directories into a single file. These formats typically include file headers, compression algorithms, file metadata, and file data.

These are just a few examples, and there are many other binary file formats used in various domains such as multimedia, software development, forensic analysis, data storage, and more. With Kaitai Struct, we can describe the structure of these formats in YAML files and develop structure-aware fuzzers to test parsers for robustness and security.

5.4.2 Data Dictionary

For developing a structure-aware fuzzer using Kaitai Struct or similar tools, we need specific data to effectively understand and manipulate the structure of input files. Here's a breakdown of the types of data typically used:

- Field Names: Descriptive names for each field in the binary format.

- Field Sizes/Lengths: Specify the size or length of each field in bits or bytes.

- Field Endianness: Specify the byte order (big-endian or little-endian) if applicable.

- Contents: Define constraints or limitations on field values

- Enumerations: Define enumerated types for fields with a limited set of possible values.

- Conditions: Define conditions under which certain fields or structures are present or have specific values.
- Comments: Add comments within the Kaitai Struct file to document the structure, rationale, or any other relevant information.

By incorporating such elements into Kaitai Struct YAML files, we can create a structured representation of the binary file format. This representation serves as the basis for generating parsing code and developing structure-aware fuzzers.

5.5 Component Design

Designing a structure aware fuzzer involves breaking down the system into components that handle various aspects of input generation, mutation, and testing. By combining the strengths of Kaitai Struct schema knowledge with targeted mutation strategies, it can significantly enhance the effectiveness of fuzzing. The structure aware fuzzer is mainly designed with three main components: Input Generator, Mutator and Validator.

5.5.1 Input Generator

It generates initial seed inputs or creates new inputs based on the structure described in the file format specifications. It utilizes information from the file format specifications to generate valid inputs that conform to the defined structure. It can incorporate randomization, grammar-based generation, or other strategies to create diverse inputs.

5.5.2 Mutator

The mutation component of a generation-based structure-aware fuzzer plays a critical role in diversifying the inputs used for testing. Its primary function is to modify existing inputs in a controlled manner, creating variations that explore different parts of the input space while ensuring the mutated inputs remain valid according to the file format specifications. This component typically employs various mutation techniques,

such as bit flips, byte rearrangements, value changes, and insertion/deletion of data, to achieve this goal.

5.5.3 Validator

The validation component of a structure-aware fuzzer is crucial for ensuring the quality and correctness of the generated inputs before they are used for testing. Its primary function is to validate the inputs to ensure they conform to the file format specifications, are structurally correct, and adhere to the defined data types, sizes, and constraints. Additionally, the validation component tests the inputs against the parser or target application to detect parsing errors, crashes, or unexpected behavior, ensuring that only valid and meaningful inputs are used for testing purposes.

5.6 Human Interface Design

The LLVMFuzzerTestOneInput function, is the entry point for libFuzzer. It takes two arguments: a pointer to the input buffer (data) and the size of the input buffer (size). This function is responsible for processing the input provided by libFuzzer. LLVMFuzzerTestOneInput receives inputs that are generated by libFuzzer itself or provided by the user through command-line options or input files specified when running the fuzzer. These inputs, represented as buffers of bytes, are passed to LLVMFuzzerTestOneInput for testing purposes. While user input may influence the configuration or parameters of the fuzzing session (such as specifying input seed files or setting fuzzing duration), the interaction occurs outside the scope of LLVMFuzzerTestOneInput.

Chapter 6

Implementation and Testing

6.1 Implementation

We propose the development of a Python-based generation fuzzer that utilises the power of Kaitai Struct (KS) for describing binary file formats.

Initially, we recognized the significance of understanding the Kaitai Struct and its YAML-based language for describing binary data structures. Through this understanding, we will generate KSY files that represent various binary formats, serving as the basis for input generation. By analyzing the structure definitions within these KSY files, we aim to grasp the underlying fields, thus gaining insight into the organization and encoding of data within the binary format.

Furthermore, we focus on the development of a dedicated library aimed at integrating libFuzzer with Kaitai Struct for efficient fuzz testing. This library will serve as a means for communication between the Kaitai Struct and libFuzzer. Through the implementation of essential functions and utilities, we will ensure compatibility and proper data exchange, while also providing clear documentation and examples to facilitate easy integration and usage.

6.1.1 KSY File formats

We've compiled a database consisting of collections of KSY files for various binary file formats such as executables, multimedia files, and database files. We curated approximately 215 KSY files, and we are currently expanding the database to include

more.

6.1.2 Python-based Generation Fuzzer

The Python-based generation fuzzer utilizes the descriptive power of Kaitai Struct. The fuzzer logic automatically generates diverse test inputs using techniques like input generation algorithms. The parsed KSY files ensure comprehensive coverage of data structures and edge cases in input generation. The success of the parser depends on whether it achieves at least 90 percent coverage when parsed using the open-source parsers for that file format.

6.1.3 Source Code

The github link for the project is https://github.com/parvathymohan932/Binary-fuzzer

6.2 Testing

To ensure the quality and reliability of the project, we conducted comprehensive testing throughout the development process. The test report validates the functionality and accuracy of the generation-based fuzzer implemented in Python. Across various test cases, including metadata extraction, data packing, random data generation, sequence and type handling, and output file writing, the fuzzer demonstrates consistent and expected behavior.

- Metadata extraction successfully retrieves endianness, file extension, and ID values as specified in the input YAML data.
- Data packing ensures that binary data is correctly formatted according to the specified endianness and data type.
- Random data generation produces binary data within the expected ranges and formats for each data type.
- The fuzzer effectively handles sequences and types defined in the YAML file, generating binary data that conforms to the specified structure.

- Output file writing generates files with the appropriate naming convention and containing the expected binary data.

The test cases generated by the fuzzer are validated using parsers obtained by compiling the corresponding Kaitai Struct file. Additionally, open-source parsers available for specific file formats are utilized to ensure the correctness of the generated test cases. This approach guarantees thorough verification of the generated binary data against the defined data structure and format specifications.

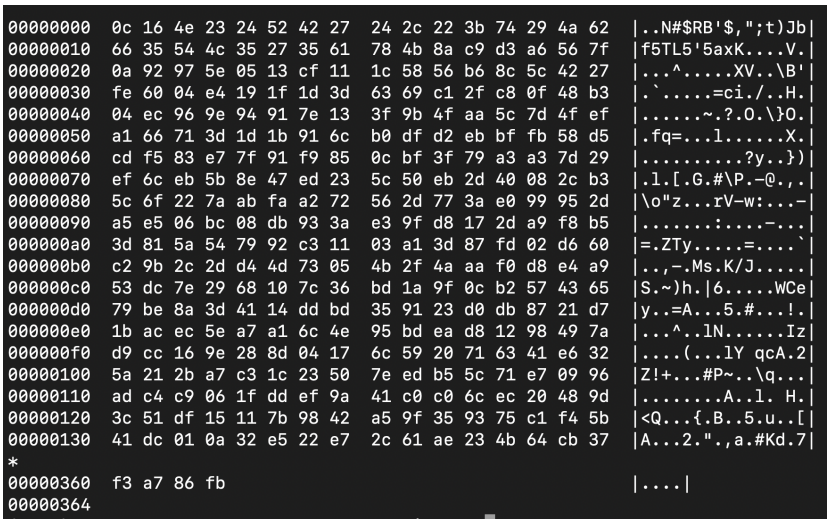


Figure 6.1: Binary representation of a test case displayed through hexdump, * represents rest of the bytes.

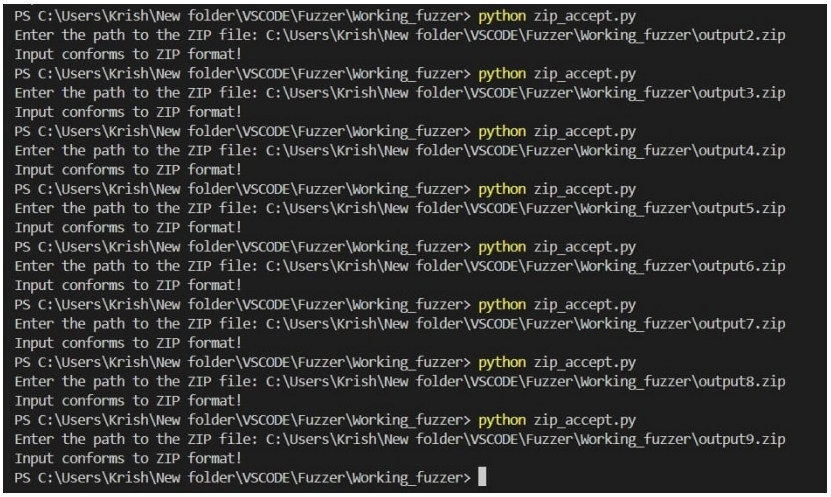


Figure 6.2: Kaitai Struct Parser utilised to check whether generated test case conforms to format

Chapter 7

Results and Discussion

In our project, one of our major accomplishments has been the successful development of a Python-based generation fuzzer. This fuzzer is designed to cover approximately 90% of Kaitai Struct features, making it a valuable tool for automating the generation of test cases tailored to various binary file formats. To ensure its effectiveness, we conducted thorough evaluations using open-source parsers specific to these formats.

Additionally, we've assembled a comprehensive database containing collections of Kaitai Struct files. These files serve as foundational blueprints for our fuzzer, providing essential reference points for generating test cases accurately.

In parallel with our fuzzer development, we've been actively engaged in learning about recent studies and research in structure-aware fuzzing. This exploration has deepened our understanding of important fuzzing tools such as LLVM and libFuzzer. While considering various options, including AFLSmart Fuzzer, we ultimately selected libFuzzer due to its alignment with our project's requirements and objectives.

Furthermore, an idea was also proposed to develop a library for integrating Kaitai Struct with libFuzzer. This integration could significantly enhance our fuzzing process by leveraging Kaitai Struct's proficiency in understanding file formats alongside libFuzzer's expertise in testing them. By combining these capabilities, we aim to enhance the security and reliability of software utilizing Kaitai Struct for handling binary files.

Chapter 8

Conclusions and Future Scope

8.1 Conclusion

In conclusion, our project introduces a novel approach to fuzz testing by utilising the descriptive power of Kaitai Struct (KSY) files for structure-aware fuzzing. Through the development of a Python-based generation fuzzer specifically designed for binary inputs, we've created a tool that has proven highly adaptable and effective in generating test cases for programs handling structured binary data. Also, the proposed integration of Kaitai Struct and libFuzzer addresses the complexities of binary formats, improving automated testing and paving the way for further developments in the field. By focusing on generating tailored test cases and ensuring comprehensive coverage, our framework aims to provide security researchers and developers with a robust tool for identifying vulnerabilities in software applications. This advancement has the potential to significantly improve fuzz testing techniques and enhance the detection of vulnerabilities, ultimately contributing to the security and reliability of software systems.

8.2 Future Scope

While our project has made significant strides in enhancing test case generation and efficiency, there are several avenues for future exploration and improvement. Here are some potential areas of future scope:

- **Integration with libFuzzer**: We plan to implement the proposed approach for integrating Kaitai Struct with libFuzzer. While we have outlined the method, time constraints prevented us from completing the implementation during the current project phase. However, integrating these two powerful tools holds significant promise for enhancing fuzz testing capabilities, especially for programs handling structured binary inputs.

- **Algorithm Refinement and Optimization**: Continuously refine and optimize algorithms governing test case generation and mutation processes. Enhance the effectiveness and efficiency of the fuzzing techniques by fine-tuning these algorithms. Explore advanced strategies, including machine learning, to intelligently guide fuzzing towards high-risk areas within the target program.

- **Expansion of Binary Format Support**: Extend support to encompass a broader array of binary formats and protocols, catering to the diverse needs of software systems. Enhance scalability and performance to effectively handle larger and more intricate file formats.

References

[1] N. Anwar and S. Kar, ”Review Paper on Various Software Testing Techniques,” *Global Journal of Computer Science and Technology*, vol. 19, no. C2, pp. 43-49, May 2019. Available: https://computerresearch.org/index.php/computer/article/view/1873

[2] S. Bhunia and M. Tehranipoor, ”Chapter 13 - Security and Trust Assessment, and Design for Security,” in *Hardware Security*, Swarup Bhunia and Mark Tehranipoor (eds.), Morgan Kaufmann, 2019, pp. 347-372. doi: 10.1016/B978-0-12-812477-2.00018-6.

[3] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, ”Fuzzing: Breaking Things with Random Inputs,” in *The Fuzzing Book*, CISPA Helmholtz Center for Information Security, 2024. [Online]. Available: https://www.fuzzingbook.org/html/Fuzzer.html, Accessed: January 18, 2024.

[4] *Kaitai Struct*, [Online]. Available: https://kaitai.io, Accessed: Insert Date Here.

[5] C. Miller and Z. N. J. Peterson, ”Analysis of mutation and generation-based fuzzing,” *Independent Security Evaluators, Tech. Rep*, vol. 4, 2007.

[6] *PNG format specification in Kaitai Struct*, [Online]. Available: http://formats.kaitai.io/png/, Accessed: April 12, 2024.

[7] ”Binary file,” *Wikipedia, The Free Encyclopedia*, [Online]. Available: https://en.wikipedia.org/wiki/Binary_file, Accessed: April 12, 2024.

[8] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury, ”Smart greybox fuzzing,” *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1980-1997, 2019.

[9] *LibFuzzer Documentation*, [Online]. Available: https://llvm.org/docs/LibFuzzer.html.

[10] *Structure-Aware Fuzzing*, Google, [Online]. Available: https://github.com/google/fuzzing/blob/master/docs/structure-aware-fuzzing.md.

[11] J. Li, B. Zhao, and C. Zhang, “Fuzzing: a survey,” *Cybersecur*, vol. 1, p. 6, Jun. 2018. DOI: 10.1186/s42400-018-0002-y

[12] P. Oehlert, “Violating assumptions with fuzzing,” *IEEE Security & Privacy*, vol. 3, no. 2, pp. 58–62, Mar. 2005.

[13] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of UNIX utilities,” *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990. DOI: 10.1145/96267.96279

[14] R. Dutra, R. Gopinath, and A. Zeller, “FormatFuzzer: Effective Fuzzing of Binary File Formats,” *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 2, Dec. 2023, Art. no. 53, pp. 1–29. DOI: 10.1145/3628157

[15] Wikipedia contributors, ”Protocol Buffers,” Wikipedia, The Free Encyclopedia, available at: https://en.wikipedia.org/wiki/Protocol_Buffers, accessed on [insert date].

[16] Google, ”libprotobuf-mutator,” GitHub. [Online]. Available: https://github.com/google/libprotobuf-mutator.

[17] S. Madnick, “Why Data Breaches Spiked in 2023,” *Harvard Business Review*, Feb. 2024. [Online]. Available: https://hbr.org/2024/02/why-data-breaches-spiked-in-2023.

[18] Imperva, ”Fuzzing (Fuzz Testing),” Imperva, [Online]. Available: https://www.imperva.com/learn/application-security/fuzzing-fuzz-testing/.

[19] R. Dutra, R. Gopinath, and A. Zeller, "FormatFuzzer: Effective Fuzzing of Binary File Formats," *arXiv:2109.11277 [cs.SE]*, 2023. DOI: 10.48550/arXiv.2109.11277. Available online: https://doi.org/10.48550/arXiv.2109.11277.

[20] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-Aware Greybox Fuzzing," *arXiv:1812.01197 [cs.CR]*, 2019. DOI: 10.48550/arXiv.1812.01197. Available online: https://doi.org/10.48550/arXiv.1812.01197.

[21] Z. Chen, Y. Lu, K. Zhu, L. Yu, and J. Zhao, "Fast Format-Aware Fuzzing for Structured Input Applications," *Applied Sciences*, vol. 12, no. 18, article 9350, 2022. DOI: 10.3390/app12189350. Available online: https://www.mdpi.com/2076-3417/12/18/9350.

[22] M. Zalewski, "American Fuzzy Lop (AFL)," GitHub repository, https://github.com/google/AFL.

[23] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury, "Smart Greybox Fuzzing," *arXiv:1811.09447 [cs.CR]*, 2018. DOI: 10.48550/arXiv.1811.09447. Available online: https://doi.org/10.48550/arXiv.1811.09447.

[24] Kathleen Abols, *Fuzzing Self-Described Structures*, Ph.D. thesis, Queen's University (Canada), 2023. ProQuest Dissertations Publishing, 30616948. Available online: https://qspace.library.queensu.ca/server/api/core/bitstreams/ddf8da12-9d1c-48b0-9add-a31b8fde26b3/content.

[25] Chen, Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. "A systematic review of fuzzing techniques." *Computers & Security* 75 (2018): 118-137.

[26] Underwood, William. "Grammar-based specification and parsing of binary file formats." *International Journal of Digital Curation* 7, no. 1 (2012): 95-106.

[27] Kaitai Struct Documentation. "General Serialization Procedure." Accessed April 24, 2024. https://doc.kaitai.io/serialization.html#_general_serialization_procedure.