

# Systematic Identification of Vulnerabilities in C and C++ Source Code through Fuzzing

**Bachelor Thesis**

Department of Computer Science  
OST – University of Applied Sciences  
Campus Rapperswil-Jona

Semester: Spring Term 2024

**Author:** Miles Strässle  
**Advisor:** Nikolaus Heners  
**External Co-Examiner:** Thomas Sutter  
**Internal Co-Examiner:** Prof. Dr. Olaf Zimmermann

# Contents

<b>1</b>	<b>Abstract</b>	<b>4</b>
1.1	Introduction . . . . .	4
1.2	Approach . . . . .	4
1.3	Conclusion . . . . .	4
<b>2</b>	<b>Management Summary</b>	<b>5</b>
2.1	Project Overview . . . . .	5
2.2	Scope and Implications for Cybersecurity . . . . .	5
2.3	Methodology . . . . .	5
2.4	Approach and Technologies . . . . .	5
2.5	Results and Future Outlook . . . . .	5
<b>3</b>	<b>Disclaimer</b>	<b>7</b>
<b>4</b>	<b>Introduction</b>	<b>8</b>
4.1	Background . . . . .	8
4.1.1	Types of Projects Suitable for Fuzzing . . . . .	8
4.1.2	Target Audience . . . . .	8
4.1.3	Why Fuzzing? . . . . .	9
4.1.4	Fuzzing in More Depth . . . . .	10
4.1.5	AFL++ . . . . .	11
4.1.6	Other Effective Fuzzers . . . . .	12
4.2	Problem Statement . . . . .	12
4.3	Objectives . . . . .	13
4.4	Scope . . . . .	13
4.5	Methodology . . . . .	14
4.6	Acknowledgment . . . . .	14
<b>5</b>	<b>Design and Architecture</b>	<b>15</b>
5.1	Conceptual Overview . . . . .	15
5.2	Subsystem and Component Design . . . . .	16
5.2.1	Component Design . . . . .	18
5.2.2	Integration Framework Diagram: . . . . .	18
5.2.3	Fuzzing Engine Process Diagram: . . . . .	18
5.3	Fuzzing Technology Overview . . . . .	19
5.3.1	AFLplusplus . . . . .	20
5.3.2	libFuzzer . . . . .	20
5.3.3	honggfuzz . . . . .	20
5.3.4	ClusterFuzz . . . . .	20
<b>6</b>	<b>Implementation and Analysis</b>	<b>22</b>
6.1	Hello AFL! . . . . .	22
6.2	libxml2 . . . . .	23
6.3	contour . . . . .	24
6.3.1	Harness Setup . . . . .	24
6.3.2	Threaded Fuzzing . . . . .	24
6.3.3	Results . . . . .	25
6.4	NASA HDTN (Tryout) . . . . .	26
6.5	libBLS . . . . .	27
6.5.1	Harness . . . . .	27

6.5.2	Start . . . . .	28
6.6	TCPDump . . . . .	28
6.6.1	AFL-Plot Analysis . . . . .	29
6.7	libjpeg-turbo . . . . .	31
6.8	libexpat . . . . .	31
6.9	pdfcrack . . . . .	31
6.9.1	GDB Analysis . . . . .	32
6.9.2	Detailed Analysis . . . . .	33
6.10	Libarchive . . . . .	36
6.10.1	Analysis of Buffer Overflow in readline Function . . . . .	36
6.10.2	Analysis of Segmentation Fault in edit_pathname Function . . . . .	37
6.11	ImageMagick . . . . .	38
6.12	Analysis and Discussion . . . . .	39
6.12.1	Metrics . . . . .	39
6.12.2	Termination conditions . . . . .	45
6.12.3	Limitations of AFL++ . . . . .	46
6.12.4	Sample Penetration Test Report for vt-parser . . . . .	47
<b>7</b>	<b>Conclusion</b> . . . . .	<b>48</b>
7.1	Achievements Compared to Initial Objectives . . . . .	48
7.2	Implications for Software Security . . . . .	48
7.3	Future Work . . . . .	49
7.4	Self-Reflection . . . . .	49
<b>8</b>	<b>Glossary</b> . . . . .	<b>50</b>
<b>9</b>	<b>References</b> . . . . .	<b>52</b>
9.1	Bibliography . . . . .	52
9.2	Figures . . . . .	54
<b>10</b>	<b>Appendices</b> . . . . .	<b>55</b>
10.1	Appendix Task Definition . . . . .	55
10.2	Appendix contour . . . . .	56
10.2.1	Threaded Fuzzing . . . . .	56
10.2.2	Crash Analysis . . . . .	57
10.2.3	Bash: Helper cmake install . . . . .	58
10.2.4	Bash: Helper container patch . . . . .	58
10.2.5	Bash: Helper seed minimizer . . . . .	58
10.3	Appendix libBLS . . . . .	58
10.3.1	libBLS Harness . . . . .	58
10.3.2	File Patch . . . . .	61
10.3.3	Build Process . . . . .	61
10.3.4	Threaded Run . . . . .	62
10.4	Appendix tcpdump . . . . .	63
10.4.1	Build . . . . .	63
10.5	Appendix libjpeg-turbo . . . . .	63
10.5.1	Build . . . . .	63
10.6	Appendix libexpat . . . . .	64
10.6.1	Build . . . . .	64
10.7	Appendix pdfcrack . . . . .	64
10.7.1	Build . . . . .	64
10.7.2	Bash: Generate Encrypted Samples . . . . .	64

10.7.3	Bash: Minimize and Analyse Crashes . . . . .	64
10.7.4	Python: Crash Cause Analysis . . . . .	65
10.7.5	Crash Details . . . . .	66
10.8	Appendix Libarchive . . . . .	67
10.8.1	Build . . . . .	67
10.8.2	Bash: Minimize and Analyze Script async . . . . .	67
10.8.3	Python: Analyse crash cause . . . . .	68
10.8.4	Grouping Analysis of Crashfiles . . . . .	69
10.8.5	ASAN Report readline . . . . .	73
10.9	Appendix ImageMagick . . . . .	74
10.9.1	Build . . . . .	74
10.9.2	Build With known Vuln-Files as Input . . . . .	74
10.10	Appendix Analysis . . . . .	75
10.10.1	Python Analysis Script for Metrics . . . . .	75
10.10.2	Config for Python Analysis Script for Metrics . . . . .	82
10.10.3	Sample Penetration Test Report for vt-parser . . . . .	85

# 1 Abstract

## 1.1 Introduction

As software becomes more complex security issues in applications grow. This research focuses on the use of high-performance fuzzing techniques and also investigates performance parameters for fuzzing in different contexts. Fuzzing is a method of finding software vulnerabilities by injecting random data into programs to reveal and fix potential security flaws. The goal is to use an advanced fuzzing framework to identify vulnerabilities in real-world open-source C and C++ software, thereby improving its robustness and security.

## 1.2 Approach

To find a suitable project for fuzzing, the search targeted software that accepts input from users or external sources, focusing on areas most likely to contain vulnerabilities. A variety of open-source C and C++ projects with significant user interaction components were selected. A fuzzing harness was then created to test these critical areas of the software, utilizing various inputs and seed values. Employing white-box fuzzing, full access to the source code allowed for more informed tests, simplifying bug identification and avoiding reverse engineering as in black-box fuzzing. Once the tests were executed, the resulting bugs and hangs were analyzed to understand their causes and potential security implications. Metrics such as the number of bugs found, the types of vulnerabilities, and the duration of tests were collected to assess the effectiveness of the fuzzing process.

## 1.3 Conclusion

This project used advanced fuzzing techniques to test real-world C and C++ open-source projects. The fuzzing framework successfully reproduced many known security vulnerabilities, proving its effectiveness and reliability. Although no new vulnerabilities were found, the high performance and efficiency of the setup make it suitable for ongoing security testing. The metrics collected - such as the number and types of bugs found and test durations - highlighted areas for improvement. The results demonstrate the robustness of the code in the context of penetration testing and security audits, underscoring the importance of continuous security testing and how fuzzing can enhance software security.

## 2 Management Summary

This project aims to systematically identify vulnerabilities in C and C++ source code using advanced fuzzing techniques. Given the increasing complexity of software and the rising prevalence of security issues, this research is both timely and crucial.

### 2.1 Project Overview

The project focuses on utilizing a sophisticated fuzzing framework to discover vulnerabilities in real-world open-source C and C++ software. By targeting software that interacts with users or external sources, the study emphasizes the importance of identifying and mitigating potential security flaws in these critical areas.

### 2.2 Scope and Implications for Cybersecurity

The scope of the project encompasses selecting suitable open-source projects, developing a fuzzing harness, and conducting comprehensive tests to uncover vulnerabilities. The implications for cybersecurity are significant, as identifying and addressing these vulnerabilities can lead to more robust and secure software, ultimately protecting users from potential threats.

### 2.3 Methodology

The methodology involves several key steps:

- **Project Selection:** Identify and select open-source C and C++ projects that accept user or external input, focusing on those most likely to contain vulnerabilities.
- **Fuzzing Harness Development:** Develop a fuzzing harness to test critical areas of the selected software. This includes configuring various inputs and seed values to thoroughly test the software.
- **White-Box Fuzzing:** Employ white-box fuzzing techniques, leveraging full access to the source code. This approach allows for more informed test designs and easier bug identification compared to black-box fuzzing, which requires reverse engineering.
- **Test Execution and Analysis:** Execute the fuzzing tests and analyze the resulting bugs and hangs. This involves understanding the causes of identified vulnerabilities and their potential security implications.
- **Metrics Collection:** Collect and analyze metrics such as the number of bugs found, types of vulnerabilities, and test durations to assess the effectiveness of the fuzzing process.

### 2.4 Approach and Technologies

The approach involves white-box fuzzing, which leverages full access to source code, allowing for informed test designs and easier bug identification. This method contrasts with black-box fuzzing, which requires reverse engineering. Various inputs and seed values are utilized to test the software's critical components. The technologies employed include advanced fuzzing frameworks capable of executing thorough and efficient tests.

### 2.5 Results and Future Outlook

The results demonstrate the effectiveness of the fuzzing framework in reproducing known vulnerabilities and assessing the robustness of the tested software. While no new vulnerabilities were discovered, the high performance of the fuzzing setup highlights its suitability for continuous

security testing. The collected metrics, such as the number and types of bugs found and test durations, provide insights for future improvements. This project underscores the importance of ongoing security testing and how fuzzing can significantly enhance software security.

### **3 Disclaimer**

In the preparation of this Bachelor thesis, I utilized ChatGPT-4 to improve the readability and language of the text, as well as to assist in the creation of the glossary. All code provided in this thesis, unless otherwise stated, is my original work. ChatGPT-4 was used exclusively as a tool for debugging assistance. I take full pride in the effort and work I have put into this project.



## 4 Introduction

*Fuzzing is the art of automatic bug finding, and it's role is to find software implementation faults, and identify them if possible.* – OWASP [1]

### 4.1 Background

As software systems become increasingly complex, the potential for security vulnerabilities grows correspondingly. One of the most effective techniques for identifying such vulnerabilities is fuzzing. Fuzzing involves providing random or semi-random data inputs to software applications to discover unexpected behavior, crashes, or security issues.

Fuzzing has gained significant traction in both academic and industrial settings due to its effectiveness in uncovering critical vulnerabilities. For example, the GNOME GLib project has integrated fuzzing into its components using OSS-Fuzz, a service designed to continuously fuzz open-source software. Details about this integration can be found in the GNOME GitHub repository [2] and the OSS-Fuzz blog post [3].

In the broader open-source community, Google's OSS-Fuzz project supports over 250 projects, providing continuous fuzzing services that have led to the discovery of numerous security vulnerabilities. Detailed statistics and success stories from OSS-Fuzz are available in Google's official blog posts and project documentation [4].

Conferences such as Black Hat and DEF CON frequently feature presentations and workshops dedicated to fuzzing, demonstrating its importance and widespread adoption in the cybersecurity community. Archives from these conferences provide valuable insights into the latest trends and real-world applications of fuzzing. You can explore these resources on the Black Hat Archives [5] and DEF CON Archives [6].

Fuzzing tools like AFL (American Fuzzy Lop) and LibFuzzer have also seen widespread adoption. The AFL GitHub repository [7] and the LibFuzzer GitHub repository [8] contain extensive documentation and community contributions that illustrate the practical applications and benefits of these tools.

Overall, the adoption of fuzzing is widespread and growing, reflecting its critical role in modern software security practices. This thesis will delve deeper into the systematic identification of vulnerabilities in C and C++ source code through fuzzing, exploring both the methodologies and the practical implications of this technique.

#### 4.1.1 Types of Projects Suitable for Fuzzing

Fuzzing is very useful for projects that need to handle various types of external inputs. Choosing the right projects for fuzzing is important for the success of fuzzing campaigns. For example, "Contour," a terminal emulator, is a complex C++ application that has to manage random user inputs.

Different codebases can vary a lot. Contour, for instance, is well-made and mostly uses standard functionality, which is usually less buggy than custom implementations. This makes it a good candidate for fuzzing. Even with its stable base, Contour has components like the vtparser that need to handle ANSI escape sequences, which can introduce subtle bugs due to specific usage patterns or integration issues. Fuzzing Contour helps uncover and fix these hidden bugs, leading to a more robust and reliable application.

#### 4.1.2 Target Audience

Software developers, who are responsible for creating and maintaining code, can greatly benefit from understanding and using fuzzing techniques. By incorporating fuzzing into their testing processes, developers can proactively identify and fix vulnerabilities, making their applications more secure and robust.

Penetration testers, or pentesters, are security experts who assess the security of systems by simulating attacks. For pentesters, fuzzing is a valuable tool for discovering vulnerabilities that could be exploited in real-world scenarios. This thesis provides insights into advanced fuzzing techniques and tools, helping pentesters enhance their methods and improve the accuracy and depth of their security assessments.

### 4.1.3 Why Fuzzing?

Traditional testing methods like Unit Testing and Static Analysis are useful but sometimes they can't find certain bugs and problems. This is where fuzzing becomes important.

Unit Tests check small parts of the code to make sure each part works correctly. However, they can miss unusual cases and how different parts of the code work together, which can cause unexpected problems or security issues. Humans, even though good at designing tests, can make mistakes and miss important scenarios.

A notable example of the effectiveness of fuzzing is the discovery of the Heartbleed bug in OpenSSL. This vulnerability, which allowed attackers to read sensitive data from the memory of affected systems, went undetected for over two years. It was discovered by two independent teams - one from Google and one from Codenomicon - using fuzzing techniques. By continuously providing invalid input, the fuzzers detected anomalous behaviour where more data than expected was returned, leading to the identification of the bug. This demonstrates the power of fuzzing in uncovering significant vulnerabilities that other testing methods might miss [9].

#### 1. Static Analysis

Static analysis techniques, like code reviews, data flow analysis, control flow analysis, dependency analysis, and code metrics, look at the code without running it. These methods can find possible problems early, such as syntax errors or potential security issues. Formal verification ensures the software meets certain specifications. However, static analysis can give false positives and negatives and might not find errors that only show up when the code is running.

#### 2. Dynamic Analysis

Dynamic analysis involves running the code and watching what happens. This includes unit testing, integration testing, regression testing, performance testing, memory leak detection, profiling, and fuzzing. Dynamic analysis is better at finding issues related to how the code runs, like performance problems and memory leaks. Fuzzing is especially good at finding hidden vulnerabilities by automatically generating and injecting random or incorrect data into the software.

#### 3. Comparison of Techniques

Each testing technique has its focus, strengths, and limitations. Here's a comparison:

Table 1: Comparison of Techniques

Technique	Focus Area	Strengths	Limitations	Best Use
Unit Testing	Code units	Targeted, Automatable	Scenario-bound	Specific function precision
Static Analysis	Codebase	Early flaw detection	False positives /negatives	Compliance, Standards en- forcement
Fuzzing	Runtime behavior	Uncovers hidden issues	Resource-heavy	Security, Vulnerability discovery

Fuzzing complements traditional testing methods by addressing their weaknesses. While unit testing and static analysis are essential for checking specific functions and ensuring code quality, fuzzing is great at finding vulnerabilities that other methods might miss. Using fuzzing helps developers improve the security and reliability of their software, making it a crucial part of modern software development.

#### 4. AFL Fuzzing Impact on Open Source Projects

Michal Zalewski’s work on AFL (American Fuzzy Lop) has been very influential. AFL has found many vulnerabilities in open source projects that other testing methods missed. According to Zalewski [10], AFL’s automated approach and powerful mutation strategies have greatly improved the security of many open-source projects.

#### 5. Fuzzing in the Software Development Life Cycle (SDLC)

Sudhakar, Arora, and Moghnie [11] have documented the importance of integrating fuzzing into the Software Development Life Cycle (SDLC). They emphasize that using fuzzing early in development helps find and fix vulnerabilities before the software is released.

#### 6. Addressing Software Memory Safety Issues

The National Security Agency (NSA) has highlighted the importance of dealing with software memory safety issues, which are often found through fuzzing. Their 2023 guidance [12] stresses the need to use fuzzing to find memory-related vulnerabilities and improve software security.

#### 7. Identifying Dangerous Software Weaknesses

MITRE’s 2023 CWE Top 25 list shows the most dangerous software weaknesses, many of which can be found using fuzzing techniques. This list [13] is an important reference for developers and security professionals working to fix common vulnerabilities.

#### 8. Programming Language Popularity and Fuzzing

The TIOBE Programming Community Index shows the popularity of different programming languages, which can affect the focus of fuzzing research. Knowing which languages are most used helps direct fuzzing efforts where they are most needed [14].

#### 9. The Fuzzing Book

“The Fuzzing Book” provides comprehensive information on fuzzing techniques and their applications. This book is a valuable resource for anyone interested in learning about fuzzing in depth [15].

### 4.1.4 Fuzzing in More Depth

#### 1. Orchestration

Orchestration in fuzzing refers to the management and coordination of the fuzzing process. This includes setting up the fuzzer, managing input seeds, monitoring the progress of fuzzing runs, and analyzing the results. Effective orchestration tools automate these workflows, making the fuzzing process more efficient. Examples of such tools include the orchestration capabilities in AFL++, which support automated fuzzing workflows and help manage the complexity of large-scale fuzzing campaigns.

#### 2. Instrumentation

Instrumentation is the process of modifying a program to collect information during its execution. This is crucial for source code fuzzing as it provides insights into the program’s behavior under test conditions. Instrumentation can be static, where the code is modified

before execution, or dynamic, where modifications are made during execution. Tools like AFL++ offer various options for efficient instrumentation, enabling detailed tracking of execution paths and helping to uncover vulnerabilities more effectively.

### 3. Sanitizers

Sanitizers are tools that detect various types of bugs and vulnerabilities in software during execution. They are integrated into the build process to identify issues such as memory errors, undefined behavior, and race conditions. Examples of commonly used sanitizers include:

- **ASAN (Address Sanitizer)**: Detects memory errors such as buffer overflows and use-after-free errors.
- **UBSAN (Undefined Behavior Sanitizer)**: Identifies undefined behavior in code, such as integer overflows.
- **TSAN (Thread Sanitizer)**: Finds race conditions in multi-threaded code.

### 4. Fuzzing Harness

A fuzzing harness is a piece of code that interfaces between the fuzzer and the software being tested. It directs the fuzz-generated inputs to the appropriate parts of the program, ensuring that the tests are meaningful and effective. A well-designed fuzzing harness can significantly enhance the effectiveness of a fuzzing campaign by accurately targeting specific areas of the code and providing relevant feedback on discovered vulnerabilities. This increases the overall efficiency of the fuzzing process and helps uncover more subtle bugs.

By integrating orchestration, instrumentation, sanitizers, and a well-designed fuzzing harness, fuzzing campaigns can be more targeted and effective. These components work together to automate the fuzzing process, provide detailed insights into program behavior, and identify a wide range of vulnerabilities, making them essential tools in the software development lifecycle.

#### 4.1.5 AFL++

##### 1. What is AFL++?

AFL++ is an enhanced version of the original American Fuzzy Lop (AFL) fuzzer. It incorporates additional features and improvements that make it more efficient and versatile for modern fuzzing needs. AFL++ offers a range of advanced options and optimizations that improve its performance and usability, making it a powerful tool for security testing and vulnerability discovery. [16].

##### 2. AFL++ in Code Coverage

AFL++ builds upon the original AFL by introducing several enhancements, particularly in terms of code coverage. The improved instrumentation options in AFL++ allow for more detailed tracking of execution paths within the target software. This results in higher code coverage and the discovery of more potential vulnerabilities. AFL++ includes features such as context-sensitive coverage, persistent mode, and various mutation strategies that significantly increase the effectiveness of fuzzing campaigns. These enhancements help in identifying deeper and more complex bugs that traditional methods might miss.

##### 3. Key Features

AFL++ comes with several key features that enhance its functionality:

- **Custom Mutators:** Allows for tailored input mutation using strategies like splice, trim, and havoc-mutate, which help generate diverse test cases and improve bug discovery.
- **Instrumentation Options:** Provides comprehensive code coverage through various instrumentation techniques, enabling more detailed tracking of execution paths.
- **Performance Improvements:** Includes optimizations for faster fuzzing cycles, making the fuzzing process more efficient and allowing for quicker identification of issues.

#### 4. Reasons for AFL++

There are several reasons why AFL++ is a preferred choice for fuzzing:

- **Compatibility:** Supports C++ and can run in Docker environments, making it versatile and easy to integrate into different development and testing workflows.
- **Features:** Offers a wide range of advanced features that enhance fuzzing effectiveness, such as improved code coverage and custom mutators.
- **Community and Support:** AFL++ is actively developed and supported by a strong community, ensuring that it stays up-to-date with the latest advancements in fuzzing techniques and tools.

By using AFL++, developers and security professionals can leverage its advanced capabilities to conduct more thorough and effective fuzzing campaigns, ultimately improving the security and reliability of their software.

#### 4.1.6 Other Effective Fuzzers

Several fuzzers can potentially perform better for specific targets depending on the use case. A comparative analysis of these fuzzing tools reveals distinct advantages and limitations across various approaches. The tools analyzed include AFL, AFL++, libFuzzer, Honggfuzz, Syzkaller, and ClusterFuzz:

- **AFL++:** Offers extensive customization options, superior performance in diverse environments, and enhanced code coverage techniques [16].
- **libFuzzer:** Integrated with LLVM, it excels in targeted, function-level fuzzing, making it ideal for library testing. Its integration with LLVM allows for efficient instrumentation [17].
- **Honggfuzz:** Known for its performance and ease of integration, especially in continuous integration pipelines. It provides robust fuzzing capabilities with minimal setup [18].
- **Syzkaller:** Primarily used for kernel fuzzing and has found numerous vulnerabilities in Linux kernel code [19].
- **ClusterFuzz:** Provides an extensive setup guide and is widely used for large-scale fuzzing efforts, such as in Google’s continuous fuzzing service for open source projects [20].

## 4.2 Problem Statement

Established programming languages like C++ are still prone to subtle weaknesses and bugs, such as buffer overflows, integer overflows, and race conditions. These vulnerabilities lead to numerous security issues in production software. MITRE’s top 10 software vulnerabilities for 2023 include “Use After Free,” “Heap-based Buffer Overflow,” and “Out-of-bounds Write,” which are common in non-memory-safe languages like C++ [13]. The NSA has explicitly advised against using C++

due to these vulnerabilities [12]. Despite these concerns, C++ remains one of the most popular programming languages according to the TIOBE Programming Community Index [14]. Source-based fuzzing is a proven method for detecting such security vulnerabilities in C++ code [16]. This thesis aims to establish a fuzzing environment using AFL++ and ClusterFuzz [20]. The project will focus on learning and implementing source-based fuzzing techniques, selecting suitable target programs, and comparing different fuzzing parameters to evaluate performance. The goal is to enhance the security and robustness of C++ software by uncovering and addressing hidden vulnerabilities through comprehensive fuzzing campaigns.

### 4.3 Objectives

The primary goal of this thesis is to create a performant fuzzing environment for C++ projects. Specific objectives include:

- Understanding and applying source-based fuzzing techniques with AFL++ [16].
- Setting up a local fuzzing environment using ClusterFuzz [20].
- Selecting and targeting suitable software projects for fuzzing, such as terminal applications.
- Comparing various fuzzing parameters in terms of performance, including code coverage, cycles, duration, and crash conditions.
- Documenting and analyzing the results of the fuzzing campaigns.
- Optionally engaging in the responsible disclosure process for any discovered vulnerabilities.
- Providing comprehensive documentation of the fuzzing environment setup for future use by other students.

The initial task description can be seen in Appendix Taskdefinition

### 4.4 Scope

This thesis focuses on source code fuzzing, specifically White Box fuzzing, which leverages the availability of source code to perform more effective and targeted vulnerability detection [15]. Unlike Black Box fuzzing, which treats the software as a closed system, White Box fuzzing allows for instrumentation and in-depth analysis, making it a powerful tool for identifying complex security issues.

The scope is defined as follows:

#### **Programming Languages:**

- The primary focus is on C++ and C due to its widespread use in system-level programming and the prevalence of memory safety issues in the language [12]. While the techniques discussed can be applicable to other languages, this thesis will not cover C# or other languages in detail.

#### **Compilers:**

- The thesis will utilize modern C++ compilers compatible with AFL++, such as ‘clang’ and ‘clang++’, which support the necessary instrumentation and sanitization features required for effective fuzzing.

#### **Operating Systems:**

- The research and implementations will primarily be conducted on Unix-like operating systems, such as Linux, due to the compatibility and availability of fuzzing tools and the ease of integration with continuous integration (CI) pipelines.

## Tools and Techniques:

- The focus will be on White Box fuzzing techniques using AFL++ and its advanced features [16]. Additionally, the use of sanitizers such as AddressSanitizer (ASAN), UndefinedBehaviorSanitizer (UBSAN), and ThreadSanitizer (TSAN) will be explored to enhance the detection of memory and concurrency issues [11].

## 4.5 Methodology

This thesis employs a systematic approach to setting up and evaluating a high-performance fuzzing environment for C++ projects. The methodology involves the following steps:

1. **Literature Review:** Conduct a thorough review of existing literature on fuzzing techniques, tools, and methodologies. Key resources include “The Fuzzing Book,” AFL++ documentation, and academic papers on fuzzing methodologies.
2. **Environment Setup:** Set up a fuzzing environment using AFL++ and ClusterFuzz. This includes installing and configuring the necessary tools, such as compilers (clang/clang++) and sanitizers (ASAN, UBSAN, TSAN).
3. **Target Selection:** Identify and select suitable C++ projects for fuzzing. Criteria for selection include the project’s relevance, complexity, and potential impact of discovered vulnerabilities.
4. **Fuzzing Execution:** Conduct fuzzing campaigns on selected targets using AFL++. Explore various fuzzing parameters to optimize performance, including input seeds, mutation strategies, and instrumentation techniques.
5. **Data Collection and Analysis:** Collect and analyze data on fuzzing performance, such as code coverage, number of cycles, duration, and crash conditions. Use tools like GDB for post-mortem analysis of crashes to understand and document vulnerabilities.
6. **Documentation:** Document the setup process, fuzzing campaigns, and findings. Provide detailed reports on the vulnerabilities discovered and their potential impact.
7. **Responsible Disclosure:** If significant vulnerabilities are found, follow a responsible disclosure process to inform the affected project maintainers and collaborate on fixing the issues.

## 4.6 Acknowledgment

I want to thank everyone who helped make this thesis possible. Special thanks to my advisor, Nikolaus Heners, for his invaluable guidance and support. And a heartfelt thanks to my family and friends for their unwavering encouragement throughout this journey.

## 5 Design and Architecture

In this chapter, we examine the architecture and design of our system. Good fuzzing projects are characterized by certain features, including:

- **Clearly Defined Interfaces and Data Formats:** Projects that use clear and well-documented interfaces and standardized data formats are easier to fuzz.
- **Complexity and Size:** Larger and more complex projects offer more attack surface and potential vulnerabilities. However, the complexity should not be so high that meaningful test coverage becomes impossible.
- **Presence of Security-Critical Components:** Projects that contain security-critical functions, such as authentication systems, cryptographic implementations, or file parsing, are particularly suitable for fuzzing.
- **Maturity and Stability of the Code:** Projects that are in a stable development stage are better suited for fuzzing because they have fewer frequent changes and thus fewer false positives.

Fuzzing is useful in various stages of the Secure Software Lifecycle:

- **Design and Architecture:** Even in the design phase, fuzzing can be considered when defining interfaces and setting security requirements.
- **Development Phase:** During implementation, fuzzing helps identify vulnerabilities in the code early. This is the best time to integrate fuzzing tests, as issues can be fixed directly.
- **Testing Phase:** In the testing phase, fuzzing is used to check the robustness of the code and ensure that no vulnerabilities have been overlooked.
- **Deployment and Maintenance:** Even after the software is deployed, continuous fuzzing can help discover newly introduced vulnerabilities, especially when new features are added or existing ones are modified.

We use Docker for our fuzzing environment, allowing a quick setup and reproducibility. This chapter lays out the rationale behind our design decisions, emphasizing the integration of different technologies to optimize fuzz testing efficiency.

We chose AFL++ over other fuzzing frameworks like LibFuzzer and Honggfuzz due to its advanced features and robust community support. The use of Docker was motivated by the need for a portable and scalable environment that can be easily set up and duplicated across multiple testing scenarios.

### 5.1 Conceptual Overview

This diagram 1 represents the high-level architecture of our system. The Docker container encapsulates AFL++, the input harness, and the test cases database, illustrating how components interact within our fuzzing environment.



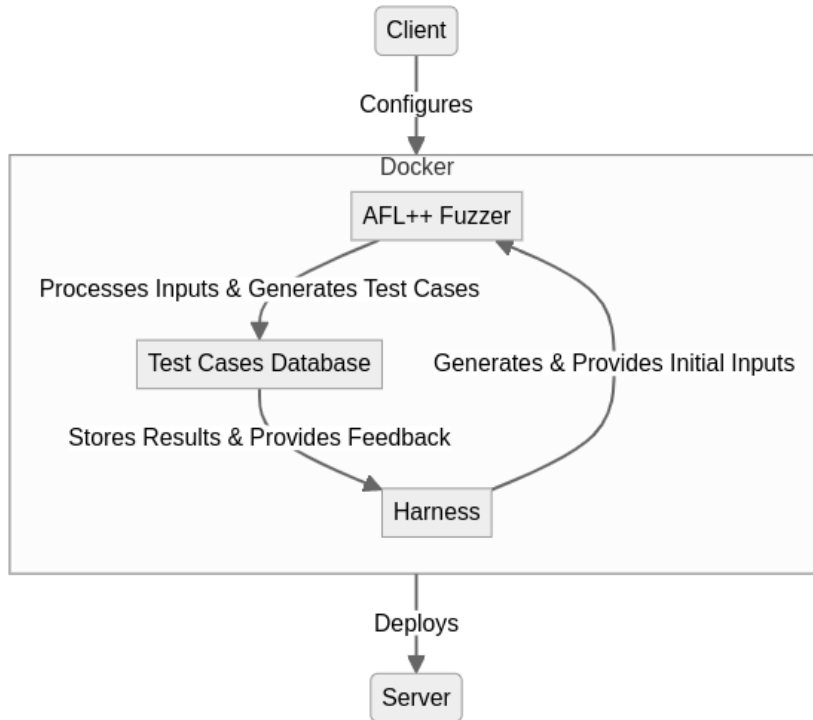


Figure 1: Design and Architecture Overview

Within the Docker container, the AFL++ fuzzer serves as the core component responsible for generating inputs that could potentially uncover vulnerabilities in the target application. The harness is designed to generate initial inputs for the fuzzer and ensure that these inputs are correctly formatted.

The test cases database plays a crucial role in storing all the generated test cases along with their results. This database preserves information for further analysis and also provides feedback to the harness. This feedback loop allows the harness to refine the inputs based on the outcomes of previous test cases, thereby enhancing the efficiency and effectiveness of the fuzzing process.

## 5.2 Subsystem and Component Design

Our system is divided into several subsystems: the Docker setup subsystem, the AFL++ configuration subsystem, and the test management subsystem. Each subsystem is responsible for a specific aspect of the fuzz testing process, ensuring modularity and ease of maintenance.

### 1. Docker Setup Subsystem

- **Responsibilities:**

- Ensures consistent and isolated environments for fuzz testing.
- Manages the creation, configuration, and deployment of Docker containers.
- Facilitates the reproducibility of fuzzing setups across different machines.

- **Key Components:**

- **Docker Image:** Base images including all necessary dependencies for AFL++ and other tools.
- **Dockerfiles:** Scripts for building Docker images.
- **Docker Compose:** Tool for defining and running multi-container Docker applications.

## 2. AFL++ Configuration Subsystem

- **Responsibilities:**

- Fine-tunes AFL++ settings for specific testing scenarios.
- Manages the input corpus and output directories for test cases and results.
- Integrates with other tools and scripts for enhanced fuzzing capabilities.

- **Key Components:**

- **AFL++ Settings:** Configuration parameters such as memory limits, timeout settings, and dictionary files.

Example:

```
afl-fuzz -i /fuzzing/input -o /fuzzing/output -m none -- ./test_program
```

- **Input Corpus:** Collection of initial input files for the fuzzer.

- **Output Directories:** Locations where AFL++ stores generated test cases and results.

## 3. Test Management Subsystem

- **Responsibilities:**

- Stores and manages the generated test cases.
- Analyzes results and provides feedback for improving fuzzing strategies.
- Generates reports summarizing the findings and potential vulnerabilities.

- **Key Components:**

- **Test Case Database:** Central repository for all test cases generated during fuzzing.
- **Result Analysis Tools:** Scripts or tools for analyzing crashes and logs.
- **Reporting Mechanisms:** Tools for generating and sharing reports on fuzzing results.

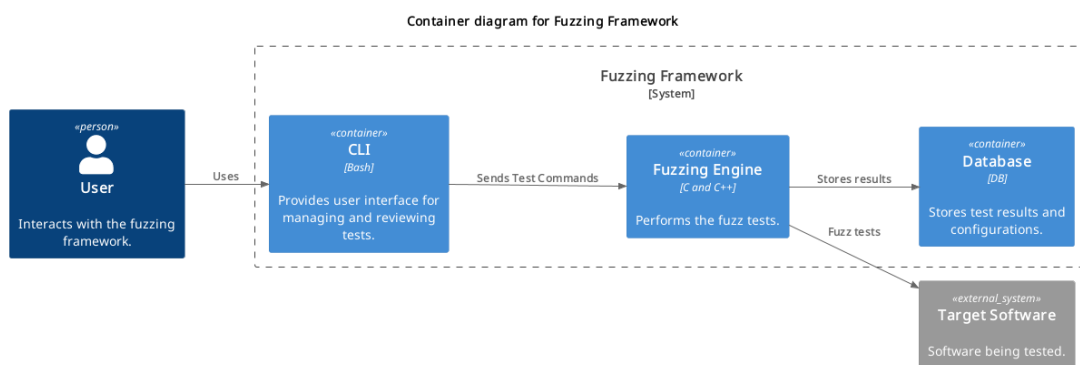


Figure 2: Container Diagram for Fuzzing Framework

Diagram 2 shows the overall flow of the fuzzing framework.

### 5.2.1 Component Design

The key components include the Docker container, the AFL++ engine, and the input harness. Each component is designed to perform distinct functions that collectively contribute to the effectiveness of the fuzz testing process.

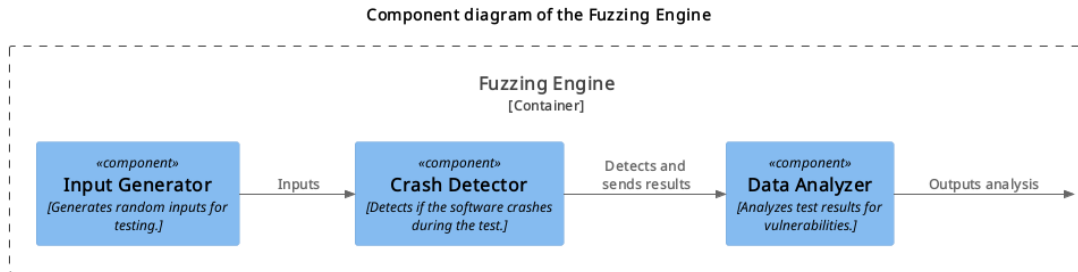


Figure 3: Component Diagram for Fuzzing Framework

### 5.2.2 Integration Framework Diagram:

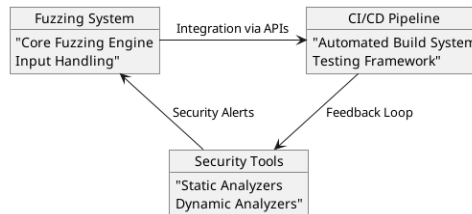


Figure 4: Integration Framework

A schematic 4 showing how the fuzzing system integrates with external systems, such as CI/CD pipelines and other security tools. It should highlight the interfaces and protocols used for integration.

### 5.2.3 Fuzzing Engine Process Diagram:

The diagram 5 shows the process of the Fuzzing Engine in bigger Detail.

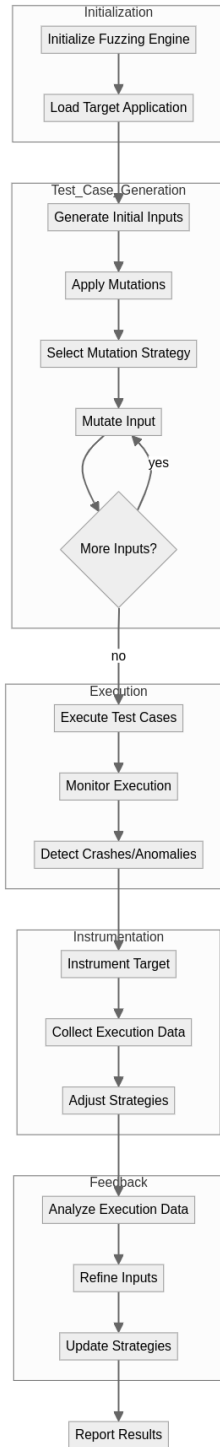


Figure 5: Fuzzing Engine Process Diagram

### 5.3 Fuzzing Technology Overview

The fuzzing technology stack employed in this project includes several key components designed to maximize the efficiency and effectiveness of vulnerability detection. Below are the main technologies and tools utilized:

### 5.3.1 AFLplusplus

**Overview:** AFL++ is an advanced fork of the original AFL (American Fuzzy Lop) fuzzer. It provides new features and improvements over the original, making it more effective for fuzzing modern applications [16].

**Features:**

- **Enhanced Fuzzing Techniques:** AFL++ includes several state-of-the-art fuzzing strategies that improve the efficiency of test case generation. [21]
- **Custom Mutators:** Allows for the integration of user-defined mutation strategies, enabling more targeted fuzzing.
- **Improved Performance:** Optimizations in the codebase result in faster fuzzing cycles and better utilization of system resources [22].

### 5.3.2 libFuzzer

**Overview:** libFuzzer is a library-based fuzzer that is part of the LLVM project. It is designed to be linked with the target program, allowing for in-process fuzzing which provides very fast execution speeds and detailed code coverage information [23].

**Features:**

- **In-Process Fuzzing:** Runs directly within the target program, enabling rapid test execution.
- **Code Coverage:** Utilizes LLVM’s coverage instrumentation to guide fuzzing [24].
- **Compatibility:** Works well with sanitizers like AddressSanitizer, ThreadSanitizer, and others, providing comprehensive bug detection [25].

### 5.3.3 honggfuzz

**Overview:** honggfuzz is a general-purpose fuzzer that supports multiple modes, including feedback-driven fuzzing and evolutionary fuzzing. It is known for its robustness and ability to handle large-scale fuzzing campaigns [26].

**Features:**

- **Feedback-Driven:** Uses runtime feedback to improve the quality of test cases.
- **Crash Analysis:** Provides detailed crash reports and stack traces to help with debugging.
- **Scalability:** Designed to scale across multiple machines, making it suitable for large projects.

### 5.3.4 ClusterFuzz

ClusterFuzz is an open-source scalable fuzzing infrastructure that automates the fuzzing process. It is designed to handle a large number of fuzzing jobs in parallel, distributing them across a cluster of machines [20].

**Steps to Try Out ClusterFuzz:**

1. **Setup Environment:** Configure the necessary environment for running ClusterFuzz, including dependencies like Docker and Google Cloud SDK.
2. **Integration:** Integrate the target binaries and configure fuzzing jobs using the ClusterFuzz interface.

3. **Run Fuzzing Campaigns:** Execute fuzzing campaigns and monitor the results through the ClusterFuzz dashboard.
4. **Analyze Results:** Collect and analyze crash reports generated by ClusterFuzz to identify and fix vulnerabilities.

#### 1. Challenges and Considerations in Implementing ClusterFuzz

Implementing ClusterFuzz in our infrastructure presented significant challenges, primarily due to its design for seamless integration with Google Cloud Storage. Here are the key difficulties we encountered and the decision that followed:

- (a) **Setup and Configuration** Setting up ClusterFuzz required configuring dependencies like Docker and Google Cloud SDK. Despite following the setup instructions meticulously, we faced numerous environment-specific issues that required extensive debugging and manual adjustments. These challenges made the initial setup highly time-consuming.
- (b) **Integration and Execution** Integrating target binaries with ClusterFuzz involved configuring fuzzing jobs and ensuring proper instrumentation using tools like libFuzzer and AFL++. However, the process proved complex, requiring significant trial and error without achieving stable operation.
- (c) **Decision to Use AFL++** After investing considerable time and effort without success, we decided to switch to AFL++. AFL++ provided a more straightforward setup and integration process, making it a more practical choice for our needs. Its compatibility with our existing infrastructure allowed us to quickly start fuzzing campaigns and focus on vulnerability detection.
- (d) **Conclusion** Our attempt to implement ClusterFuzz highlighted the complexities of using it outside its intended Google Cloud environment. Despite extensive efforts, we were unable to get it running reliably. The decision to switch to AFL++ allowed us to proceed efficiently with our fuzzing initiatives.

## 6 Implementation and Analysis

In this section, we explore the practical implementation of fuzzing techniques on various software projects to uncover vulnerabilities and enhance their robustness. By constructing and deploying fuzzing harnesses, we systematically test these programs under different scenarios to identify potential flaws and security issues.

### 6.1 Hello AFL!

To understand fuzzing a bit better I decided to write a program that crashes all the time. This was generated by ChatGPT4 when asking for a very simple fuzzing harness. There are calls to `std::abort()` which would trigger a **SIGABRT** Signal which aborts the process in which the executable is running.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <iostream>

using namespace std;

int main() {
    string str;

    cout << "enter input string: ";
    getline(cin, str);
    cout << str << endl << str[0] << endl;

    if (str[0] == 0 || str[str.length() - 1] == 0) {
        abort();
    } else {
        int count = 0;
        char prev_num = 'x';
        while (count != str.length() - 1) {
            char c = str[count];
            if (c >= 48 && c <= 57) {
                if (c == prev_num + 1) {
                    abort();
                }
                prev_num = c;
            }
            count++;
        }
    }

    return 0;
}
```

To build the harness, a very simple cmake setup was used to set the AFL compilers and enable the AddressSanitizer.

```
cmake_minimum_required(VERSION 3.10)
project(helloWorldHarness)

set(CMAKE_CXX_STANDARD 20)

set(CMAKE_C_COMPILER "/AFLplusplus/afl-clang-fast")
set(CMAKE_CXX_COMPILER "/AFLplusplus/afl-clang-fast++")

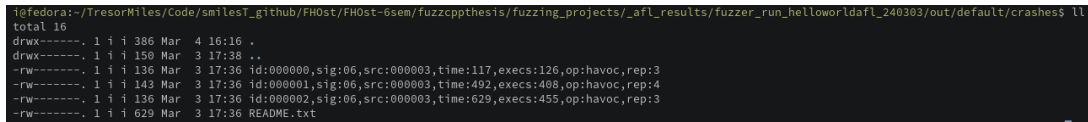
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fsanitize=address -g")
```

```
add_executable(${PROJECT_NAME} harness.cc)
```

The harness runs inside container to ensure encapsulation from the host system. The provided seeds are either random data or hex-strings. The Fuzzer will itself mutate the input and create new ones.

```
/AFLplusplus/afl-fuzz -i /src/fuzzing_projects/helloworldafl/seeds -o out -m none -d --  
↪ /src/fuzzing_projects/helloworldafl/build/helloWorldHarness
```

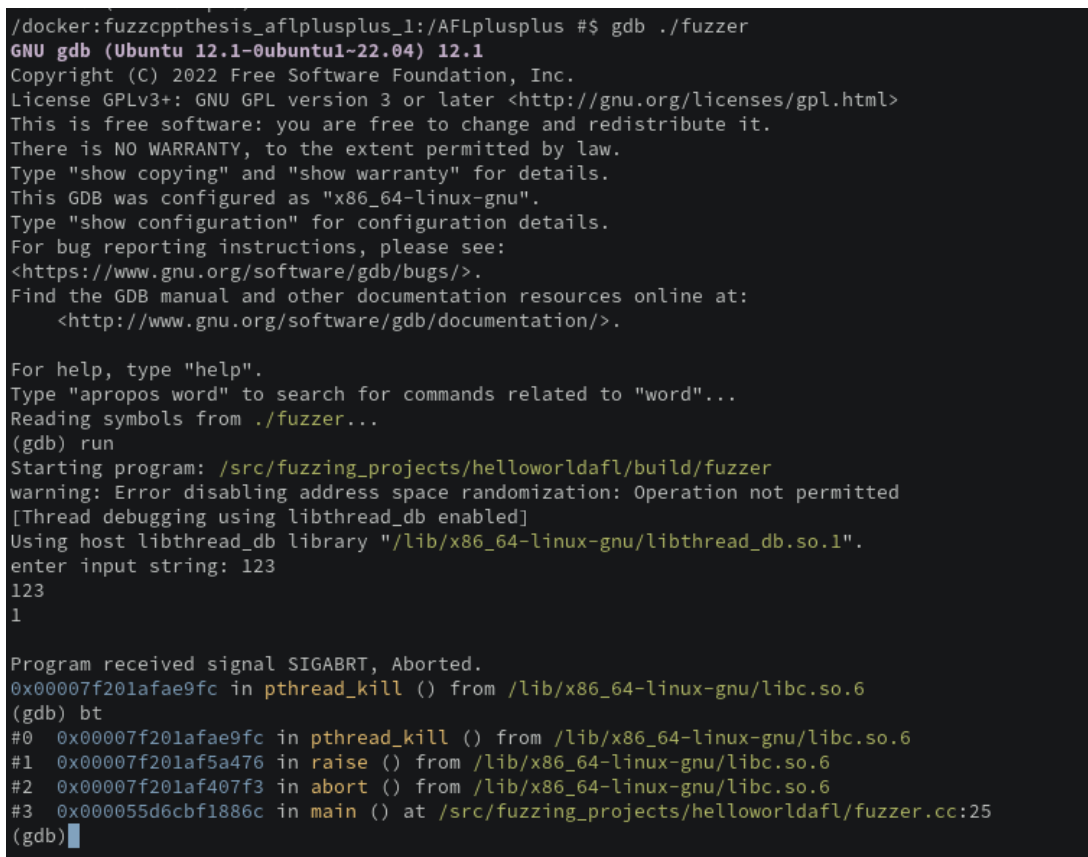
Due to the short size of the harness, the findings of crashes did not take long.



```
jqfedora:~/TresorHiles/Code/smiles/github/FHost/FHost-gsea/fuzzcppthesis/fuzzing_projects/_afl_results/fuzzer_run_helloworldafl_240303/out/default/crashes$ ll  
total 16  
drwx----- 1 i i 386 Mar 4 16:16 .  
drwx----- 1 i i 158 Mar 3 17:38 ..  
-rw----- 1 i i 136 Mar 3 17:36 id:000000,sig:06,src:000003,time:117,execs:126,op:havoc,rep:3  
-rw----- 1 i i 143 Mar 3 17:36 id:000001,sig:06,src:000003,time:492,execs:408,op:havoc,rep:4  
-rw----- 1 i i 136 Mar 3 17:36 id:000002,sig:06,src:000003,time:629,execs:455,op:havoc,rep:3  
-rw----- 1 i i 629 Mar 3 17:36 README.txt
```

Figure 6: Screenshot of Crashfiles

The screenshot above 6 shows files that caused a crash when fed as input to the harness.



```
/docker:fuzzcppthesis_aflplusplus_1:/AFLplusplus # $ gdb ./fuzzer  
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1  
Copyright (C) 2022 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law.  
Type "show copying" and "show warranty" for details.  
This GDB was configured as "x86_64-linux-gnu".  
Type "show configuration" for configuration details.  
For bug reporting instructions, please see:  
<https://www.gnu.org/software/gdb/bugs/>.  
Find the GDB manual and other documentation resources online at:  
  <http://www.gnu.org/software/gdb/documentation/>.  
  
For help, type "help".  
Type "apropos word" to search for commands related to "word"...  
Reading symbols from ./fuzzer...  
(gdb) run  
Starting program: /src/fuzzing_projects/helloworldafl/build/fuzzer  
warning: Error disabling address space randomization: Operation not permitted  
[Thread debugging using libthread_db enabled]  
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".  
enter input string: 123  
123  
1  
  
Program received signal SIGABRT, Aborted.  
0x00007f201afae9fc in pthread_kill () from /lib/x86_64-linux-gnu/libc.so.6  
(gdb) bt  
#0 0x00007f201afae9fc in pthread_kill () from /lib/x86_64-linux-gnu/libc.so.6  
#1 0x00007f201af5a476 in raise () from /lib/x86_64-linux-gnu/libc.so.6  
#2 0x00007f201af407f3 in abort () from /lib/x86_64-linux-gnu/libc.so.6  
#3 0x000055d6cbf1886c in main () at /src/fuzzing_projects/helloworldafl/fuzzer.cc:25  
(gdb)
```

Figure 7: Screenshot of GDB with Stacktrace

To analyse the root cause of the crash, we can simply feed in the crashfile as input to the harness. In figure 7 the string “123” was used to cause a crash and demonstrate the behaviour.

## 6.2 libxml2

To gain a deeper understanding of the fuzzing process, libxml2 with a known vulnerability was chosen. [27, 28]

The results and metrics where only used for learning and are not included in this thesis.



## 6.3 contour

Contour Terminal is a modern, open-source terminal emulator designed for performance and extensive customization. [29]

Its architecture includes various components, with the VT parser being crucial for handling and interpreting terminal control sequences and escape codes, ensuring accurate text rendering and functionality. The VT parser, short for Virtual Terminal parser, processes sequences used for cursor movement, text formatting, and other terminal behaviors, which are essential for the terminal's interaction with shell applications. By efficiently parsing and executing these sequences, the VT parser contributes to the terminal's responsiveness and correctness. Contour's design emphasizes modularity, allowing the VT parser and other components to be independently developed and optimized.

### 6.3.1 Harness Setup

Requirements:

- Install sudo in container Appendix helper-bash-container-patch
- Ensure cmake-version is new enough (3.29 in this case) Appendix helper-bash-cmake-install

Given that Contour is based on CMake and features a modern modular design, a harness was written by reusing already existing Unit-Test Components.

```
int main()
{
    string str;
    MockParserEvents textListener;
    auto p = vtparser::Parser<vtparser::ParserEvents>(textListener);
    getline(cin, str);
    p.parseFragment(str);
}
```

and the CMake-File supplemented by this lines:

```
add_executable(vtparser_fuzz
    main.cpp
)
target_link_libraries(vtparser_fuzz vtparser)
```

The entire project was built following the developers' instructions, with the exception of enabling LTO (Link Time Optimization) and ASAN (AddressSanitizer).

```
AFL_USE_ASAN=1 CC=afl-clang-lto CXX=afl-clang-lto++ cmake --preset linux-release
AFL_USE_ASAN=1 CC=afl-clang-lto CXX=afl-clang-lto++ cmake --build --preset linux-release
```

This configuration utilizes AFL-Clang-LTO mode, which applies Link Time Optimization to instrument the program at the intermediate representation (IR) level. This approach offers more precise control, enhanced performance, and improved bug detection compared to traditional source-level instrumentation. It is especially advantageous for large or complex codebases, where standard instrumentation might introduce significant performance overhead.

### 6.3.2 Threaded Fuzzing

AFL++ supports multithreading, allowing for more efficient and faster fuzzing by leveraging multiple CPU cores simultaneously. In this project, the harness was configured to run on 20 cores to maximize the utilization of available processing power and improve the overall speed and effectiveness of the fuzzing process.

The script for the threaded fuzzing is here: Appendix contour-threaded-fuzzing

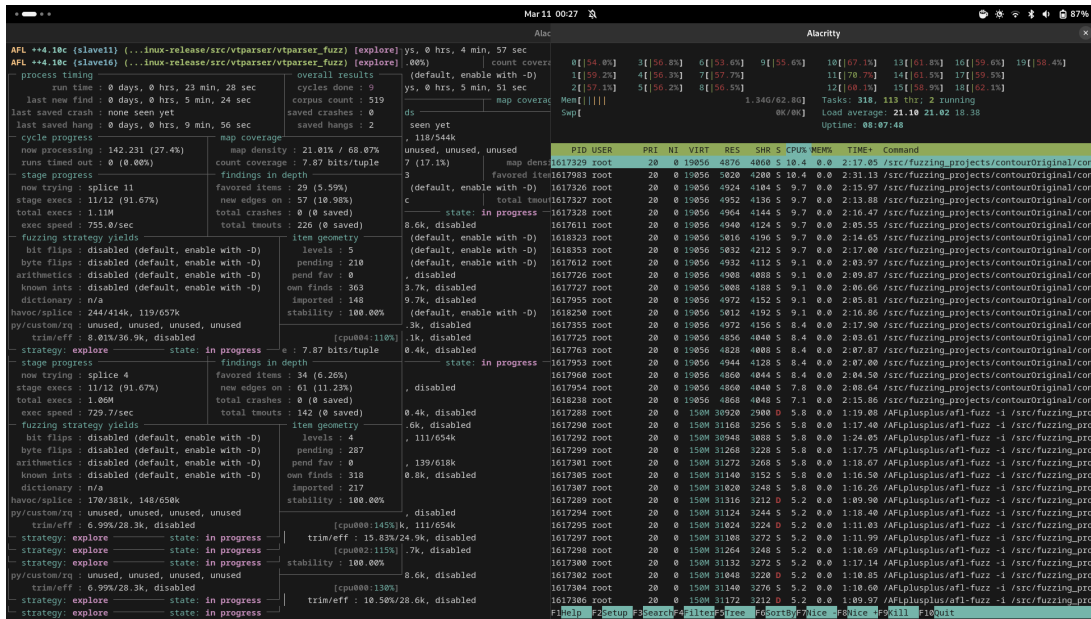


Figure 8: Screenshot of AFL++ with htop.

As visible in the screenshot 8 many instances were executed in parallel. It also shows that the fuzzer observed 2 hangs in the meanwhile. This was mainly related to very long input size.

### 6.3.3 Results

The fuzzer was executed twice using the original code, which is still in the development phase. For the seed inputs, ANSI crash sequences and ANSI escape sequences were provided in their un-minified form, as they would otherwise be minimized to empty strings. Appendix helper-bash-seed-minimizer

The **contour/vtparser** primarily utilizes standard libraries like **std::fmt**, which are advantageous due to their robustness, reliability. Standard functions are well-tested and optimized for performance, ensuring that the code is maintainable and less prone to errors. Additionally, contour leverages modern C++ features, including extensive use of namespaces and advanced C++11/14/17 standards, which contribute to a clean and modular codebase.

During the initial runs with the original code, despite encountering two hangs, no crashes were detected. This highlights the stability and resilience of the existing implementation.

To thoroughly test the effectiveness of the fuzzer, a deliberate modification was made to **Parser.cpp** to introduce a fixed-size buffer vulnerability. This proof of concept (PoC) was specifically created to challenge the fuzzer by embedding a known vulnerability. The modification is detailed below:

```
// Introduce fixed-size buffer vulnerability here
std::array<char, 10> buffer; // Small fixed-size buffer
size_t pos = 0;

for (auto const&& [rangeCount, u]: enumerate(t.second))
{
    if (rangeCount)
    {
        if (pos >= buffer.size())
        {
            // Vulnerability: writing beyond the buffer bounds
            buffer[pos++] = ',';
        }
    }
}
```

```

    }
    if (rangeCount % 3 == 0)
    {
        buffer[pos++] = '\\';
        buffer[pos++] = 'n';
    }
}
if (u.first == u.last)
    pos += snprintf(buffer.data() + pos, buffer.size() - pos, "%02X", u.first);
else
    pos += snprintf(buffer.data() + pos, buffer.size() - pos, "%02X-%02X", u.first, u.last);
}

os << buffer.data(); // Potentially overflowed buffer content
os << "\"";
os << "]";
os << ";\n";

```

This intentional vulnerability led to numerous crashes, effectively demonstrating the fuzzer’s capability to detect and handle buffer overflows. To systematically analyze these crashes, a Python script was used, as described in the Appendix `contour-crash-analysis`. The analysis clearly indicated that the crashes were caused by the buffer overflow introduced by the PoC modification, validating the fuzzer’s effectiveness in identifying such issues.

## 6.4 NASA HDTN (Tryout)

The High-Data-Rate Delay-Tolerant Networking (HDTN) system developed by NASA is designed to efficiently handle large volumes of data in challenging networking environments. HDTN aims to facilitate robust data transmission over networks with high latency and potential disruptions, making it ideal for space communications.

NASA also provides bounties for findings in their codebase.

In this tryout, the code and architecture of HDTN were thoroughly analyzed. Despite this detailed examination, no effective harness points were identified for fuzz testing. This is largely due to NASA’s extensive use of standard functions and the inherently resilient design of HDTN. The system is engineered to handle corrupted files and adverse conditions with high reliability, which makes it less susceptible to common vulnerabilities that fuzzers typically exploit. [30] HDTN’s architecture is modular and leverages modern C++ standards, including extensive use of namespaces and advanced programming techniques.

As a result, the project was not subjected to fuzz testing since no suitable attack points were found. The resilience of HDTN to corrupted data and its reliance on well-established standard libraries underline the system’s robustness. These factors contribute to the challenge of finding effective harness points for fuzz testing, as the codebase is inherently designed to manage and mitigate typical sources of failure. Despite these challenges, the analysis of HDTN’s architecture and codebase provides valuable insights into its robust design and operational reliability.

The architecture diagram 9 provides a comprehensive overview of HDTN. [31]

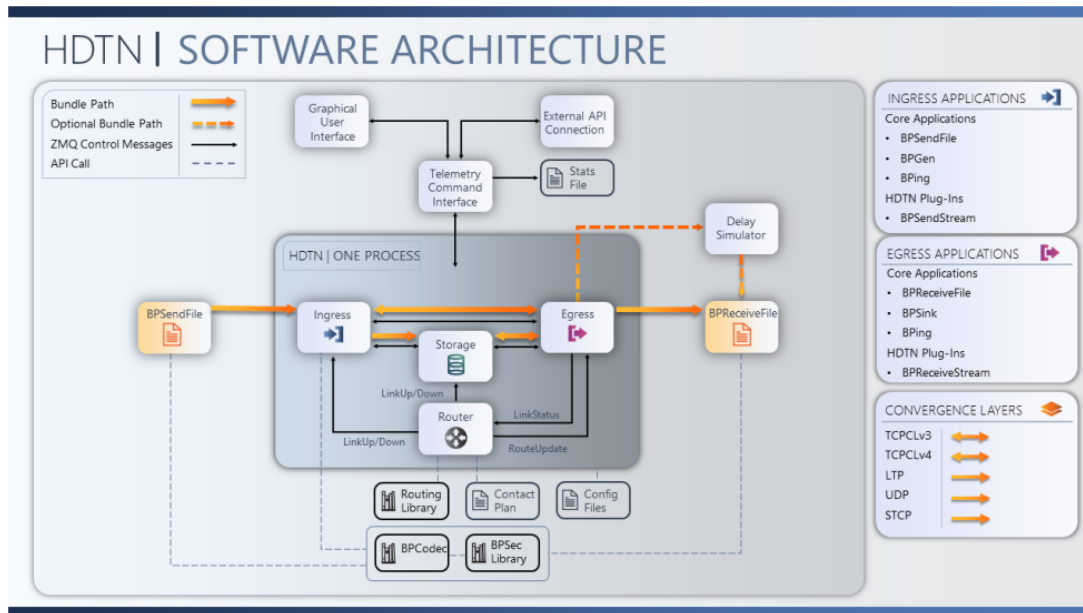


Figure 9: HDTN Software Architecture (NASA User Guide).

## 6.5 libBLS

libBLS [32] is a cryptographic library that implements Boneh-Lynn-Shacham (BLS) signatures. BLS signatures are a form of short digital signatures that allow for secure and efficient aggregation of multiple signatures into a single compact one. This property makes BLS signatures particularly useful in scenarios where bandwidth or storage is limited, such as in blockchain technologies, distributed systems, and secure messaging protocols.

The library leverages another cryptographic library called libff [33], which provides finite field and elliptic curve arithmetic. libff is crucial for performing the underlying mathematical operations required for BLS signatures, including operations over large prime fields and elliptic curve groups. Together, libBLS and libff enable the creation, aggregation, and verification of BLS signatures with high efficiency and security.

The expectations for finding bugs are high considering that the project is still under development.

### 6.5.1 Harness

A harness was created to test the functionality of libBLS by automating the process of signing and verifying input messages. The harness performs the following steps:

- **Private Key Generation:** It generates private keys required for signing the messages.
- **Message Signing:** It signs the input messages using a threshold scheme with multiple signers.
- **Signature Verification:** It verifies the aggregated signature to ensure its validity.

If any of these steps fail, the harness will invoke `std::abort` to terminate the program, resulting in a crash. This ensures that any issues or vulnerabilities within the library are promptly detected during testing. The harness thus serves as an effective tool for validating the correctness and robustness of the BLS signature implementation in libBLS. Appendix libbbs-harness

To build the whole project, first a simple fix has to be applied Appendix libbbs-patch which is in the latest “dev”-branch on libbbs. After that, the harness can be injected and build together with the main program. Appendix libbbs-build



```

AFL ++4.10c {default} (/root/fuzzing_tcpdump/install/bin/tcpdump) [explore]
┌── process timing ────────────────────────────┐ ┌── overall results ────────────────────┐
│ run time : 0 days, 0 hrs, 0 min, 7 sec      │ │ cycles done : 0                      │
│ last new find : 0 days, 0 hrs, 0 min, 0 sec │ │ corpus count : 1350                  │
│ last saved crash : none seen yet            │ │ saved crashes : 0                    │
│ last saved hang : none seen yet            │ │ saved hangs : 0                      │
├── cycle progress ───────────────────────────┐ │ ┌── map coverage ───────────────────┐ │
│ now processing : 869.0 (64.4%)             │ │ │ map density : 0.90% / 12.97%     │ │
│ runs timed out : 0 (0.00%)                │ │ │ count coverage : 2.23 bits/tuple  │ │
├── stage progress ───────────────────────────┐ │ │ ┌── findings in depth ───────────┐ │ │
│ now trying : havoc                         │ │ │ favored items : 416 (30.81%)     │ │ │
│ stage execs : 2387/9600 (24.86%)          │ │ │ new edges on : 478 (35.41%)     │ │ │
│ total execs : 11.9k                       │ │ │ total crashes : 0 (0 saved)     │ │ │
│ exec speed : 381.9/sec                    │ │ │ total tmouts : 0 (0 saved)     │ │ │
├── fuzzing strategy yields ─────────────────┐ │ │ ┌── item geometry ─────────────────┐ │ │
│ bit flips : disabled (default, enable with -D) │ │ │ levels : 2                      │ │ │
│ byte flips : disabled (default, enable with -D) │ │ │ pending : 564                    │ │ │
│ arithmetics : disabled (default, enable with -D) │ │ │ pend fav : 413                    │ │ │
│ known ints : disabled (default, enable with -D) │ │ │ own finds : 72                     │ │ │
│ dictionary : havoc mode                     │ │ │ imported : 0                       │ │ │
│ havoc/splice : 0/0, 0/0                     │ │ │ stability : 100.00%               │ │ │
│ py/custom/rq : unused, unused, unused, unused │ │ │ ┌── [cpu000: 5%] ───────────┐ │ │
│ trim/eff : 0.00%/76, disabled              │ │ │ └── [cpu000: 5%] ───────────┐ │ │
├── strategy: explore ─────────────────────────┐ │ └── state: started :- ───────────┐ │
└── strategy: explore ─────────────────────────┐ │ └── state: started :- ───────────┐ │

```

Figure 11: Screenshot of AFL++.

```

AFL ++4.10c {default} (/root/fuzzing_tcpdump/install/bin/tcpdump) [explore]
┌── process timing ────────────────────────────┐ ┌── overall results ────────────────────┐
│ run time : 0 days, 0 hrs, 14 min, 53 sec   │ │ cycles done : 0                      │
│ last new find : 0 days, 0 hrs, 0 min, 1 sec │ │ corpus count : 2029                  │
│ last saved crash : none seen yet            │ │ saved crashes : 0                    │
│ last saved hang : none seen yet            │ │ saved hangs : 0                      │
├── cycle progress ───────────────────────────┐ │ ┌── map coverage ───────────────────┐ │
│ now processing : 1833.0 (90.3%)            │ │ │ map density : 0.63% / 14.03%     │ │
│ runs timed out : 0 (0.00%)                │ │ │ count coverage : 2.29 bits/tuple  │ │
├── stage progress ───────────────────────────┐ │ │ ┌── findings in depth ───────────┐ │ │
│ now trying : trim 4/4                     │ │ │ favored items : 779 (38.39%)     │ │ │
│ stage execs : 30/138 (21.74%)             │ │ │ new edges on : 859 (42.34%)     │ │ │
│ total execs : 191k                        │ │ │ total crashes : 0 (0 saved)     │ │ │
│ exec speed : 423.6/sec                    │ │ │ total tmouts : 11.3k (0 saved)   │ │ │
├── fuzzing strategy yields ─────────────────┐ │ │ ┌── item geometry ─────────────────┐ │ │
│ bit flips : disabled (default, enable with -D) │ │ │ levels : 4                      │ │ │
│ byte flips : disabled (default, enable with -D) │ │ │ pending : 1195                    │ │ │
│ arithmetics : disabled (default, enable with -D) │ │ │ pend fav : 741                    │ │ │
│ known ints : disabled (default, enable with -D) │ │ │ own finds : 752                     │ │ │
│ dictionary : havoc mode                     │ │ │ imported : 0                       │ │ │
│ havoc/splice : 576/146k, 176/19.2k         │ │ │ stability : 100.00%               │ │ │
│ py/custom/rq : unused, unused, unused, unused │ │ │ ┌── [cpu000: 5%] ───────────┐ │ │
│ trim/eff : 2.57%/11.3k, disabled           │ │ │ └── [cpu000: 5%] ───────────┐ │ │
├── strategy: explore ─────────────────────────┐ │ └── state: in progress ───────────┐ │
└── strategy: explore ─────────────────────────┐ │ └── state: in progress ───────────┐ │

```

Figure 12: Screenshot of AFL++.

### 6.6.1 AFL-Plot Analysis

This plot 13 shows the number of unique edges (or transitions between basic blocks in the target program) discovered over time. It indicates how thoroughly the fuzzer is exploring new execution paths within the target program. A steady increase suggests that the fuzzer is continually finding new areas to explore.

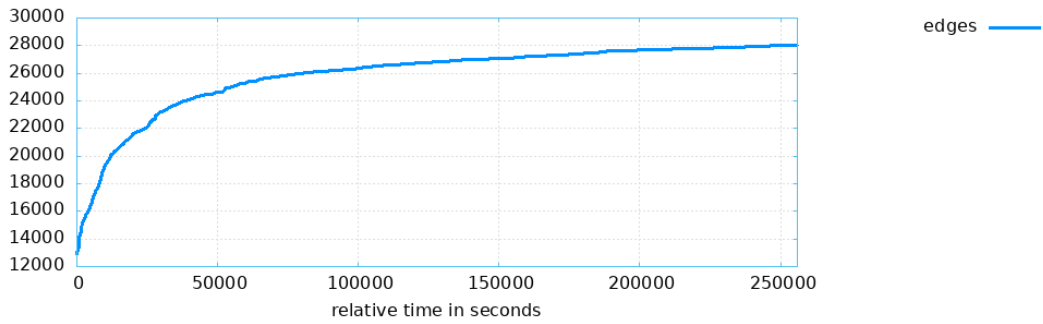


Figure 13: TCPDump-Edges

This image 14 represents the execution speed of the fuzzer, measured in executions per second. It provides insights into how efficiently the fuzzer is running. High execution speed can indicate efficient fuzzing, while a drop might suggest performance bottlenecks or increased complexity in the test cases being executed.

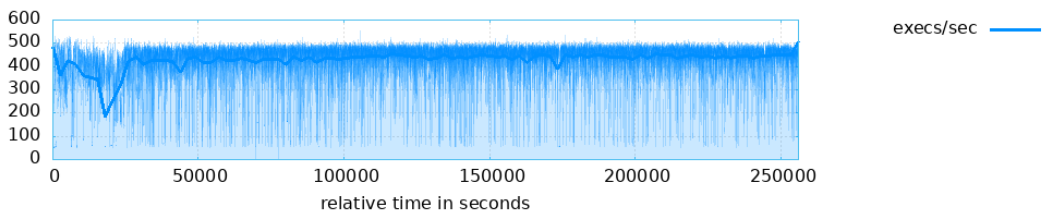


Figure 14: TCPDump-ExecSpeed

This graph 15 shows the frequency of test case executions that hit the most frequently accessed code paths. It helps in identifying if certain parts of the code are being overly targeted, which could lead to biased fuzzing results. Ideally, the fuzzer should cover a wide range of code paths rather than focusing too much on a few.

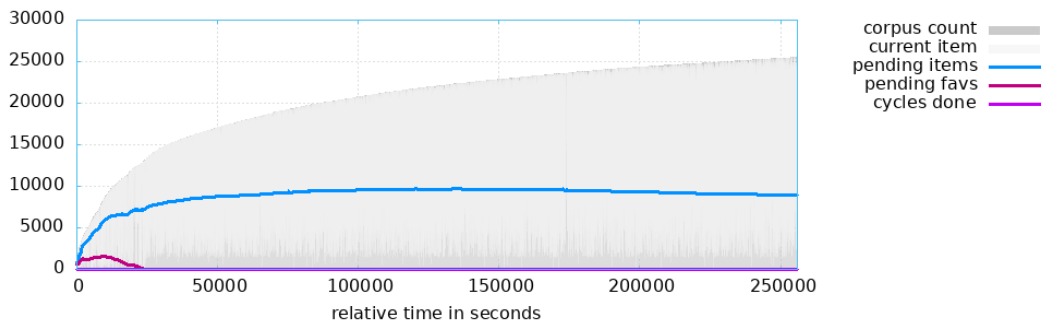


Figure 15: TCPDump-HighFrequency

This plot 16 displays the frequency of executions for the least accessed code paths. It is useful for identifying areas of the code that are less frequently tested, which might be critical for finding edge case bugs. Increasing coverage in these areas can improve the overall effectiveness of the fuzzing campaign.



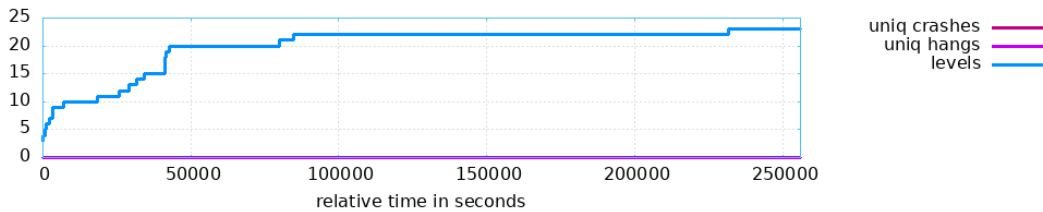


Figure 16: TCPDump-LowFrequency

These plots together provide a comprehensive view of the fuzzing process, highlighting the areas of the target program being explored, the efficiency of the fuzzing, and potential areas for improvement in test coverage.

## 6.7 libjpeg-turbo

libjpeg-turbo is a JPEG image codec that uses SIMD instructions (MMX, SSE2, NEON, AltiVec) to accelerate baseline JPEG compression and decompression on x86, x86-64, ARM, and PowerPC systems, as well as progressive JPEG compression on x86 and x86-64 systems. [35]

To set up libjpeg-turbo for fuzz testing, the following steps were executed: Appendix libjpeg-build

During the fuzzing process, the tool did not identify any crashes or hangs.

## 6.8 libexpat

libexpat is an XML parser library written in C. It is a stream-oriented parser in which an application registers handlers for things the parser might find in the XML document (like start tags). [36]

To set up libexpat for fuzz testing, the following steps were executed: Appendix libexpat-build  
During the fuzzing process, the tool did not identify any crashes or hangs.

## 6.9 pdfrack

pdfrack is a command-line tool designed to recover passwords from PDF files, enabling users to gain access to protected documents by attempting to brute-force the password. It supports both user and owner password recovery, making it a versatile utility for individuals who have lost access to their PDF files. The tool is particularly useful in scenarios where the password is not known, and access to the contents of the document is crucial. [37]

The project was suggested by the Advisor of this thesis due to known vulnerabilities.

The build and start process of the harness can be seen here: Appendix pdfrack-build

To fuzz pdfrack, a variety of encrypted PDF files were generated using the following resources from archive.org:

- Jane Eyre by Charlotte Brontë [38]
- First Love and Other Stories by Ivan Turgenev [39]
- The Merry-Go-Round by W. Somerset Maugham [40]

The encrypted samples were generated as described in Appendix pdfrack-bash-generate-encrypted-samples.

The fuzzing process successfully identified a number of issues within pdfrack. Notably, several crashes were detected and analyzed. The detailed analysis of these crashes revealed the following: The crashfiles were first minimized and a detailed explanation generated using Appendix pdfrack-bash-minimize-and-analyse-crashes



The Video Tutorial on pdfcrack [41] was also analyzed after the initial fuzzer run with findings, but could not be reproduced due to missing input files.

### 6.9.1 GDB Analysis

When using GDB to reproduce and verify the crash, one gets a similar output to 17

```
(gdb) run
Starting program: /root/fuzzing_pdfcrack/pdfcrack-0.15/pdfcrack -f /root/fuzzing_pdfcrack/out/default/crashes/id:000000,sig:06,src:000259
+000447,time:9437102,execs:1575855,op:splice,rep:8
warning: Error disabling address space randomization: Operation not permitted
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
=====
==115756==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7f9303800320 at pc 0x56524c3bb911 bp 0x7ffe0a6506f0 sp 0x7ffe0a6506e8
e8
READ of size 1 at 0x7f9303800320 thread T0
[Detaching after fork from child process 115759]
#0 0x56524c3bb910 in objStringToByte /root/fuzzing_pdfcrack/pdfcrack-0.15/pdfparser.c:230:14
#1 0x56524c3b6c8f in parseRegularString /root/fuzzing_pdfcrack/pdfcrack-0.15/pdfparser.c:316:13
#2 0x56524c3b1c6e in findTrailerDict /root/fuzzing_pdfcrack/pdfcrack-0.15/pdfparser.c:399:12
#3 0x56524c3b1c6e in getEncryptedInfo /root/fuzzing_pdfcrack/pdfcrack-0.15/pdfparser.c:805:13
#4 0x56524c368d16 in main /root/fuzzing_pdfcrack/pdfcrack-0.15/main.c:252:11
#5 0x7f930536fd8f (/lib/x86_64-linux-gnu/libc.so.6+0x29d8f) (BuildId: c289da5071a3399de893d2af81d6a30c62646e1e)
#6 0x7f930536fe3f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x29e3f) (BuildId: c289da5071a3399de893d2af81d6a30c62646e1e)
#7 0x56524c28ee04 in _start (/root/fuzzing_pdfcrack/pdfcrack-0.15/pdfcrack+0x5ae04) (BuildId: 8ac4c024dcac32c3)

Address 0x7f9303800320 is located in stack of thread T0 at offset 288 in frame
#0 0x56524c3b654f in parseRegularString /root/fuzzing_pdfcrack/pdfcrack-0.15/pdfparser.c:289

This frame has 1 object(s):
[32, 288) 'buf' (line 293) <== Memory access at offset 288 overflows this variable
HINT: this may be a false positive if your program uses some custom stack unwind mechanism, swapcontext or vfork
(longjmp and C++ exceptions +are+ supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow /root/fuzzing_pdfcrack/pdfcrack-0.15/pdfparser.c:230:14 in objStringToByte
Shadow bytes around the buggy address:
0x7f9303800080: f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8
0x7f9303800100: f8 f8 f8 f8 f3 f3 f3 f3 f3 f3 00 00 00 00
0x7f9303800180: 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x7f9303800200: f1 f1 f1 f1 00 00 00 00 00 00 00 00 00 00
0x7f9303800280: 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x7f9303800300: 00 00 00 00[f3]f3 f3 f3 f3 f3 f3 00 00 00
0x7f9303800380: 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x7f9303800400: 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x7f9303800480: 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x7f9303800500: 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x7f9303800580: 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
Addressable: 00
Partially addressable: 01 02 03 04 05 06 07
Heap left redzone: fa
Freed heap region: fd
Stack left redzone: f1
Stack mid redzone: f2
Stack right redzone: f3
Stack after return: f5
Stack use after scope: f8
Global redzone: f9
Global init order: f6
Poisoned by user: f7
Container overflow: fc
Array cookie: ac
Intra object redzone: bb
ASan internal: fe
Left alloca redzone: ca
Right alloca redzone: cb
==115756==ABORTING
[Inferior 1 (process 115756) exited with code 01]
(gdb)
```

Figure 17: Screenshot of GDB

When breaking at the stated line, one could partially see the local variables and the call to the suspicious function. 18

```
(gdb) break pdfparser.c:230
Breakpoint 1 at 0x55e254c19215: file pdfparser.c, line 230.
(gdb) run
Starting program: /root/fuzzing_pdfcrack/pdfcrack-0.15/pdfcrack -f /root/fuzzing_pdfcrack/out/default/crashes/id:000000,sig:06,src:000259+000447,time:9437102,execs:1575855,op:splice,rep:8
warning: Error disabling address space randomization: Operation not permitted
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, objStringToByte (str=0x7ff609900220 "6\337o\310r\207246\023\{'\;L\244 0 R /GS2 244 0 R /GS2 2250 0 R\360\377\377\377endobj\n1744 0 obj\n<<1 0 \275 /ID \n1745 0 obj\n<< /J2i0 2470 0 R\324\344\274\n5\333R\202\t8\036C\336L", len=256) at pdfparser.c:230
230      switch(str[i]) {
(gdb) info locals
_vla_expr0 = 256
tmp = "6\337o\310r\207246\023\{'\;L", '\000' <repeats 242 times>
i = <optimized out>
l = 13
b = <optimized out>
j = <optimized out>
d = <optimized out>
ret = <optimized out>
(gdb)
```

Figure 18: Screenshot of GDB Breakpoint

We can also see the traces of the call. 19

```
(gdb) bt
#0  objStringToByte (
    str=0x7ff609900220 "6\337o\310r\207246\023\{'\;L\244 0 R /GS2 244 0 R /GS2 2250 0 R\360\377\377\377endobj\n1744 0 obj\n<<1 0 \275 /ID
\n1745 0 obj\n<< /J2i0 2470 0 R\324\344\274\n5\333R\202\t8\036C\336L", len=256) at pdfparser.c:230
#1  0x000055bf5476ac90 in parseRegularString (file=<optimized out>) at pdfparser.c:316
#2  0x000055bf54765c6f in findTrailerDict (file=0x6150000000080, e=0x6060000000020) at pdfparser.c:399
#3  getEncryptedInfo (file=0x6150000000080, e=<optimized out>) at pdfparser.c:805
#4  0x000055bf5471cd17 in main (argc=<optimized out>, argv=<optimized out>) at main.c:252
```

Figure 19: Screenshot of GDB Traces

## 6.9.2 Detailed Analysis

**Stack-Buffer-Overflow:** A significant error was uncovered involving a stack-buffer-overflow in the ‘pdfparser.c’ file, precisely at line 230 within the ‘objStringToByte’ function. The overflow was triggered by improper bounds handling of an array used to process strings within the PDF files.

The cause analysis and the details of these crashes are described in Appendix pdfcrack-python-crash-cause-analysis. It returns the same location for every crash.

ID: id:0000XX, Cause: stack-buffer-overflow, Location: pdfparser.c:230:14 in objStringToByte

The crash occurs due to a stack-buffer-overflow, as indicated by the ASan report Appendix pdfcrack-crash-details

- **Address:** 0x7ff609900320
- **Accessed by function:** objStringToByte at source code position ‘pdfparser.c:230:14’
- **Memory access location relative to ‘buf’:** The address is located at offset 288 in a frame where ‘buf’ is defined as [32, 288). This suggests ‘buf’ has a size that can store up to 256 bytes, but the access is at the 288th byte, which is beyond the upper bound of this buffer.

### Explanation of ASan Memory Layout

- ‘=>0x7ff609900300: 00 00 00 00[f3]f3 f3 f3 f3 f3 00 00 00 00’
  - ‘=>:’ This indicates the current line where the overflow or relevant memory access error is being pointed out by ASan.
  - ‘0x7ff609900300:’ This is the base address of the line of memory being displayed.
  - ‘00 00 00 00:’ These bytes are normal and not part of any special zones; they are addressable and currently set to zero.

- ‘[f3]f3 f3 f3 f3 f3 f3:’ The ‘f3’ bytes are part of ASan’s stack right redzone. The square bracket ‘[f3]’ marks the start of the overflow. The point where illegal access begins. ASan uses redzones to detect when a memory operation accesses memory beyond what is allocated for a buffer. Each ‘f3’ byte is used to mark memory that should not be accessed, helping catch overflows like this.
  - ‘00 00 00 00:’ More normal addressable bytes that follow the redzone, indicating the end of the affected memory region.
- **Stack Right Redzone (‘f3’):** These bytes are placed by ASan to detect over-reads and over-writes beyond the allocated stack buffer. Accessing these bytes is indicative of a buffer overflow, where the program writes or reads beyond the memory it was supposed to use. This can lead to corruption of data, crashes, or exploitation of the program.

The presence of the ‘[f3]’ byte at ‘0x7ff609900320’ (as referenced by the error line) indicates that the program attempted to read or write to this redzone, confirming a buffer overflow situation. This means that the code accessed memory it was not supposed to, which is a common security vulnerability and a potential cause of program instability or crashes.

### Practical Implication

- **Buffer Overflows:** This type of error can cause unpredictable behavior in software, including crashes, data corruption, and security vulnerabilities that might allow attackers to execute arbitrary code.
- **Fixing the Issue:** The root cause of such overflows often lies in incorrect calculations for buffer sizes or failing to properly check boundaries when accessing buffers. Code review and proper testing, including the use of tools like ASan during development, are crucial to identify and correct such issues before software is deployed.

Lets have a look at this code.

```
static p_str*
objStringToByte(const uint8_t* str, const unsigned int len) {
    unsigned int i, j, l;
    uint8_t b, d;
    uint8_t tmp[len];
    p_str *ret;

    for(i=0, l=0; i<len; i++, l++) {
        b = str[i];
        if(b == '\\') {
            /**
             * We have reached a special character or the beginning of a octal
             * up to three digit number and should skip the initial backslash
             */
            i++;
            switch(str[i]) {
                case 'n':
                    b = 0x0a;
                    break;
                case 'r':
                    b = 0x0d;
                    break;
                case 't':
                    b = 0x09;
                    break;
                case 'b':
                    b = 0x08;
                    break;
            }
        }
    }
}
```

```

    case 'f':
        b = 0x0c;
        break;
    case '(':
        b = '(';
        break;
    case ')':
        b = ')';
        break;
    case '\\':
        b = '\\';
        break;
    default:
        if(str[i] >= '0' && str[i] < '8') {
            d = 0;
            for(j=0; i < len && j < 3 &&
                str[i] >= '0' && str[i] < '8' &&
                (d*8)+(str[i]-'0') < 256; j++, i++) {
                d *= 8;
                d += (str[i]-'0');
            }

            /**
             * We need to step back one step if we reached the end of string
             * or the end of digits (like for example \000)
             */
            if(i < len || j < 3) {
                i--;
            }

            b = d;
        }
    }
    tmp[1] = b;
}

ret = malloc(sizeof(p_str));
ret->content = malloc(sizeof(uint8_t)*(1));
ret->len = 1-1;

memcpy(ret->content, tmp, 1);

return ret;
}

```

### Source of Error

The provided code snippet from the description of ‘objStringToByte’ processes escape sequences. An error likely occurs when an escape sequence is processed, and the variable ‘i’ or ‘l’ is not correctly managed, leading to an out-of-bound access of ‘tmp’.

- **Variable-Length Array:** The function uses ‘uint8\_t tmp[len];’ which means the buffer size is set based on the input length. If the input length (‘len’) is 256 and any processing of the string increases ‘l’ to exceed this number (e.g., by misunderstanding or miscomputing escape sequences), ‘tmp[l]’ will attempt to write beyond the allocated space.
- **Escape Sequence Handling:** If there is any misinterpretation or incorrect calculation that causes ‘i’ to skip less or more than it should or ‘l’ to increment improperly during the processing of escape sequences, this would lead to accessing ‘tmp’ out of bounds.

## Hypothetical Error-Inducing Input

Given the explanation above, a plausible problematic input could be a string ending with an escape sequence that is not properly closed or parsed.

### 6.10 Libarchive

Libarchive is library that provides an efficient method for reading and writing various archive formats, including tar, cpio, zip, and many others. It is widely used in numerous applications for handling compressed files and archives. The library offers extensive functionality for developers, making it a popular choice for tasks that involve file archiving and extraction. Given its widespread usage and complexity, the library is susceptible to vulnerabilities, which makes it an important target for security analysis and fuzz testing.

The project with version 3.1.2. was suggested by the Advisor of this thesis due to known vulnerabilities. [42]

The build and start process of the harness can be seen here: Appendix libarchive-build

The fuzzing process successfully identified a number of issues within libarchive. Notably, several crashes were detected and analyzed. The detailed analysis of these crashes revealed the following:

The crashfiles were first minimized and a detailed explanation generated using Appendix libarchive-bash-minimize-and-analyze-crashes

Due to the huge amount of crashes, the cause were grouped with the following script Appendix libarchive-python-analyse-crash-cause which still showed alot of different locations. Appendix libarchive-grouping-analysis-of-crashfiles

#### 6.10.1 Analysis of Buffer Overflow in readline Function

The 'readline' function is designed to read lines of text from an archive. However, improper handling of buffer boundaries and memory operations can lead to a heap-buffer overflow. The function reads input data and manages newline characters, escaped sequences, and buffer sizes.

##### 1. Code Breakdown

The function reads data into a buffer and checks for newline characters:

```
t = __archive_read_ahead(a, 1, &bytes_read);
if (t == NULL)
    return (0);
if (bytes_read < 0)
    return (ARCHIVE_FATAL);
s = t; /* Start of line? */
p = memchr(t, '\n', bytes_read);
if (p != NULL) {
    bytes_read = 1 + ((const char *)p) - s;
}
```

It then ensures the buffer is adequately allocated:

```
if (total_size + bytes_read + 1 > limit) {
    archive_set_error(&a->archive,
        ARCHIVE_ERRNO_FILE_FORMAT,
        "Line too long");
    return (ARCHIVE_FATAL);
}
if (archive_string_ensure(&mtree->line,
    total_size + bytes_read + 1) == NULL) {
    archive_set_error(&a->archive, ENOMEM,
        "Can't allocate working buffer");
    return (ARCHIVE_FATAL);
}
```





```

AFL ++4.10c {default} (.../fuzzing_imagemagick/install/bin/magick) [explore]
process timing overall results
| run time : 0 days, 0 hrs, 2 min, 58 sec | cycles done : 1 |
| last new find : none yet (odd, check syntax!) | corpus count : 28 |
| last saved crash : none seen yet | saved crashes : 0 |
| last saved hang : none seen yet | saved hangs : 0 |
cycle progress map coverage
3.59 (10.7%) 12.28% / 12.28% %
0 (0.00%) 5.57 bits/tuple
stage progress findings in depth
splice 2 1 (3.57%)
0/12 (0.00%) 1 (3.57%)
13.8k 0 (0 saved)
82.62/sec (slow!) 0 (0 saved)
fuzzing strategy yields item geometry
disabled (default, enable with -D) 1
disabled (default, enable with -D) 0
disabled (default, enable with -D) 0
disabled (default, enable with -D) 0
n/a 0
0/2962, 0/10.6k 100.00%
unused, unused, unused, unused
98.60%/11, disabled 10%]
strategy: explore state: started :-)

```

Figure 20: Screenshot of AFL

The advisor also provided the blog post and linked to the commit which should fix the issue. A prior version was chosen but even though the issue could not be reproduced. [45] [46]

## 6.12 Analysis and Discussion

### 6.12.1 Metrics

During the fuzzing process, different metrics were collected. Here is a short Explanations of the important Fuzzing Metrics: [16]

- **run-time** This metric indicates the total time the fuzzer has been running. It helps assess the duration of the fuzzing campaign and is crucial for understanding how long it takes to uncover potential vulnerabilities.
- **cycles-done** This represents the number of fuzzing cycles completed. Each cycle typically involves running the target program with various inputs to discover crashes or bugs. A higher number of cycles indicates more thorough testing.
- **time-wo-finds** This metric shows the amount of time since the last unique crash or bug was discovered. It's useful for determining periods of inactivity and might indicate when the fuzzer's effectiveness has plateaued.
- **execs-per-sec** This measures the execution speed of the fuzzer, indicating how many test cases it can run per second. A higher value means the fuzzer is more efficient, allowing more inputs to be tested in a shorter time. 21



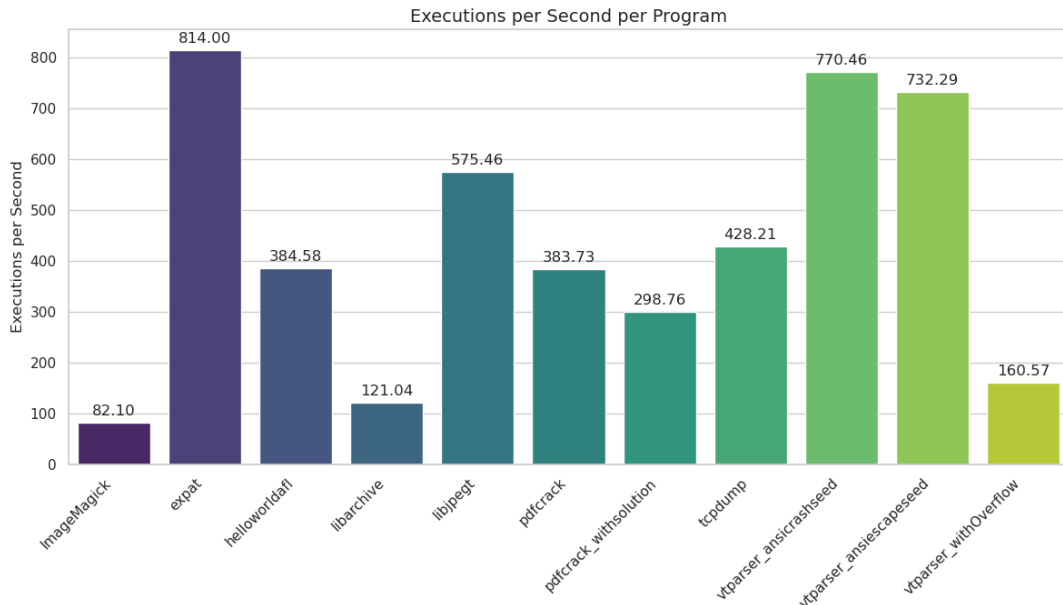


Figure 21: Execution Speed in Execs Per Second

- bitmap-cvg** Bitmap coverage refers to the percentage of code coverage achieved by the fuzzer. It measures how much of the target program's code has been exercised by the test inputs, helping to understand the fuzzer's effectiveness in exploring the program's logic. 22

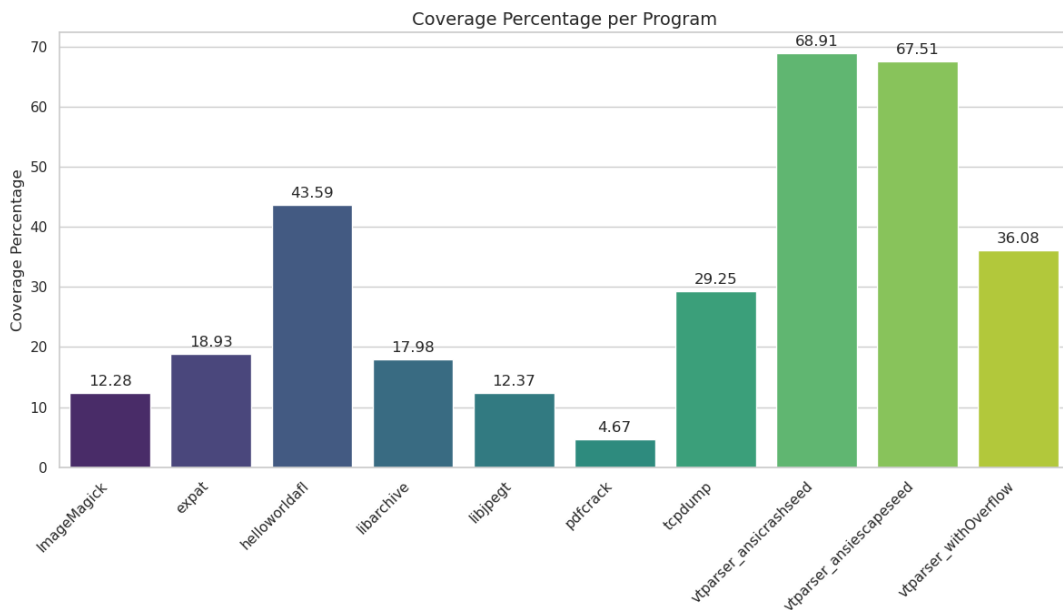


Figure 22: Code Coverage in Bitmap Coverage

- saved-crashes** This metric counts the number of unique crashes saved by the fuzzer. Each crash represents a potential vulnerability, so this number gives an indication of how many bugs have been discovered.
- edges-found** Edges found relate to the unique execution paths discovered during fuzzing. It helps in understanding how many different code paths have been triggered, which is essential for comprehensive testing.

- **total-edges** This is the total number of edges or paths the fuzzer has discovered in the program’s control flow graph. It indicates the overall coverage and thoroughness of the fuzzing campaign.

For the aggregation a python script was used Appendix python-analysis-metrics together with this config Appendix Config for Python Analysis Script for Metric. Since the projects in contour (the vtparser fuzzes) were executed in a multithreaded manner, the mean of the values was chosen.

Table 2: Metrics Aggregation for Fuzzing Programs (Run-time, Cycles-done, Time-without-finds, Execs-per-sec)

Program	run-time	cycles-done	time-wo-finds	execs-per-sec
ImageMagick	65.3	2795.0	0.0	82.1
expat	69.8	7456.0	149105.0	814.0
helloworldafl	0.0	10.0	98.0	384.6
libarchive	61.4	6.0	2006.0	121.0
libjpeg	68.9	234.0	81554.0	575.5
pdfcrack	191.4	384.0	116385.0	383.7
pdfcrack <sub>withsolution</sub>	239.6	1386.0	157879.0	298.8
tcpdump	71.1	7.0	452.0	428.2
vtparser <sub>ansicrashseed</sub>	1.5	37.1	1633.1	770.5
vtparser <sub>ansiescapeseed</sub>	0.3	7.2	190.6	732.3
vtparser <sub>withOverflow</sub>	140.5	293.7	270622.7	160.6

Table 3: Metrics Aggregation for Fuzzing Programs (Bitmap-cvg, Saved-crashes, Edges-found, Total-edges)

Program	bitmap-cvg	saved-crashes	edges-found	total-edges
ImageMagick	12.3	0.0	7.0	57.0
expat	18.9	0.0	141.0	745.0
helloworldafl	43.6	3.0	17.0	39.0
libarchive	18.0	387.0	11110.0	61777.0
libjpeg	12.4	0.0	1441.0	11649.0
pdfcrack	4.7	75.0	345.0	7393.0
pdfcrack <sub>withsolution</sub>	0.0	0.0	378.0	8388608.0
tcpdump	29.2	0.0	28056.0	95921.0
vtparser <sub>ansicrashseed</sub>	68.9	0.0	246.0	357.0
vtparser <sub>ansiescapeseed</sub>	67.5	0.0	241.0	357.0
vtparser <sub>withOverflow</sub>	36.1	3.0	364.0	1009.0

- **Coverage vs Runtime** This graph 23 plots code coverage against the run time. There is no trend visible, that would show that very high run-times would lead to higher coverage.

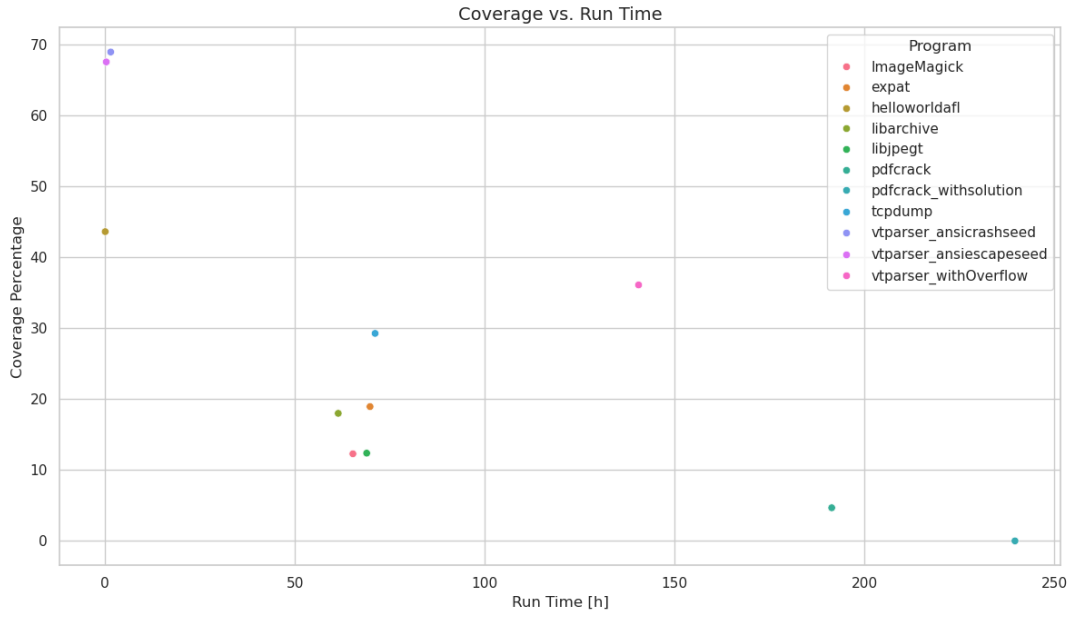


Figure 23: Code Coverage vs Runtime

- **Run Time Distribution** This graph shows the distribution of run times across different fuzzing sessions. The projects were let run from many hours, to weeks. 24

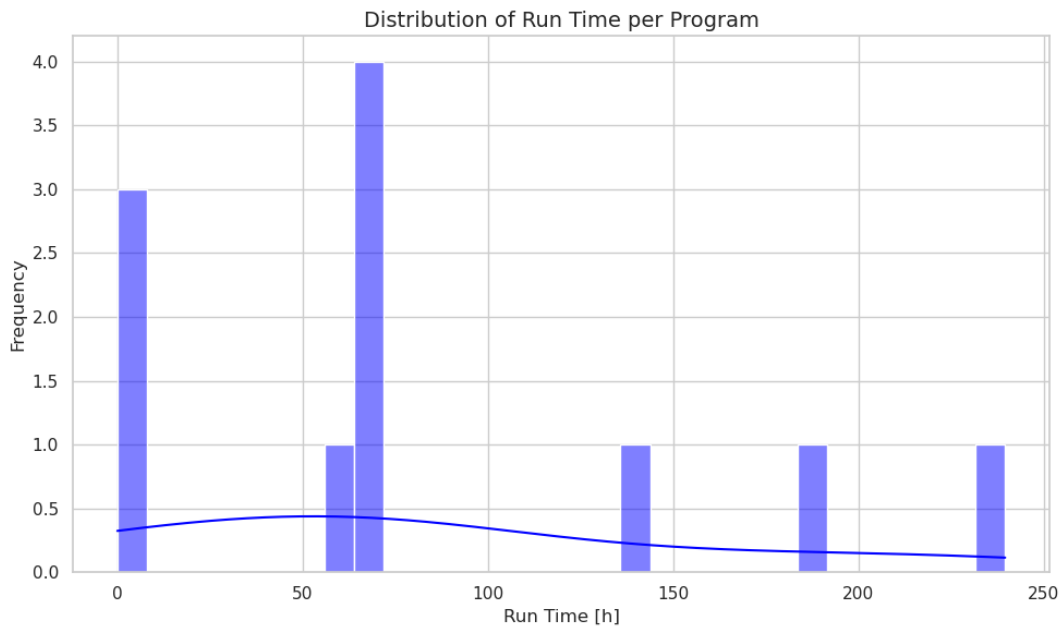


Figure 24: Distribution of Run Times

- **Edges Found Distribution** This graph shows the distribution of edges (unique paths) found during fuzzing. A higher number of edges found indicates better exploration of the code. The distribution also varies heavily under the different projects. 25

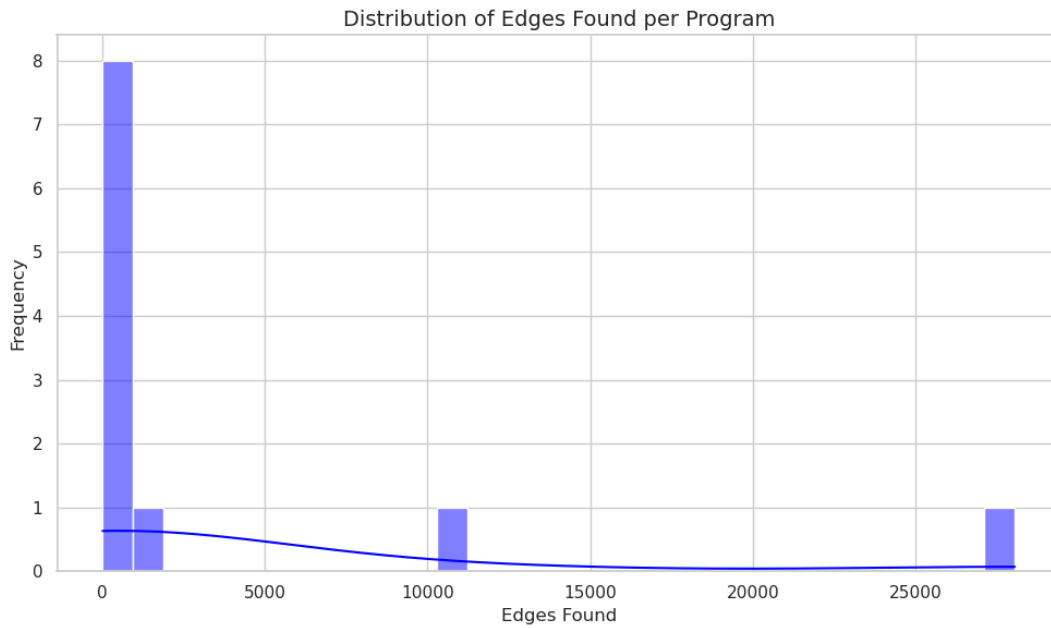


Figure 25: Distribution of Edges Found

- Bitmap Coverage Distribution** This graph shows the distribution of bitmap coverage achieved in different fuzzing sessions. Higher values are better, indicating more comprehensive code coverage. As visible, there are various different percentage rates over the projects, peaking at approximately 15-20%. This is also related to the size of the program which was fuzzed; a codebase which handles a higher amount of edge-cases would have a lower coverage since it would take longer to explore all the branches. 26

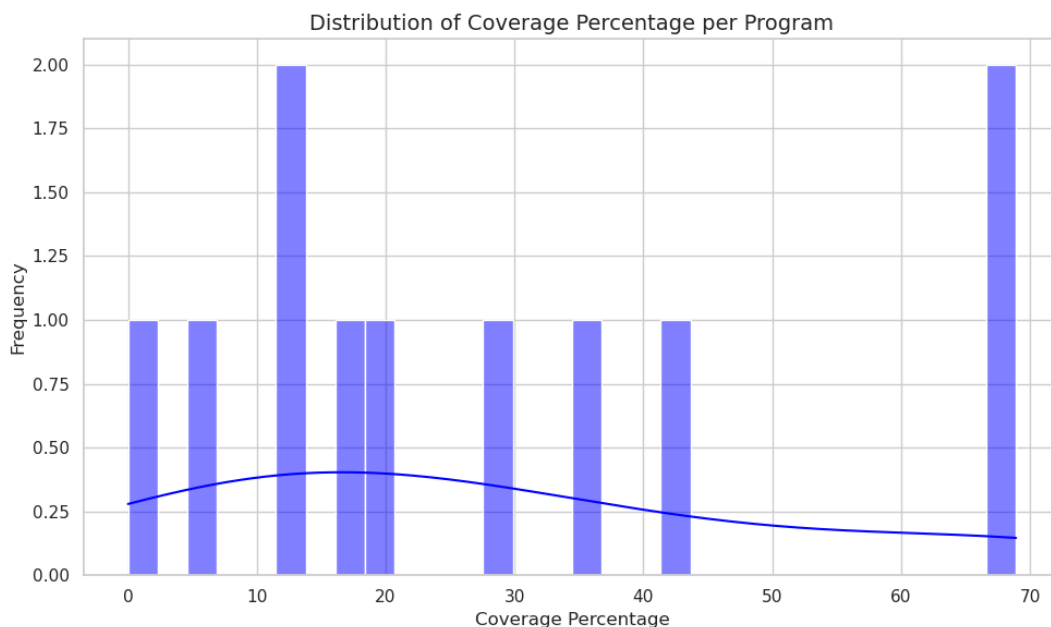


Figure 26: Distribution of Bitmap Coverage

- Correlation Heatmap** This heatmap shows the correlation between different metrics. 27

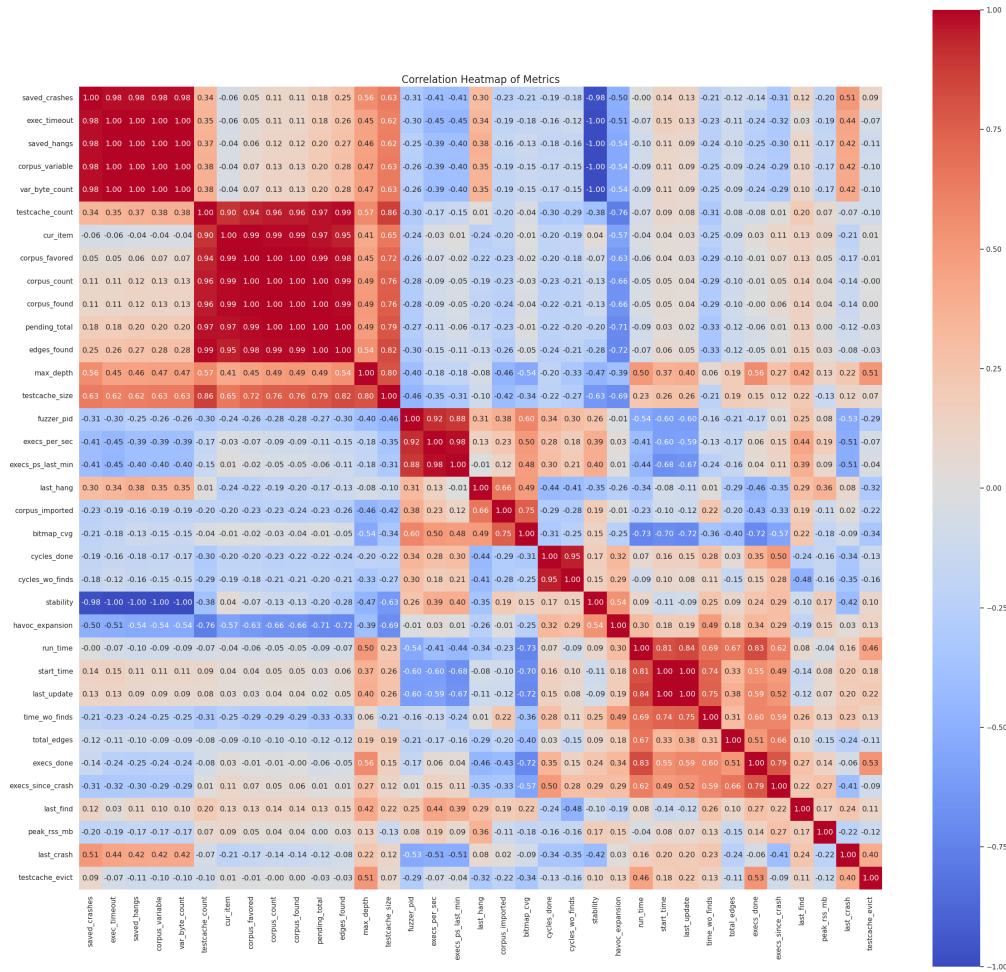


Figure 27: Correlation Between Different Metrics

- **Stability vs Saved Crashes** The correlation between stability and saved crashes is almost -1, indicating a strong inverse relationship. This suggests that higher stability in the fuzzing process results in fewer saved crashes. A highly stable fuzzer might not be exploring the program as aggressively, leading to fewer discovered vulnerabilities.
- **Corpus Count vs Corpus Found** The correlation is almost 1 for corpus count vs corpus found, indicating a nearly perfect positive relationship. This suggests that the number of corpus entries processed is almost equal to the number of corpus entries found. Essentially, it confirms that all or nearly all corpus inputs are being processed efficiently. This consistency is crucial for ensuring that every input is adequately tested.
- **Edges Found vs Corpus Found** There is a strong correlation of almost 1 between edges found and corpus found. This indicates that as more corpus inputs are processed, a proportional increase in unique execution paths (edges) is discovered. This strong relationship underscores the importance of a diverse corpus in uncovering different execution paths in the target program.
- **Edges Found vs Bitmap Coverage** There is no significant correlation (-0.05) between edges found and bitmap coverage. This lack of correlation suggests that finding new execution paths does not necessarily translate to higher code coverage. It implies that while new paths are being discovered, they might not be significantly contributing to exercising new code regions.

- **Runtime vs Bitmap Coverage** There is almost no correlation between runtime and bitmap coverage. This indicates that the duration of fuzzing does not strongly affect the extent of code coverage achieved. It suggests that merely running the fuzzer for a longer time might not yield better coverage, emphasizing the need for effective test cases and strategies to enhance coverage.
- **Max Depth Trend** This graph shows the maximum depth of code paths explored over time. Increasing max depth over time is good, indicating that the fuzzer is exploring deeper into the code, but no clear correlation can be found. 28

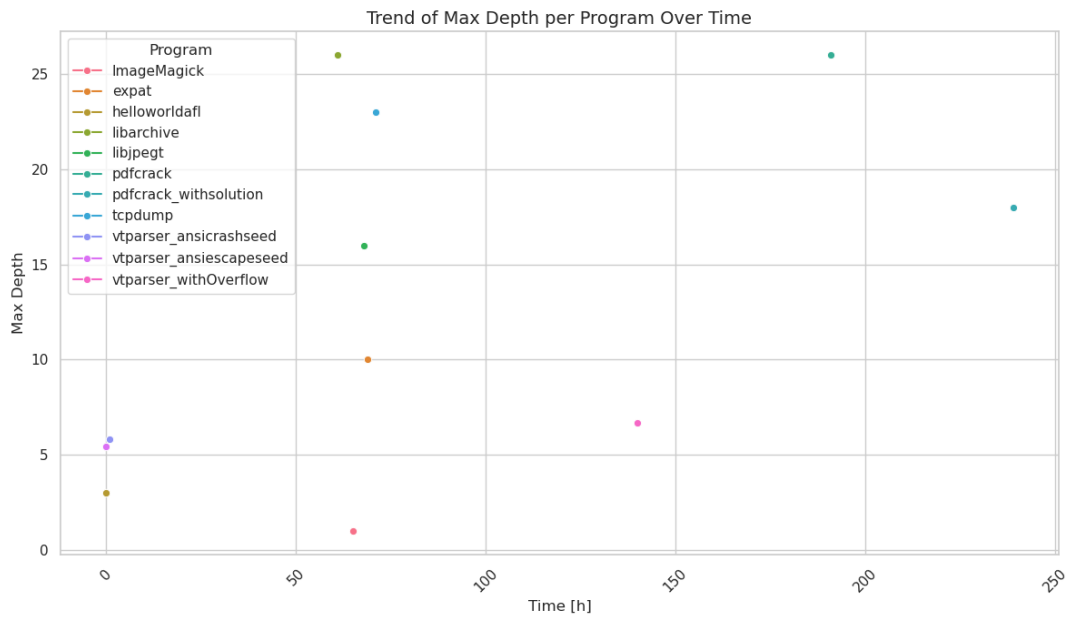


Figure 28: Trend of Maximum Depth Explored

### 6.12.2 Termination conditions

Some useful termination conditions for the fuzzing process include:

- **Time-Based:** Stopping the fuzzing process after a predefined amount of time, such as hours or days.
- **Cycle-Based:** Halting after a certain number of fuzzing cycles are completed.
- **Crash Discovery:** Ending the fuzzing session after a specific number of unique crashes or bugs have been found.
- **Code Coverage:** Terminating when a certain percentage of code coverage is achieved.
- **Performance Plateau:** Stopping when no new significant crashes or code paths are discovered over a set period (e.g., no new crashes found in the last 24 hours).

It is often difficult to see significant correlations between different metrics and the outcomes. Therefore, in a continuous integration/continuous deployment (CI/CD) environment, it is essential to account for previous metrics and compare them over time to optimize the fuzzing process.

1. Practical Guidelines for Termination Conditions Considering the variability across different projects, here are some generalized guidelines that could apply more broadly:

**Time-Based:**

- **Initial Assessment:** For an initial assessment, run the fuzzer for at least 24-48 hours to gather baseline data.
- **Routine Testing:** For ongoing tests in a CI/CD environment, use shorter periods such as 6-12 hours, adjusted based on historical data.

#### Cycle-Based:

- **Baseline Data:** Conduct initial runs to determine an average cycle count where most vulnerabilities are found.
- **Adaptive Termination:** Use adaptive limits based on the complexity of the project. For example, set a termination point at twice the average number of cycles where the last vulnerability was found.

#### Crash Discovery:

- **Initial Runs:** During initial runs, set a high threshold (e.g., 100 crashes) to understand the crash frequency.
- **Adaptive Limit:** Gradually lower this threshold based on historical data to a point where the discovery of new unique crashes diminishes (e.g., 10-20 crashes for routine runs).

#### Code Coverage:

- **Initial Benchmark:** Aim for a high coverage (e.g., 90%) during initial assessments to establish a comprehensive baseline.
- **Routine Runs:** Set a more achievable / Pareto-optimal target for regular CI/CD runs, adjusting based on past results.

#### Performance Plateau:

- **Dynamic Adjustment:** Implement a dynamic threshold where the fuzzing process stops if no new crashes or code paths are discovered over a period that is 10-20% of the total run time. For example, if running for 24 hours, stop if no new results are seen in the last 2-4 hours.

These guidelines provide a more flexible framework that can be adjusted based on the specific characteristics and historical data of each project.

### 6.12.3 Limitations of AFL++

There may be specific projects where AFL++ does not find any vulnerabilities. This can be due to limitations in the tool itself, the nature of the code being tested, or the parameters and configurations used during the fuzzing process.

#### 1. Identified Limitations for Specific Projects:

- **libarchive:** The I/O-bound nature and complex file structures may not be fully covered by AFL++, potentially missing logical bugs.
- **ImageMagick:** Diverse file formats and performance overhead due to sanitizers can limit AFL++'s effectiveness in testing all paths and catching time-sensitive bugs.

#### 2. Strengths of AFL++ for Specific Projects:

- **libarchive:**

- **Memory Safety:** AFL++ is particularly good at finding memory safety issues, such as buffer overflows and use-after-free errors, which are common in file parsing libraries like libarchive.
- **Crash Discovery:** AFL++’s ability to discover crashes quickly makes it effective for identifying critical vulnerabilities early in the testing process.
- **ImageMagick:**
  - **Broad Coverage:** AFL++ can be configured to test a wide range of image formats, providing comprehensive coverage of the different functionalities in ImageMagick.
  - **Speed:** AFL++’s high execution speed allows for extensive testing within a short period, making it suitable for CI/CD pipelines where time is critical.

### 3. Suggested Fuzzers for Other Projects:

- **Network Protocol Implementations (e.g., OpenSSL):**
  - **Suggested Fuzzer:** libFuzzer or Honggfuzz
  - **Reason:** Better suited for in-memory fuzzing and handling protocol state machines.
- **Complex Parsers (e.g., XML Parsers):**
  - **Suggested Fuzzer:** libFuzzer
  - **Reason:** Efficient for targeting specific functions with deep input processing logic.
- **Web Browsers (e.g., Chromium):**
  - **Suggested Fuzzer:** ClusterFuzz
  - **Reason:** Designed for large-scale fuzzing with the ability to handle complex targets like browsers.
- **Kernel Drivers:**
  - **Suggested Fuzzer:** Syzkaller
  - **Reason:** Specifically designed for fuzzing kernel code, requiring handling of syscalls and kernel state.

#### 6.12.4 Sample Penetration Test Report for vt-parser

When contracting an external company, what would be the penetration test report? A sample project report template might include:

- **Chosen Fuzzer**
- **Approach Tried**
- **Fuzzer Run Results**
- **Findings:** Any vulnerabilities discovered or no vulnerabilities found.
- **Further Actions:** Suggestions like trying another fuzzing tool, running for a longer period, or adjusting parameters.
- **Types of Vulnerabilities Tested:** Explained in terms understandable to non-programmers, focusing on C++ vulnerabilities.

and could look like Appendix Sample Penetration Test Report for vt-parser



## 7 Conclusion

In this thesis, we systematically explored the identification of vulnerabilities in C and C++ source code using AFL++ (American Fuzzy Lop Plus Plus), focusing heavily on the implementation and analysis of fuzzing techniques. Our primary objective was to enhance the security and reliability of software by employing AFL++ to uncover potential security flaws that might be overlooked by conventional testing methods.

### 7.1 Achievements Compared to Initial Objectives

The primary goal of this thesis was to establish a performant fuzzing environment for C++ programs, compare performance parameters, and examine productive software for vulnerabilities. The key objectives outlined in the initial problem description and how they were achieved are summarized below:

#### Setup of a Fuzzing Environment:

- The environment was successfully established using AFL++.
- ClusterFuzz was evaluated for setting up a local fuzzing environment.

#### Target Program Selection and Harness Creation:

- Various open-source C++ projects were selected for fuzzing.
- Fuzzing harnesses were created to test these projects, focusing on areas with significant user interaction to maximize the chances of uncovering vulnerabilities.

#### Performance Comparison:

- Different fuzzing parameters, including input types and coverage-guided strategies, were compared.
- The metrics collected during fuzzing runs (code coverage, number of crashes, duration) provided insights into the performance and effectiveness of the fuzzing setup.

#### Fuzzing of Software Projects:

- The fuzzing process was extended to various terminal applications and other software projects, demonstrating the versatility and applicability of AFL++ in different contexts.

#### Documentation and Analysis:

- Detailed documentation of the fuzzing environment and the analysis of found crash conditions were provided.
- Although no new vulnerabilities were discovered, the known vulnerabilities identified validated the effectiveness of the fuzzing process.

### 7.2 Implications for Software Security

The insights gained from our implementation and analysis have several important implications for enhancing software security:

- **CI/CD Integration:** Integrating AFL++ into CI/CD pipelines could automate continuous security testing. This approach ensures that vulnerabilities are detected and addressed promptly, reducing the risk of security breaches in production environments.

- **Metrics for Improvement:** The collected metrics serve as a valuable baseline for further optimization of the fuzzing process. These metrics can guide the development of more efficient fuzzing strategies and improve the overall security posture of software projects.
- **Proof of Concept Validation:** The successful demonstration of known vulnerability detection validates AFL++ as a critical component of modern security testing frameworks, underscoring its importance in identifying and mitigating security risks.

### 7.3 Future Work

While this research has laid a solid foundation, several areas warrant further exploration to enhance the effectiveness and efficiency of fuzzing with AFL++:

- **Advanced Fuzzing Algorithms:** Future research should focus on developing advanced fuzzing algorithms that can handle more complex software structures and dependencies. Incorporating machine learning techniques could generate more sophisticated test inputs and improve fuzzing efficiency.
- **Broader Application:** Applying AFL++ to a wider range of real-world applications, including IoT devices and embedded systems, would provide a more comprehensive assessment of its effectiveness in diverse environments.
- **Integration with Other Security Tools:** Combining AFL++ with other security testing methodologies, such as static analysis and symbolic execution, could offer a more robust and comprehensive vulnerability detection framework.

In conclusion, the systematic implementation and analysis of AFL++ for fuzzing C and C++ source code have proven to be effective in identifying security vulnerabilities. Although no new vulnerabilities were discovered, our research successfully validated the proof of concept and collected valuable metrics that can be leveraged for CI/CD integrations. This work underscores the importance of adopting continuous and automated security testing practices to safeguard software applications against emerging threats. By addressing the challenges and leveraging the strengths of AFL++, developers can significantly enhance the security and robustness of their software systems.

### 7.4 Self-Reflection

Working on this thesis has been a challenging yet rewarding experience. Delving into C++ code to identify bugs and vulnerabilities was both difficult and educational. Setting up AFL++ for fuzzing required a lot of effort and problem-solving.

Despite the hard work, it was a bit discouraging not to find any new bugs. However, the project reinforced my understanding of fuzzing techniques and software security. I learned a lot about managing complex dependencies and the importance of thorough testing.

This experience has significantly broadened my knowledge of C++ and software security. It has been a valuable learning journey, equipping me with skills that will be useful in my future endeavors.

## 8 Glossary

- **AFL (American Fuzzy Lop)**: A fuzzer that uses genetic algorithms to find bugs by generating and testing lots of inputs.
- **AFL++**: An improved version of AFL with extra features for better fuzzing.
- **Fuzzer**: A tool that generates many inputs for testing a program to find bugs.
- **Fuzzing**: A testing technique that sends random data to a program to find bugs and vulnerabilities.
- **Fuzzing Harness**: A framework that lets the fuzzer interact with the software being tested.
- **Parallel Processing**: Running multiple tasks at the same time to save time and increase efficiency.
- **Docker**: A platform for running applications in containers, which are isolated environments with all needed files.
- **Crash File**: An input file that causes a program to crash, used to identify bugs.
- **tmin (Testcase Minimizer)**: A tool that makes crash files smaller while keeping the bug reproducible.
- **Vulnerability**: A weakness in software that can be exploited to cause unintended behavior.
- **Core Dump**: A file that shows the memory state of a program when it crashes, used for debugging.
- **Seed Corpus**: A starting set of test cases used for fuzzing.
- **Sanitizers**: Tools that find various kinds of bugs like memory errors and undefined behavior.
- **Stack Overflow**: An error when too much memory is used on the call stack.
- **Heap Overflow**: An error where a program writes more data to a memory block than allocated.
- **Use-After-Free**: An error where a program uses memory after it has been freed.
- **Double-Free**: An error where a program frees the same memory location twice.
- **Undefined Behavior**: Program actions that have unpredictable results, often leading to bugs.
- **Static Analysis**: Examining code without running it to find bugs.
- **Dynamic Analysis**: Examining code while it runs to find bugs.
- **Exploit**: Code that takes advantage of a bug to cause unintended behavior.
- **Patch**: Code updates that fix or improve a program.
- **Regression Testing**: Testing to ensure that recent changes don't break existing features.
- **CI/CD (Continuous Integration/Continuous Deployment)**: Automating the process of integrating and deploying code frequently.

- **Bug Triage:** Reviewing and prioritizing bug reports to decide which to fix first.
- **Reproducibility:** The ability to consistently reproduce a bug.
- **Black-Box Testing:** Testing the functionality of an application without knowing its internal workings.
- **White-Box Testing:** Testing the internal workings of an application.
- **Gray-Box Testing:** A mix of black-box and white-box testing.
- **Input Sanitization:** Cleaning user inputs to prevent malicious data from causing harm.
- **Boundary Value Analysis:** Testing at the edges of input ranges.
- **Crash Reproduction:** Reliably reproducing a software crash.
- **Debugging:** Finding and fixing bugs in a program.
- **Code Instrumentation:** Adding extra code to monitor program behavior.
- **Memory Leak:** When a program incorrectly manages memory allocations.
- **Stack Trace:** A report of the active stack frames at a certain point during program execution.
- **Buffer Overflow:** When a program writes data beyond the boundaries of pre-allocated buffers.
- **Input Validation:** Ensuring that a program operates on clean and correct data.
- **Unit Test:** Testing individual parts of a program.
- **Integration Test:** Testing combined parts of a program to ensure they work together.

## 9 References

### 9.1 Bibliography

- [1] OWASP. *Fuzzing*. Accessed: 2024-06-01. 2024. URL: <https://owasp.org/www-community/Fuzzing>.
- [2] GNOME. *GNOME GLib Project - Fuzzing Integration*. Accessed: 2024-04-14. 2024. URL: <https://github.com/GNOME/glib/tree/main/fuzzing>.
- [3] Google. *Announcing OSS-Fuzz: Continuous Fuzzing for Open Source Software*. Accessed: 2024-04-14. 2016. URL: <https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html>.
- [4] Google. *OSS-Fuzz Overview*. Accessed: 2024-04-14. 2024. URL: <https://github.com/google/oss-fuzz>.
- [5] Black Hat. *Black Hat Archives*. Accessed: 2024-04-14. 2024. URL: <https://www.blackhat.com/html/archives.html>.
- [6] DEF CON. *DEF CON Archives*. Accessed: 2024-04-14. 2024. URL: <https://defcon.org/html/links/dc-archives.html>.
- [7] Google. *AFL GitHub Repository*. Accessed: 2024-04-14. 2024. URL: <https://github.com/google/AFL>.
- [8] LLVM. *LibFuzzer GitHub Repository*. Accessed: 2024-04-14. 2024. URL: <https://github.com/llvm/llvm-project/tree/main/compiler-rt/lib/fuzzer>.
- [9] ReadWrite. “The Heartbleed Bug: What it is and how to fix it”. In: (). Accessed: 2024-05-02. URL: <https://readwrite.com/heartbleed-bug/>.
- [10] Zalewski, Michal. “AFL Fuzzing Impact on Open Source Projects”. In: (2023). Accessed: 2024-06-02. URL: <https://lcamtuf.coredump.cx/afl/>.
- [11] Adith Sudhakar, Mohit Arora, and Souheil Moghnie. *Focus on Fuzzing: Fuzzing Within the SDLC*. Accessed: 2024-06-02. 2020. URL: <https://safecode.org/blog/focus-on-fuzzing-fuzzing-within-the-sdlc/>.
- [12] NSA. *NSA Releases Guidance on How to Protect Against Software Memory Safety Issues*. 2023. URL: <https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3215760/nsa-releases-guidance-on-how-to-protect-against-software-memory-safety-issues/>.
- [13] MITRE. *2023 CWE Top 25 Most Dangerous Software Weaknesses*. 2023. URL: [https://cwe.mitre.org/top25/archive/2023/2023\\_kev\\_list.html](https://cwe.mitre.org/top25/archive/2023/2023_kev_list.html).
- [14] TIOBE. *TIOBE Programming Community Index for June 2023*. 2023. URL: <https://www.tiobe.com/tiobe-index/>.
- [15] Andreas Zeller et al. *The Fuzzing Book*. 2021. URL: <https://www.fuzzingbook.org/>.
- [16] aflplusplus. *aflplusplus*. Accessed: 2024-04-01. 2024. URL: <https://github.com/AFLplusplus/AFLplusplus>.
- [17] *libFuzzer and LLVM*. Accessed: 2024-06-03. URL: <https://llvm.org/docs/LibFuzzer.html>.
- [18] *Honggfuzz CI*. Accessed: 2024-06-03. URL: <https://github.com/google/honggfuzz>.
- [19] *Syzkaller*. Accessed: 2024-06-03. URL: <https://github.com/google/syzkaller>.
- [20] clusterfuzz. *clusterfuzz*. Accessed: 2024-04-01. 2024. URL: <https://google.github.io/clusterfuzz/>.

- [21] aflplusplus. *aflplusplusInDepth*. Accessed: 2024-04-01. 2024. URL: [https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/fuzzing\\_in\\_depth.md](https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/fuzzing_in_depth.md).
- [22] aflplusplus. *aflplusplusdocs*. Accessed: 2024-04-01. 2024. URL: <https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/README.md>.
- [23] libfuzzer. *libfuzzer*. Accessed: 2024-04-01. 2024. URL: <https://llvm.org/docs/LibFuzzer.html>.
- [24] libfuzzerdocs. *libfuzzerdocs*. Accessed: 2024-04-01. 2024. URL: <https://llvm.org/docs/LibFuzzer.html#overview>.
- [25] libfuzzerasan. *libfuzzerasan*. Accessed: 2024-04-01. 2024. URL: <https://llvm.org/docs/LibFuzzer.html#using-address-sanitizer>.
- [26] hongfuzz. *hongfuzz*. Accessed: 2024-04-01. 2024. URL: <https://github.com/google/honggfuzz>.
- [27] aflplusplus. *aflplusplusTuts*. Accessed: 2024-04-01. 2024. URL: <https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/tutorials.md>.
- [28] aflplusplus. *aflpluspluslibxml2*. Accessed: 2024-04-01. 2024. URL: [https://aflplusplus.com/docs/tutorials/libxml2\\_tutorial/](https://aflplusplus.com/docs/tutorials/libxml2_tutorial/).
- [29] contour-terminal. *contour-terminal*. Accessed: 2024-04-04. 2024. URL: <https://github.com/contour-terminal>.
- [30] nasa-hdtn. *nasa-hdtn*. Accessed: 2024-04-04. 2024. URL: <https://github.com/nasa/HDTN.git>.
- [31] nasa-hdtn-user-guide. *nasa-hdtn-user-guide*. Accessed: 2024-04-04. 2024. URL: [https://ntrs.nasa.gov/api/citations/20230015434/downloads/HDTN\\_Users\\_Guide\\_AfterEditor2.pdf](https://ntrs.nasa.gov/api/citations/20230015434/downloads/HDTN_Users_Guide_AfterEditor2.pdf).
- [32] libBLS. *libBLS*. Accessed: 2024-04-04. 2024. URL: <https://github.com/skalenetwork/libBLS>.
- [33] libff. *libff*. Accessed: 2024-04-04. 2024. URL: <https://github.com/scipr-lab/libff>.
- [34] tcpdump. *tcpdump*. Accessed: 2024-04-13. 2024. URL: <https://github.com/the-tcpdump-group/tcpdump>.
- [35] libjpeg-turbo. *libjpeg-turbo*. Accessed: 2024-04-12. 2024. URL: <https://github.com/libjpeg-turbo/libjpeg-turbo>.
- [36] libexpat. *libexpat*. Accessed: 2024-04-14. 2024. URL: <https://github.com/libexpat/libexpat>.
- [37] pdfcrack. *pdfcrack*. Accessed: 2024-04-14. 2024. URL: <https://sourceforge.net/projects/pdfcrack/files/pdfcrack/>.
- [38] Charlotte Brontë. *Jane Eyre*. Accessed: 2024-04-14. 2024. URL: [https://ia601800.us.archive.org/12/items/janeeyre0000bron\\_q6s0/janeeyre0000bron\\_q6s0.pdf](https://ia601800.us.archive.org/12/items/janeeyre0000bron_q6s0/janeeyre0000bron_q6s0.pdf).
- [39] Ivan Turgenev. *First Love and Other Stories*. Accessed: 2024-04-14. 2024. URL: <https://ia601309.us.archive.org/21/items/firstloveandoth00turggoog/firstloveandoth00turggoog.pdf>.
- [40] W. Somerset Maugham. *The Merry-Go-Round*. Accessed: 2024-04-14. 2024. URL: <https://ia803106.us.archive.org/6/items/merrygoround00well/merrygoround00well.pdf>.
- [41] youtube. *pdfcrack Tutorial*. Accessed: 2024-04-14. 2024. URL: <https://www.youtube.com/watch?v=8VLNPIIgKbQ>.
- [42] libarchive. *libarchive*. Accessed: 2024-04-14. 2024. URL: <http://www.libarchive.org/downloads/libarchive-3.1.2.tar.gz>.

- [43] ImageMagick. *ImageMagick*. Accessed: 2024-04-14. 2024. URL: <https://github.com/ImageMagick/ImageMagick/archive/refs/tags/7.1.1-4.tar.gz>.
- [44] ImageMagick. *ImageMagick Github*. Accessed: 2024-04-14. 2024. URL: <https://github.com/ImageMagick/ImageMagick>.
- [45] ImageMagick. *ImageMagick Github Commit of Bugfix*. Accessed: 2024-04-14. 2024. URL: <https://github.com/ImageMagick/ImageMagick/commit/d7a8bdd7bb33cf8e58bc01b4a4f2ea5466f8c6>
- [46] agilehunt. *CVE-2023-1906*. Accessed: 2024-04-14. 2024. URL: <https://blog.agilehunt.com/blogs/security/cve-2023-1906-heap-based-buffer-overflow-in-imagemagick>.

## 9.2 Figures

1	Design and Architecture Overview . . . . .	16
2	Container Diagram for Fuzzing Framework . . . . .	17
3	Component Diagram for Fuzzing Framework . . . . .	18
4	Integration Framework . . . . .	18
5	Fuzzing Engine Process Diagram . . . . .	19
6	Screenshot of Crashfiles . . . . .	23
7	Screenshot of GDB with Stacktrace . . . . .	23
8	Screenshot of AFL++ with htop. . . . .	25
9	HDTN Software Architecture (NASA User Guide). . . . .	27
10	Screenshot of AFL++. . . . .	28
11	Screenshot of AFL++. . . . .	29
12	Screenshot of AFL++. . . . .	29
13	TCPDump-Edges . . . . .	30
14	TCPDump-ExecSpeed . . . . .	30
15	TCPDump-HighFrequency . . . . .	30
16	TCPDump-LowFrequency . . . . .	31
17	Screenshot of GDB . . . . .	32
18	Screenshot of GDB Breakpoint . . . . .	33
19	Screenshot of GDB Traces . . . . .	33
20	Screenshot of AFL . . . . .	39
21	Execution Speed in Execs Per Second . . . . .	40
22	Code Coverage in Bitmap Coverage . . . . .	40
23	Code Coverage vs Runtime . . . . .	42
24	Distribution of Run Times . . . . .	42
25	Distribution of Edges Found . . . . .	43
26	Distribution of Bitmap Coverage . . . . .	43
27	Correlation Between Different Metrics . . . . .	44
28	Trend of Maximum Depth Explored . . . . .	45

# 10 Appendices

## 10.1 Appendix Task Definition

# Aufgabenstellung

## Titel der Arbeit

Performantes quellenbasiertes Fuzzing zur Detektion von Sicherheitslücken in C++ Programmen

## Problembeschrieb

Etablierte Programmiersprachen wie C++ sind nach wie vor anfällig für subtile Schwachstellen  
↪ und Bugs wie  
Buffer Overflows, Integer Overflows und Race Conditions. Die Zahl der daraus resultierenden  
↪ Sicherheitslücken  
in Produktivsoftware ist weiterhin hoch. Die MITRE Organisation listet in den Top 10  
↪ Softwareschwachstellen  
des Jahres 2023 die ersten drei Schwachstellen als "Use After Free", "Heap-based Buffer  
↪ Overflow" und "Out-of-bounds  
Write", allesamt Implementierungsschwachstellen der zugrunde liegenden Sprache. [1] Derartige  
↪ Schwachstellen sind  
typisch für Programmiersprachen, die nicht memory-safe sind. Die NSA rät inzwischen explizit  
↪ von der Verwendung von  
C++ ab. [2] Nichtsdestotrotz ist C++ eine der beliebtesten Programmiersprachen laut TIOBE  
↪ Programming Community Index [3]

Quellenbasiertes Fuzzing ist eine etablierte Methode zur Detektion derartiger Sicherheitslücken  
↪ in C++ Code.

Ziel der Arbeit ist es, eine Umgebung für performantes Fuzzing von C++ Programmen aufzubauen,  
↪ Performanceparameter  
zu vergleichen und produktive Softwareprogramme, z.B. open-source coding in C++ damit auf  
↪ Sicherheitslücken zu überprüfen.

[1] [https://cwe.mitre.org/top25/archive/2023/2023\\_kev\\_list.html](https://cwe.mitre.org/top25/archive/2023/2023_kev_list.html)

[2] <https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3215760/nsa-releases-guidanc>  
↪ e-on-how-to-protect-against-software-memory-safety-issues/

[3] <https://www.tiobe.com/tiobe-index/>

## Formulierung eines konkreten Auftrags

Ziel der Arbeit ist der Aufbau einer performanten Fuzzingumgebung für C++ Projekte.

↪ Teilaufgaben bestehen aus:

- Einarbeiten in quellenbasiertes Fuzzing mit AFL++, z.B. mit bpftrace, ipdecap, libxml2,  
↪ Tutorial im OST Hacking Lab
- Prüfen von clusterfuzz zum Aufbau einer lokalen Fuzzingumgebung im INS Labor
- Prüfen von Zielprogrammen zum Fuzzing, z.B. Terminals,  
↪ <https://github.com/microsoft/terminal>
- Vergleich unterschiedlicher Fuzzingparameter (Input, ggf. blackbox vs. whitebox) in  
↪ Bezug auf die Fuzzingperformance (Codeabdeckung, Zyklen, Dauer, gefundene  
↪ Crashbedingungen)
- Fuzzing von verschiedenen Terminalapplikationen, Vergleich der Resultate
- Fuzzing weiterer Softwareprojekte, ggf. mit Partnerunternehmen des INS Teams
- OPTIONAL: Prüfen alternativer "Crashbedingungen"
- Dokumentation und Analyse gefundener Crashbedingungen in C++ (optional: responsible  
↪ vulnerability disclosure process)
- Dokumentation der Fuzzingumgebung (z.B. clusterfuzz im INS lab) zur Verwendung durch  
↪ andere Studierende

Aufgrund der ausgeprägten C++ Vorkenntnisse des Studenten ist der Fokus auf quellcode-basiertem  
↪ C++ Fuzzing.



Ressourcen zur Einarbeitung und Vertiefung sind

### ### Fuzzing: Einführung und Wissen

```
https://www.fuzzingbook.org/
https://www.fuzzingbook.org/html/ConfigurationFuzzer.html#Exercise-3:-Extracting-and-Fuzzin
↳ g-C-Command-Line-Options
https://aflplus.plus/docs/tutorials/
https://github.com/google/clusterfuzz
https://google.github.io/clusterfuzz/
https://google.github.io/clusterfuzz/setting-up-fuzzing/heartbleed-example/
https://www.cs.ru.nl/~erikpoll/talks/Fuzzing101.pdf
https://lcamtuf.blogspot.com/2014/09/quick-notes-about-bash-bug-its-impact.html
https://lcamtuf.blogspot.com/2014/10/bash-bug-how-we-finally-cracked.html
```

### ### Vulnerable Testprogramme

```
https://samate.nist.gov/SARD
```

### ## Umfang und Form der erwarteten Resultate bei Abgabe der Arbeit, wenn abweichend von Punkt 5.5

Die erwarteten Resultate sind wie in Abschnitt 5.5 "Leitfaden für Bachelor- und  
↳ Studienarbeiten" einzureichen.

### ## Beteiligte Personen (Studierende, Betreuungsperson, weitere Beteiligte soweit bekannt, ↳ namentlich Industriepartner)

Studierende: Miles Strässle  
Betreuungspersonen: Nikolaus Heners  
Industriepartner: optional im Verlauf des Projekts

### ## Anfangs- und Abgabetermin

### ## Gewichtung der einzelnen Bewertungsteile (siehe Punkte 6.4 und 6.5)

1. Organisation und Durchführung: 10%
2. Formale Qualität des Berichts: 10%
3. Analyse, Entwurf und Auswertung: 20%
4. Technische Umsetzung: 40%
5. Bachelorprüfung: 20%

### ## Zulässige Hilfsmittel und weitere Betreuung (z.B. durch externen Partner)

Die Verwendung von Large Language Models (ChatGPT,...) in der Studienarbeit muss angegeben  
↳ werden.

Mögliche Partnerschaften mit Industriepartnern können sich im Laufe des Projekts ergeben.

## 10.2 Appendix contour

### 10.2.1 Threaded Fuzzing

```
#!/bin/bash
NUM_CORES=$(nproc)
MASTER_CORE=$((NUM_CORES - 1))
AFL_CMD="/AFLplusplus/afl-fuzz"
INPUT_DIR="/src/fuzzing_projects/contourOriginal/contour/seeds"
OUTPUT_DIR="out"
TARGET="/src/fuzzing_projects/contourOriginal/contour/build/linux-release/src/vtparser/vtparser"
↳ _fuzz"

# Start master instance
$AFL_CMD -i $INPUT_DIR -o $OUTPUT_DIR -M master -m none -d -- $TARGET &
```

```

# Start slave instances
for ((i=1; i<NUM_CORES; i++)); do
    SLAVE_NAME="slave$i"
    $AFL_CMD -i $INPUT_DIR -o $OUTPUT_DIR -S $SLAVE_NAME -m none -d -- $TARGET 2>>1 &
done

echo "Started AFL++ with 1 master and $((NUM_CORES - 1)) slave instances."

```

## 10.2.2 Crash Analysis

```

# Analysis crash-files similar to this:
#
# ID: id:000001, Cause: heap-buffer-overflow, Location: scan.cpp:265:38 in unicode
# ==8034==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60300000003e at pc
↳ 0x5633cbe768b5 bp 0x7ffed1f83e10 sp 0x7ffed1f83e08
# READ of size 1 at 0x60300000003e thread T0
#   #0 0x5633cbe768b4 in unicode::scan_text(unicode::scan_state&,
↳ std::basic_string_view<char, std::char_traits<char>>, unsigned long,
↳ unicode::grapheme_cluster_receiver&) /src/fuzzing_projects/contour/_deps/sources/libunicode_
↳ -23d7b30166a914b10526bb8fe7a469a9610c07dc/src/libunicode/scan.cpp:265:38
#   #1 0x5633cbe6bece in unicode::scan_text(unicode::scan_state&,
↳ std::basic_string_view<char, std::char_traits<char>>, unsigned long)
↳ /src/fuzzing_projects/contour/_deps/sources/libunicode-23d7b30166a914b10526bb8fe7a469a9610c_
↳ 07dc/src/libunicode/scan.cpp:226:12
#   #2 0x5633cbe6bece in vtparser::Parser<vtparser::ParserEvents, false>::parseBulkText(char
↳ const*, char const*) /src/fuzzing_projects/contour/src/vtparser/Parser-impl.h:378:48
#   #3 0x5633cbe6b36e in vtparser::Parser<vtparser::ParserEvents,
↳ false>::parseFragment(gsl::span<char const, 18446744073709551615ul>)
↳ /src/fuzzing_projects/contour/src/vtparser/Parser-impl.h:329:56
#   #4 0x5633cbe6b36e in main /src/fuzzing_projects/contour/src/vtparser/main.cpp:47:7
#   #5 0x7eff80cc6d8f (/lib/x86_64-linux-gnu/libc.so.6+0x29d8f) (BuildId:
↳ c289da5071a3399de893d2af81d6a30c62646e1e)
#   #6 0x7eff80cc6e3f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x29e3f)
↳ (BuildId: c289da5071a3399de893d2af81d6a30c62646e1e)
#   #7 0x5633cbd8ff84 in _start
↳ (/src/fuzzing_projects/contour/build/linux-release/src/vtparser/vtparser_fuzz+0x71ef84)
↳ (BuildId: f9195481a04e1845)

```

```

import os
import re

```

```

def parse_crash_details(filepath):
    """
    Parses a file to find the source of the crash.
    """
    cause = ""
    function_line = ""
    with open(filepath, "r") as file:
        content = file.read()
        # Find the cause of the crash from the SUMMARY line
        summary_match = re.search(r"SUMMARY: AddressSanitizer: ([\w-]+)", content)
        if summary_match:
            cause = summary_match.group(1)

        # Find the function and line number of the crash from the SUMMARY line
        function_line_match = re.search(
            r"SUMMARY: AddressSanitizer: [\w-]+ (?!.*/)?([~/]+:\d+:\d+ in \w+)", content
        )
        if function_line_match:
            function_line = function_line_match.group(1)

```

```

    return cause, function_line

def main():
    analysis_dir =
    ↪ "/home/i/TresorMiles/Code/smilesT_github/FH0st/FH0st-6sem/fuzzcpptthesis/fuzzing_project
    ↪ s/_afl_results/vtparser/fuzzer_run_vtparser_withOverflow_all_crashes/analysis"
    for filename in os.listdir(analysis_dir):
        if filename.endswith("_analysis"):
            crash_id = filename.split(",")[0]
            filepath = os.path.join(analysis_dir, filename)
            cause, function_line = parse_crash_details(filepath)
            if cause and function_line:
                print(f"ID: {crash_id}, Cause: {cause}, Location: {function_line}")
            else:
                print(f"ID: {crash_id}, No detailed crash info found.")

main()

```

### 10.2.3 Bash: Helper cmake install

Installs a newer cmake version than apt

```

wget https://github.com/Kitware/CMake/releases/download/v3.29.0-rc2/cmake-3.29.0-rc2-linux-x86_
↪ 64.sh
cd cmake*
chmod +x cmake-3.29.0-rc2-linux-x86_64.sh
./cmake-3.29.0-rc2-linux-x86_64.sh

```

### 10.2.4 Bash: Helper container patch

```

apt update
# Sudo needed for contour dependencies
apt install -y sudo

```

### 10.2.5 Bash: Helper seed minimizer

Minimizes the seeds

```

SEED_DIR="./seeds"
OUTPUT_DIR="./seeds/seed-minimized"
TARGET_BINARY="/src/fuzzing_projects/contourOriginal/contour/build/linux-release/src/vtparser/v
↪ tparser_fuzz"

mkdir -p "$OUTPUT_DIR"

for SEED in "$SEED_DIR"/*; do
    FILENAME=$(basename "$SEED")
    /AFLplusplus/afl-tmin -i "$SEED" -o "$OUTPUT_DIR/${FILENAME}_minimized" -- "$TARGET_BINARY" @@
done

echo "Minimization complete. Minimized seeds are in $OUTPUT_DIR"

```

## 10.3 Appendix libBLS

### 10.3.1 libBLS Harness

```

#include <stdlib.h>
#include <iostream>
#include <string>

```

```

#include <array>
#include <cstdio>
#include <fstream>
#include <iostream>
#include <memory>
#include <stdexcept>
#include <string>

// #include "dkg_key_gen.cpp"
namespace Harness {

inline int t{};
inline int n{};
inline std::string root_dir{"src/fuzzing_projects/libBLSFuzz/build"};
inline std::string build_dir{root_dir + "../../libBLS/build/"};
inline std::string message_path{root_dir + "../../message/data.in"};
inline std::string signature_path{root_dir + "/signature.json"};

// Function to execute a shell command
inline auto executeCommand(const std::string& command) -> bool {
    std::string cmd = Harness::build_dir + command;
    std::cout << "Executing: " << cmd << std::endl;
    int ret = system(cmd.c_str());
    return ret == 0;
}

// Execute a shell command and optionally capture the output
inline auto executeCommandWithOutput(const std::string& command) -> std::string {
    std::array<char, 64> buffer;
    std::string result;
    std::unique_ptr<FILE, decltype(&pclose)> pipe(popen(command.c_str(), "r"),
                                                pclose);

    if (!pipe) {
        throw std::runtime_error("popen() failed!");
    }
    while (fgets(buffer.data(), buffer.size(), pipe.get()) != nullptr) {
        result += buffer.data();
    }
    return result;
}

inline auto hashMessage(std::string& msg) -> std::string {
    // Write the message to a temporary file
    std::string tempFilePath = "/tmp/message_to_hash.txt";
    std::ofstream tempFile(tempFilePath);
    if (!tempFile.is_open()) {
        throw std::runtime_error(
            "Failed to open temporary file for writing message.");
    }
    tempFile << msg;
    tempFile.close();

    // Hash the temporary file
    std::string command =
        "openssl dgst -sha256 " + tempFilePath + " | awk '{print $2}'";
    std::string hash = executeCommandWithOutput(command);

    // Cleanup temporary file
    std::remove(tempFilePath.c_str());

    return hash;
}
}

```

```

inline auto hashMessageAndWriteToFile(std::string& msg) -> void {
    std::string hash = Harness::hashMessage(msg);

    // Write the hash to data.in
    std::ofstream outputFile(Harness::message_path);
    if (!outputFile.is_open()) {
        throw std::runtime_error("Failed to open output file for writing hash.");
    }
    outputFile << hash;
    outputFile.close();
}

// Generating private keys
inline auto generatePrivateKeys() -> bool {
    std::string command =
        "./dkg_keygen --t " + std::to_string(Harness::t) + " --n " + std::to_string(Harness::n);
    return Harness::executeCommand(command);
}

// generate a common signature
inline auto signMessageWithAll() -> bool {
    std::string command = "./sign_bls --t " + std::to_string(Harness::t) + " --n " +
        std::to_string(Harness::n) + " --key " + Harness::root_dir +
        "/BLS_keys "
        " --input " +
        Harness::message_path + " --output " + "signature.json";
    return Harness::executeCommand(command);
}

// Verifying the signature remains unchanged
inline auto verifySignature() -> bool {
    std::string command = "./verify_bls --t " + std::to_string(Harness::t) + " --n " +
        std::to_string(Harness::n) + " --input " + Harness::signature_path;
    return Harness::executeCommand(command);
}

// sign individually and combine those signatures
// UNTESTED
inline auto signIndividually() -> bool {
    bool ret{true};

    for (int j = 0; j < Harness::n; ++j) {
        ret &= Harness::executeCommand("./sign_bls --t " + std::to_string(Harness::t) + " --n " +
            std::to_string(Harness::n) + " --j " + std::to_string(j) +
            " --key " + Harness::root_dir + "/" + std::to_string(j) +
            " --input " + Harness::message_path + " --output " +
            Harness::signature_path + "/signature_from_" +
            std::to_string(j) + "th_participant.json");
    }
    ret &= Harness::executeCommand("./bls_glue --t " + std::to_string(Harness::t) + " --n " +
        std::to_string(Harness::n) + " --input " + Harness::signature_path +
        "/signature_from_*.json --output " + Harness::root_dir);

    return ret;
}

// Generating all keys using generate_key_system
// UNTESTED
inline auto generateAllKeys(const std::string& output = "") -> bool {
    std::string command = "./generate_key_system --t " + std::to_string(Harness::t) +
        " --n " + std::to_string(Harness::n);
    if (!output.empty()) {

```

```

    command += " --output " + output;
}
return Harness::executeCommand(command);
}

};
// namespace Harness

auto main(int argc, char** argv) -> int {
    std::string str;
    getline(std::cin, str);

    Harness::t = 2;
    Harness::n = 3;

    Harness::hashMessageAndWriteToFile(str);
    std::cout << "hashMessageAndWriteToFile" << std::endl;

    if (Harness::generatePrivateKeys()) {
        std::cout << "generatePrivateKeys" << std::endl;
    } else {
        std::abort();
    }

    if (Harness::signMessageWithAll()) {
        std::cout << "signMessage" << std::endl;
    } else {
        std::abort();
    }

    if (Harness::verifySignature()) {
        std::cout << "verifySignature" << std::endl;
    } else {
        std::abort();
    }

}
}

```

### 10.3.2 File Patch

```

diff --git a/dkg/dkg.h b/dkg/dkg.h
index 43de8ee..57b6e4b 100644
--- a/dkg/dkg.h
+++ b/dkg/dkg.h
@@ -27,8 +27,7 @@
    #include <vector>

    #include <libff/algebra/curves/alt_bn128/alt_bn128_pp.hpp>
-#include <libff/algebra/fields/fp.hpp>
-
+#include <libff/algebra/fields/prime_base/fp.hpp>
    namespace libBLS {

    class Dkg {

```

### 10.3.3 Build Process

```

#!/bin/bash

# Build dependencies libff first
apt update

```

```

apt install build-essential git libboost-all-dev cmake libgmp3-dev libssl-dev pkg-config
↪ libsodium-dev

# ATTENTION! Do this on the host where you cloned the repo
# git submodule init && git submodule update

cd libff

# Build
mkdir build && cd build
cmake ..
make
make install

# Check
make check

# Build libBLS (Container)
# Enable Sanitizers
cd ..
CURRENT_DIR=$(pwd)
CMAKE_LISTS_FILE="$CURRENT_DIR/libBLS/CMakeLists.txt"
sed -i '/set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS}/s/"$/ -fsanitize=address -g/'
↪ "$CMAKE_LISTS_FILE"

# Ignore incompatible compiler flags
sed -i 's/=maybe-uninitialized/=uninitialized/g' "$CMAKE_LISTS_FILE"

# Dont make warnings to errors
sed -i 's/-Werror //g' "$CMAKE_LISTS_FILE"

cd libBLS

# Fix error in bls library
patch -p1 < ../libBLSFuzz/bls_fixInclude.patch

# Dependency binaries
apt install texinfo yasm libgnutls28-dev

# Build libBLS
#
# Deps on HOST!!
# cd deps
# bash ./build.sh
# cd ..

# add to cmake

# Copy Harness
cp /src/fuzzing_projects/libBLSFuzz/src/harness.cpp /src/fuzzing_projects/libBLS/tools

# Configure the project and create a build directory.
cmake -H. -Bbuild
# Build all default targets using all cores.
cmake --build build -- -j$(nproc)

```

### 10.3.4 Threaded Run

```

#!/bin/bash
NUM_CORES=$(nproc)
MASTER_CORE=$((NUM_CORES - 1))
AFL_CMD="/AFLplusplus/afl-fuzz"
INPUT_DIR="/src/fuzzing_projects/libBLSFuzz/seeds"

```

```

OUTPUT_DIR="out"
TARGET="/src/fuzzing_projects/libBLSFuzz/build/libBLSHarness"

# Start master instance
$AFL_CMD -i $INPUT_DIR -o $OUTPUT_DIR -M master -m none -d -- $TARGET &

# Start slave instances
for ((i=1; i<NUM_CORES; i++)); do
    SLAVE_NAME="slave$i"
    $AFL_CMD -i $INPUT_DIR -o $OUTPUT_DIR -S $SLAVE_NAME -m none -d -- $TARGET 2>&1 &
done

echo "Started AFL++ with 1 master and $((NUM_CORES - 1)) slave instances."

```

## 10.4 Appendix tcpdump

### 10.4.1 Build

```

wget https://github.com/the-tcpdump-group/tcpdump/archive/refs/tags/tcpdump-4.99.4.tar.gz

wget https://github.com/the-tcpdump-group/libpcap/archive/refs/tags/libpcap-1.10.4.tar.gz
mv libpcap-libpcap-1.10.4/ libpcap-1.10.4

export LLVM_CONFIG="llvm-config-11"
CC=afl-clang-lto ./configure --enable-shared=no --prefix="$HOME/fuzzing_tcpdump/install/"
AFL_USE_ASAN=1 make

cd $HOME/fuzzing_tcpdump/tcpdump-tcpdump-4.9.2/
AFL_USE_ASAN=1 CC=afl-clang-lto ./configure --prefix="$HOME/fuzzing_tcpdump/install/"
AFL_USE_ASAN=1 make
AFL_USE_ASAN=1 make install

afl-fuzz -m none -i $HOME/fuzzing_tcpdump/tcpdump-tcpdump-4.99.4/tests/ -o
↪ $HOME/fuzzing_tcpdump/out/ -s 123 -- $HOME/fuzzing_tcpdump/install/bin/tcpdump -vvvXX -ee
↪ -nn -r @@

docker exec -it fuzccpthesis_aflplusplus_1 screen -S afl_fuzz_session bash -c 'afl-fuzz -m
↪ none -i $HOME/fuzzing_tcpdump/tcpdump-tcpdump-4.99.4/tests/ -o $HOME/fuzzing_tcpdump/out/
↪ -s 123 -- $HOME/fuzzing_tcpdump/install/bin/tcpdump -vvvXX -ee -nn -r @@'

```

## 10.5 Appendix libjpeg-turbo

### 10.5.1 Build

```

cd $HOME
mkdir fuzzing_jpeg && cd fuzzing_jpeg/

wget https://github.com/libjpeg-turbo/libjpeg-turbo/releases/download/3.0.2/libjpeg-turbo-3.0.2_
↪ .tar.gz
tar -xzvf libjpeg-turbo-3.0.2.tar.gz

cd libjpeg*
mkdir -p build
cd build
AFL_USE_ASAN=1 CC=afl-clang-lto CXX=afl-clang-lto++ cmake ..
↪ -DCMAKE_INSTALL_PREFIX=$HOME/fuzzing_jpeg/install/

AFL_USE_ASAN=1 make -j16
AFL_USE_ASAN=1 make install

afl-fuzz -m none -i $HOME/fuzzing_jpeg/libjpeg-turbo-3.0.2/testimages -o
↪ $HOME/fuzzing_jpeg/out/ -s 123 -- $HOME/fuzzing_jpeg/install/bin/cjpeg @@

```



```
docker exec -it fuzzcpptthesis aflplusplus_1 screen -S afl_fuzz_session bash -c 'afl-fuzz -m
↳ none -i $HOME/fuzzing_jpeg/libjpeg-turbo-3.0.2/testimages -o $HOME/fuzzing_jpeg/out/ -s
↳ 123 -- $HOME/fuzzing_jpeg/install/bin/cjpeg @@'
```

## 10.6 Appendix libexpat

### 10.6.1 Build

```
cd $HOME
mkdir fuzzing_expats && cd fuzzing_expats/

wget https://github.com/libexpat/libexpat/releases/download/R_2_6_2/expat-2.6.2.tar.gz
tar -xzf expat-2.6.2.tar.gz

AFL_USE_ASAN=1 CC=afl-clang-lto CXX=afl-clang-lto++ ./configure
↳ --prefix="$HOME/fuzzing_expats/install/"
AFL_USE_ASAN=1 make
AFL_USE_ASAN=1 make install

docker exec -it fuzzcpptthesis aflplusplus_1 screen -S afl_fuzz_session_expats bash -c 'afl-fuzz
↳ -m none -i $HOME/fuzzing_expats/testdata/largefiles -o $HOME/fuzzing_expats/out/ -s 123 --
↳ $HOME/fuzzing_expats/install/bin/xmlwf @@'
```

## 10.7 Appendix pdfcrack

### 10.7.1 Build

```
cd $HOME
mkdir -p fuzzing_pdfcrack && cd fuzzing_pdfcrack

wget https://sourceforge.net/projects/pdfrack/files/pdfrack/pdfrack-0.15/pdfrack-0.15.tar.gz
tar -xvf pdfrack-0.15.tar.gz
cd pdfrack-0.15

AFL_USE_ASAN=1 CC=afl-clang-lto CXX=afl-clang-lto++ make

/src/fuzzing_projects/pdfrack/encrypted_archive_pdfs/

docker exec -it pdfrack_aflplusplus_1 screen -S afl_fuzz_session bash -c 'afl-fuzz -m none -i
↳ /src/fuzzing_projects/pdfrack/encrypted_archive_pdfs/ -o $HOME/fuzzing_pdfcrack/out/ --
↳ $HOME/fuzzing_pdfcrack/pdfrack-0.15/pdfrack @@'
```

### 10.7.2 Bash: Generate Encrypted Samples

```
input="archive_pdfs/merrygoround00well.pdf"
num_files=3

for i in $(seq 1 $num_files); do
  user_pass="user${i}"
  owner_pass="owner${i}"
  output="encrypted_${i}2.pdf"
  qpdf --encrypt "$user_pass" "$owner_pass" 128 -- "$input" "$output"
  echo "Generated $output with user password $user_pass and owner password $owner_pass"
done
```

### 10.7.3 Bash: Minimize and Analyse Crashes

```
crash_dir="$HOME/fuzzing_pdfcrack/out/default/crashes"
pdfrack_dir="$HOME/fuzzing_pdfcrack/pdfrack-0.15"
```

```

output_dir="$HOME/src/fuzzing_projects/pdfcrack/analysis"
minimized_dir="$HOME/src/fuzzing_projects/pdfcrack/minimized"

mkdir -p "$output_dir"
mkdir -p "$minimized_dir"

for crash_file in "$crash_dir"/*; do
    base_name=$(basename "$crash_file")

    minimized_file="${minimized_dir}/${base_name}_min"
    output_file="${output_dir}/${base_name}_analysis"

    afl-tmin -i "$crash_file" -o "$minimized_file" -- "${pdfcrack_dir}/pdfcrack" -f @@

    if [ $? -ne 0 ]; then
        echo "Failed to minimize $crash_file. Check $minimized_file for details."
    else
        echo "Minimized $crash_file successfully. Processing..."

        "${pdfcrack_dir}/pdfcrack" -f "$minimized_file" > "$output_file" 2>&1

        if [ $? -ne 0 ]; then
            echo "Failed to execute $minimized_file. See $output_file for details."
        else
            echo "Processed $minimized_file successfully. Results in $output_file."
        fi
    fi
done

echo "Complete. Output files are in $output_dir."

```

#### 10.7.4 Python: Crash Cause Analysis

```

import os
import re

def parse_crash_details(filepath):
    """
    Parses a file to find the source of the crash.
    """
    cause = ""
    function_line = ""
    with open(filepath, "r") as file:
        content = file.read()
        # Find the cause of the crash from the SUMMARY line
        summary_match = re.search(r"SUMMARY: AddressSanitizer: ([\w-]+)", content)
        if summary_match:
            cause = summary_match.group(1)

        # Find the function and line number of the crash from the SUMMARY line
        function_line_match = re.search(
            r"SUMMARY: AddressSanitizer: [\w-]+ (?!.*/)?([~/]+:\d+:\d+ in \w+)", content
        )
        if function_line_match:
            function_line = function_line_match.group(1)

    return cause, function_line

def main():
    analysis_dir = "/pdfcrack/pdfcrack-min/pdfcrack/analysis"
    for filename in os.listdir(analysis_dir):

```

```

if filename.endswith("_analysis"):
    crash_id = filename.split(",")[0]
    filepath = os.path.join(analysis_dir, filename)
    cause, function_line = parse_crash_details(filepath)
    if cause and function_line:
        print(f"ID: {crash_id}, Cause: {cause}, Location: {function_line}")
    else:
        print(f"ID: {crash_id}, No detailed crash info found.")

main()

```

## 10.7.5 Crash Details

```

=====
==119919==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7fb0c0e00320 at pc
↳ 0x56263174e911 bp 0x7ffcbb5c17b0 sp 0x7ffcbb5c17a8
READ of size 1 at 0x7fb0c0e00320 thread T0
#0 0x56263174e910 in objStringToByte /root/fuzzing_pdfcrack/pdfcrack-0.15/pdfparser.c:230:14
#1 0x562631749c8f in parseRegularString
↳ /root/fuzzing_pdfcrack/pdfcrack-0.15/pdfparser.c:316:13
#2 0x562631744c6e in findTrailerDict /root/fuzzing_pdfcrack/pdfcrack-0.15/pdfparser.c:399:12
#3 0x562631744c6e in getEncryptedInfo
↳ /root/fuzzing_pdfcrack/pdfcrack-0.15/pdfparser.c:805:13
#4 0x5626316fbd16 in main /root/fuzzing_pdfcrack/pdfcrack-0.15/main.c:252:11
#5 0x7fb0c28ced8f (/lib/x86_64-linux-gnu/libc.so.6+0x29d8f) (BuildId:
↳ c289da5071a3399de893d2af81d6a30c62646e1e)
#6 0x7fb0c28cee3f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x29e3f) (BuildId:
↳ c289da5071a3399de893d2af81d6a30c62646e1e)
#7 0x562631621e04 in _start (/root/fuzzing_pdfcrack/pdfcrack-0.15/pdfcrack+0x5ae04)
↳ (BuildId: 8ac4c024dcac32c3)

Address 0x7fb0c0e00320 is located in stack of thread T0 at offset 288 in frame
#0 0x56263174954f in parseRegularString /root/fuzzing_pdfcrack/pdfcrack-0.15/pdfparser.c:289

This frame has 1 object(s):
[32, 288) 'buf' (line 293) <== Memory access at offset 288 overflows this variable
HINT: this may be a false positive if your program uses some custom stack unwind mechanism,
↳ swapcontext or vfork
(longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow
↳ /root/fuzzing_pdfcrack/pdfcrack-0.15/pdfparser.c:230:14 in objStringToByte
Shadow bytes around the buggy address:
0x7fb0c0e00080: f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8
0x7fb0c0e00100: f8 f8 f8 f8 f3 f3 f3 f3 f3 f3 f3 f3 00 00 00 00
0x7fb0c0e00180: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x7fb0c0e00200: f1 f1 f1 f1 00 00 00 00 00 00 00 00 00 00 00 00
0x7fb0c0e00280: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x7fb0c0e00300: 00 00 00 00 [f3]f3 f3 f3 f3 f3 f3 f3 00 00 00 00
0x7fb0c0e00380: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x7fb0c0e00400: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x7fb0c0e00480: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x7fb0c0e00500: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x7fb0c0e00580: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
Addressable: 00
Partially addressable: 01 02 03 04 05 06 07
Heap left redzone: fa
Freed heap region: fd
Stack left redzone: f1
Stack mid redzone: f2
Stack right redzone: f3
Stack after return: f5
Stack use after scope: f8

```

```

Global redzone:      f9
Global init order:  f6
Poisoned by user:   f7
Container overflow:  fc
Array cookie:       ac
Intra object redzone: bb
ASan internal:      fe
Left alloca redzone: ca
Right alloca redzone: cb
==119919==ABORTING

```

## 10.8 Appendix Libarchive

### 10.8.1 Build

```

cd $HOME
mkdir -p fuzzing_libarchive && cd fuzzing_libarchive

wget http://www.libarchive.org/downloads/libarchive-3.1.2.tar.gz
tar -xvf libarchive-3.1.2.tar.gz
cd libarchive-3.1.2

AFL_USE_ASAN=1 CC=afl-clang-lto CXX=afl-clang-lto++ ./configure
↪ --prefix="$HOME/fuzzing_libarchive/install/"
AFL_USE_ASAN=1 make
AFL_USE_ASAN=1 make install

cd $HOME/fuzzing_libarchive/libarchive-3.1.2/libarchive/test/
rm *mtree*
rm *rar_binary_data*

docker exec -it fuzzcpptthesis aflplusplus_1 screen -S afl_fuzz_session_libarchive bash -c
↪ 'afl-fuzz -m none -i $HOME/fuzzing_libarchive/libarchive-3.1.2/libarchive/test/ -o
↪ $HOME/fuzzing_libarchive/out/ -- $HOME/fuzzing_libarchive/install/bin/bsdtar -xvf @@'

```

### 10.8.2 Bash: Minimize and Analyze Script async

```

crash_dir="$HOME/fuzzing_libarchive/out/default/crashes"
libarchive_dir="$HOME/fuzzing_libarchive/install/bin"
output_dir="$HOME/fuzzing_libarchive/analysis"
minimized_dir="$HOME/fuzzing_libarchive/minimized"

# Create necessary directories
mkdir -p "$output_dir"
mkdir -p "$minimized_dir"

# Function to process each crash file
process_crash() {
    crash_file=$1
    base_name=$(basename "$crash_file")

    minimized_file="${minimized_dir}/${base_name}_min"
    output_file="${output_dir}/${base_name}_analysis"

    # Minimize the crash file
    afl-tmin -i "$crash_file" -o "$minimized_file" -- "${libarchive_dir}/bsdtar" -xvf @@
    result=$?

    if [ $result -ne 0 ]; then
        echo "Failed to minimize $crash_file. Check $minimized_file for details."
    else
        echo "Minimized $crash_file successfully. Processing..."
    fi
}

```

```

# Execute minimized file and capture the output
"${libarchive_dir}/bsdtar" -xvf "$minimized_file" > "$output_file" 2>&1
result=$?

if [ $result -ne 0 ]; then
    echo "Failed to execute $minimized_file. See $output_file for details."
else
    echo "Processed $minimized_file successfully. Results in $output_file."
fi
fi
}

export -f process_crash
export crash_dir minimized_dir output_dir libarchive_dir

# Find all crash files and process them in parallel
find "$crash_dir" -type f | parallel process_crash

echo "Complete. Output files are in $output_dir."

```

### 10.8.3 Python: Analyse crash cause

```

import os
import re

def parse_crash_details(filepath):
    """
    Parses a crash file to extract the precise location of the crash and the type of error.
    """
    try:
        with open(filepath, "r", errors="ignore") as file:
            content = file.read()

        # Regex to find the exact function and line number where the crash occurred
        crash_sites = re.findall(
            r"#\d+\s+(0x[w+])\s+in\s+(\S+)\s+(\S+:\d+:\d+)", content
        )
        error_type_match = re.search(r"ERROR: AddressSanitizer: ([\w-]+)", content)
        error_type = (
            error_type_match.group(1) if error_type_match else "Unknown Error Type"
        )

        # Return the first two crash files if found
        if crash_sites:
            results = [
                (error_type, function, location)
                for _, function, location in crash_sites[:2]
            ]
            return results
        else:
            summary_match = re.search(
                r"SUMMARY: AddressSanitizer: [\w-]+ (.*)", content
            )
            if summary_match:
                return [(error_type, "Summary provided", summary_match.group(1))]
            return [(error_type, "No crash location found", "")]
    except Exception as e:
        return [(f"Error processing file: {str(e)}", "")]

def main():
    analysis_dir = "/libarchive/minimized_crashes_analysis_2404/analysis"

```

```

results = {}

for filename in os.listdir(analysis_dir):
    if filename.endswith("_analysis"):
        crash_id = filename.split(",")[0]
        filepath = os.path.join(analysis_dir, filename)
        crash_details = parse_crash_details(filepath)

        for error_type, function, location in crash_details:
            key = (function, location)
            if key not in results:
                results[key] = {"count": 0, "ids": [], "error_type": error_type}
            results[key]["count"] += 1
            results[key]["ids"].append(crash_id)

# Output results
for (function, location), details in results.items():
    print(
        f"Error Type: {details['error_type']}, Function: {function}, Location: {location}"
    )
    print(f" IDs: {' ', ' '.join(details['ids'])}")
    print(f" Count: {details['count']}")
    print()

main()

```

## 10.8.4 Grouping Analysis of Crashfiles

Error Type: heap-buffer-overflow, Function: next\_code, Location: /root/fuzzing\_libarchive/libar  
↪ chive-3.1.2/libarchive/archive\_read\_support\_filter\_compress.c:386:20  
IDs: id:000344, id:000224, id:000225, id:000297, id:000242  
Count: 5

Error Type: heap-buffer-overflow, Function: compress\_filter\_read, Location: /root/fuzzing\_libar  
↪ chive/libarchive-3.1.2/libarchive/archive\_read\_support\_filter\_compress.c:287:10  
IDs: id:000344, id:000224, id:000225, id:000297, id:000242  
Count: 5

Error Type: heap-buffer-overflow, Function: readline, Location: /root/fuzzing\_libarchive/libarc  
↪ hive-3.1.2/libarchive/archive\_read\_support\_format\_mtree.c:1861:5  
IDs: id:000333, id:000342, id:000335, id:000338, id:000345, id:000330, id:000334, id:000353,  
↪ id:000349, id:000367, id:000371, id:000369, id:000001, id:000386, id:000384, id:000199,  
↪ id:000200, id:000204, id:000222, id:000229, id:000235, id:000236, id:000237, id:000244,  
↪ id:000245, id:000246, id:000252, id:000255, id:000258, id:000257, id:000264, id:000268,  
↪ id:000269, id:000273, id:000274, id:000277, id:000278, id:000284, id:000283, id:000285,  
↪ id:000287, id:000291, id:000298, id:000299, id:000306, id:000302, id:000309, id:000310,  
↪ id:000312, id:000322, id:000317, id:000316, id:000326, id:000328, id:000324, id:000323  
Count: 56

Error Type: heap-buffer-overflow, Function: read\_mtree, Location: /root/fuzzing\_libarchive/libar  
↪ rchive-3.1.2/libarchive/archive\_read\_support\_format\_mtree.c:937:9  
IDs: id:000333, id:000342, id:000335, id:000338, id:000345, id:000330, id:000334, id:000353,  
↪ id:000349, id:000367, id:000371, id:000369, id:000001, id:000386, id:000384, id:000199,  
↪ id:000200, id:000204, id:000222, id:000229, id:000235, id:000236, id:000237, id:000244,  
↪ id:000245, id:000246, id:000252, id:000255, id:000258, id:000257, id:000264, id:000268,  
↪ id:000269, id:000273, id:000274, id:000277, id:000278, id:000284, id:000283, id:000285,  
↪ id:000287, id:000291, id:000298, id:000299, id:000306, id:000302, id:000309, id:000310,  
↪ id:000312, id:000322, id:000317, id:000316, id:000326, id:000328, id:000324, id:000323  
Count: 56

Error Type: SEGV, Function: remove\_option, Location: /root/fuzzing\_libarchive/libarchive-3.1.2/  
↪ libarchive/archive\_read\_support\_format\_mtree.c:745:8

IDs: id:000340, id:000341, id:000365, id:000368, id:000382, id:000314, id:000313, id:000315  
Count: 8

Error Type: SEGV, Function: process\_global\_set, Location: /root/fuzzing\_libarchive/libarchive-3.1.2/libarchive/archive\_read\_support\_format\_mtree.c:781:3  
↳

IDs: id:000340, id:000341, id:000365, id:000368, id:000382, id:000314, id:000313, id:000315  
Count: 8

Error Type: SEGV, Function: read\_archive, Location:

↳ /root/fuzzing\_libarchive/libarchive-3.1.2/tar/read.c:314:8

IDs: id:000351, id:000332, id:000348, id:000339, id:000357, id:000358, id:000359, id:000337,  
↳ id:000364, id:000362, id:000378, id:000376, id:000379, id:000347, id:000363, id:000002,  
↳ id:000385, id:000003, id:000343, id:000356, id:000009, id:000374, id:000010, id:000346,  
↳ id:000011, id:000352, id:000012, id:000014, id:000013, id:000016, id:000015, id:000017,  
↳ id:000018, id:000373, id:000019, id:000020, id:000383, id:000021, id:000022, id:000023,  
↳ id:000024, id:000025, id:000027, id:000026, id:000028, id:000030, id:000029, id:000031,  
↳ id:000032, id:000034, id:000035, id:000360, id:000033, id:000036, id:000038, id:000037,  
↳ id:000041, id:000039, id:000040, id:000042, id:000043, id:000045, id:000044, id:000051,  
↳ id:000052, id:000049, id:000050, id:000059, id:000058, id:000047, id:000381, id:000060,  
↳ id:000061, id:000048, id:000062, id:000063, id:000064, id:000046, id:000065, id:000066,  
↳ id:000067, id:000057, id:000068, id:000071, id:000070, id:000069, id:000072, id:000073,  
↳ id:000074, id:000076, id:000075, id:000350, id:000329, id:000081, id:000083, id:000082,  
↳ id:000084, id:000085, id:000088, id:000086, id:000087, id:000090, id:000091, id:000055,  
↳ id:000093, id:000092, id:000056, id:000053, id:000094, id:000099, id:000101, id:000103,  
↳ id:000102, id:000100, id:000105, id:000107, id:000054, id:000110, id:000113, id:000111,  
↳ id:000112, id:000122, id:000123, id:000109, id:000115, id:000124, id:000127, id:000128,  
↳ id:000114, id:000129, id:000121, id:000118, id:000119, id:000116, id:000134, id:000125,  
↳ id:000130, id:000136, id:000138, id:000104, id:000126, id:000144, id:000145, id:000131,  
↳ id:000106, id:000148, id:000120, id:000150, id:000117, id:000137, id:000132, id:000153,  
↳ id:000135, id:000151, id:000133, id:000143, id:000159, id:000146, id:000149, id:000152,  
↳ id:000163, id:000164, id:000157, id:000154, id:000158, id:000167, id:000168, id:000165,  
↳ id:000170, id:000161, id:000171, id:000172, id:000155, id:000176, id:000156, id:000178,  
↳ id:000177, id:000173, id:000181, id:000005, id:000175, id:000182, id:000004, id:000160,  
↳ id:000162, id:000169, id:000166, id:000174, id:000179, id:000194, id:000196, id:000180,  
↳ id:000195, id:000007, id:000006, id:000198, id:000206, id:000201, id:000208, id:000202,  
↳ id:000183, id:000215, id:000214, id:000184, id:000217, id:000223, id:000139, id:000141,  
↳ id:000232, id:000212, id:000142, id:000238, id:000209, id:000234, id:000211, id:000240,  
↳ id:000221, id:000249, id:000248, id:000241, id:000210, id:000253, id:000203, id:000259,  
↳ id:000262, id:000267, id:000192, id:000270, id:000272, id:000193, id:000276, id:000261,  
↳ id:000331, id:000254, id:000281, id:000282, id:000216, id:000289, id:000265, id:000008,  
↳ id:000271, id:000294, id:000266, id:000286, id:000186, id:000243, id:000308, id:000303,  
↳ id:000292, id:000305, id:000311, id:000319, id:000321, id:000327, id:000140, id:000325,  
↳ id:000290, id:000247, id:000296, id:000260, id:000197, id:000293, id:000301, id:000307,  
↳ id:000304, id:000275, id:000263

Count: 267

Error Type: SEGV, Function: tar\_mode\_x, Location:

↳ /root/fuzzing\_libarchive/libarchive-3.1.2/tar/read.c:104:2

IDs: id:000351, id:000332, id:000348, id:000339, id:000357, id:000358, id:000359, id:000337,  
↳ id:000364, id:000362, id:000378, id:000376, id:000379, id:000347, id:000363, id:000002,  
↳ id:000385, id:000003, id:000343, id:000356, id:000009, id:000374, id:000010, id:000346,  
↳ id:000011, id:000352, id:000012, id:000014, id:000013, id:000016, id:000015, id:000017,  
↳ id:000018, id:000373, id:000019, id:000020, id:000383, id:000021, id:000022, id:000023,  
↳ id:000024, id:000025, id:000027, id:000026, id:000028, id:000030, id:000029, id:000031,  
↳ id:000032, id:000034, id:000035, id:000360, id:000033, id:000036, id:000038, id:000037,  
↳ id:000041, id:000039, id:000040, id:000042, id:000043, id:000045, id:000044, id:000051,  
↳ id:000052, id:000049, id:000050, id:000059, id:000058, id:000047, id:000381, id:000060,  
↳ id:000061, id:000048, id:000062, id:000063, id:000064, id:000046, id:000065, id:000066,  
↳ id:000067, id:000057, id:000068, id:000071, id:000070, id:000069, id:000072, id:000073,  
↳ id:000074, id:000076, id:000075, id:000350, id:000329, id:000081, id:000083, id:000082,  
↳ id:000084, id:000085, id:000088, id:000086, id:000087, id:000090, id:000091, id:000055,  
↳ id:000093, id:000092, id:000056, id:000053, id:000094, id:000099, id:000101, id:000103,  
↳ id:000102, id:000100, id:000105, id:000107, id:000054, id:000110, id:000113, id:000111,  
↳ id:000112, id:000122, id:000123, id:000109, id:000115, id:000124, id:000127, id:000128,  
↳ id:000114, id:000129, id:000121, id:000118, id:000119, id:000116, id:000134, id:000125,  
↳ id:000130, id:000136, id:000138, id:000104, id:000126, id:000144, id:000145, id:000131,  
↳ id:000106, id:000148, id:000120, id:000150, id:000117, id:000137, id:000132, id:000153,  
↳ id:000135, id:000151, id:000133, id:000143, id:000159, id:000146, id:000149, id:000152,  
↳ id:000163, id:000164, id:000157, id:000154, id:000158, id:000167, id:000168, id:000165,  
↳ id:000170, id:000161, id:000171, id:000172, id:000155, id:000176, id:000156, id:000178,  
↳ id:000177, id:000173, id:000181, id:000005, id:000175, id:000182, id:000004, id:000160,  
↳ id:000162, id:000169, id:000166, id:000174, id:000179, id:000194, id:000196, id:000180,  
↳ id:000195, id:000007, id:000006, id:000198, id:000206, id:000201, id:000208, id:000202,  
↳ id:000183, id:000215, id:000214, id:000184, id:000217, id:000223, id:000139, id:000141,  
↳ id:000232, id:000212, id:000142, id:000238, id:000209, id:000234, id:000211, id:000240,  
↳ id:000221, id:000249, id:000248, id:000241, id:000210, id:000253, id:000203, id:000259,  
↳ id:000262, id:000267, id:000192, id:000270, id:000272, id:000193, id:000276, id:000261,  
↳ id:000331, id:000254, id:000281, id:000282, id:000216, id:000289, id:000265, id:000008,  
↳ id:000271, id:000294, id:000266, id:000286, id:000186, id:000243, id:000308, id:000303,  
↳ id:000292, id:000305, id:000311, id:000319, id:000321, id:000327, id:000140, id:000325,  
↳ id:000290, id:000247, id:000296, id:000260, id:000197, id:000293, id:000301, id:000307,  
↳ id:000304, id:000275, id:000263  
Count: 267

Error Type: SEGV, Function: header\_bin\_be, Location: /root/fuzzing\_libarchive/libarchive-3.1.2/ |  
↳ libarchive/archive\_read\_support\_format\_cpio.c:909:31  
IDs: id:000354, id:000355, id:000366, id:000370, id:000108, id:000207, id:000213, id:000219,  
↳ id:000220, id:000227, id:000256, id:000228, id:000300, id:000320  
Count: 14

Error Type: SEGV, Function: archive\_read\_format\_cpio\_read\_header, Location: /root/fuzzing\_libar |  
↳ chive/libarchive-3.1.2/libarchive/archive\_read\_support\_format\_cpio.c:375:7  
IDs: id:000354, id:000355, id:000366, id:000370, id:000108, id:000207, id:000213, id:000219,  
↳ id:000220, id:000227, id:000256, id:000228, id:000300, id:000320  
Count: 14

Error Type: Unknown Error Type, Function: No crash location found, Location:  
IDs: README.txt\_analysis, id:000380, id:000079, id:000077, id:000191  
Count: 5

Error Type: heap-buffer-overflow, Function: archive\_read\_format\_tar\_read\_header, Location: /roo |  
↳ t/fuzzing\_libarchive/libarchive-3.1.2/libarchive/archive\_read\_support\_format\_tar.c:506:11  
IDs: id:000336, id:000377, id:000095, id:000096, id:000097, id:000098, id:000185, id:000187,  
↳ id:000189, id:000188, id:000190, id:000205, id:000147, id:000226, id:000231, id:000239,  
↳ id:000250, id:000251, id:000230, id:000295, id:000318  
Count: 21

Error Type: heap-buffer-overflow, Function: \_archive\_read\_next\_header2, Location:  
↳ /root/fuzzing\_libarchive/libarchive-3.1.2/libarchive/archive\_read.c:636:7  
IDs: id:000336, id:000377, id:000095, id:000096, id:000097, id:000098, id:000185, id:000187,  
↳ id:000189, id:000188, id:000190, id:000205, id:000147, id:000226, id:000231, id:000239,  
↳ id:000250, id:000251, id:000230, id:000295, id:000318



Count: 21

Error Type: heap-buffer-overflow, Function: detect\_form, Location: /root/fuzzing\_libarchive/libarchive-3.1.2/libarchive/archive\_read\_support\_format\_mtree.c:662:26  
↪ archive-3.1.2/libarchive/archive\_read\_support\_format\_mtree.c:662:26  
IDs: id:000375  
Count: 1

Error Type: heap-buffer-overflow, Function: read\_mtree, Location: /root/fuzzing\_libarchive/libarchive-3.1.2/libarchive/archive\_read\_support\_format\_mtree.c:934:8  
↪ rchive-3.1.2/libarchive/archive\_read\_support\_format\_mtree.c:934:8  
IDs: id:000375  
Count: 1

Error Type: heap-buffer-overflow, Function: lha\_read\_file\_extended\_header, Location: /root/fuzzing\_libarchive/libarchive-3.1.2/libarchive/archive\_read\_support\_format\_lha.c:1252:8  
↪ ing\_libarchive/libarchive-3.1.2/libarchive/archive\_read\_support\_format\_lha.c:1252:8  
IDs: id:000000, id:000361  
Count: 2

Error Type: heap-buffer-overflow, Function: lha\_read\_file\_header\_2, Location: /root/fuzzing\_libarchive/libarchive-3.1.2/libarchive/archive\_read\_support\_format\_lha.c:1031:8  
↪ archive/libarchive-3.1.2/libarchive/archive\_read\_support\_format\_lha.c:1031:8  
IDs: id:000000, id:000361  
Count: 2

Error Type: heap-use-after-free, Function: bid\_entry, Location: /root/fuzzing\_libarchive/libarchive-3.1.2/libarchive/archive\_read\_support\_format\_mtree.c:537:7  
↪ hive-3.1.2/libarchive/archive\_read\_support\_format\_mtree.c:537:7  
IDs: id:000080, id:000078  
Count: 2

Error Type: heap-use-after-free, Function: detect\_form, Location: /root/fuzzing\_libarchive/libarchive-3.1.2/libarchive/archive\_read\_support\_format\_mtree.c:646:15  
↪ rchive-3.1.2/libarchive/archive\_read\_support\_format\_mtree.c:646:15  
IDs: id:000080, id:000078  
Count: 2

Error Type: SEGV, Function: process\_extra, Location: /root/fuzzing\_libarchive/libarchive-3.1.2/libarchive/archive\_read\_support\_format\_zip.c:1716:16  
↪ libarchive/archive\_read\_support\_format\_zip.c:1716:16  
IDs: id:000089, id:000279  
Count: 2

Error Type: SEGV, Function: zip\_read\_local\_file\_header, Location: /root/fuzzing\_libarchive/libarchive-3.1.2/libarchive/archive\_read\_support\_format\_zip.c:1101:2  
↪ rchive-3.1.2/libarchive/archive\_read\_support\_format\_zip.c:1101:2  
IDs: id:000089, id:000279  
Count: 2

Error Type: heap-buffer-overflow, Function: read\_header.2507, Location: /root/fuzzing\_libarchive/libarchive-3.1.2/libarchive/archive\_read\_support\_format\_rar.c:1472:6  
↪ e/libarchive-3.1.2/libarchive/archive\_read\_support\_format\_rar.c:1472:6  
IDs: id:000372, id:000233, id:000280  
Count: 3

Error Type: heap-buffer-overflow, Function: archive\_read\_format\_rar\_read\_header, Location: /root/fuzzing\_libarchive/libarchive-3.1.2/libarchive/archive\_read\_support\_format\_rar.c:877:14  
↪ t/fuzzing\_libarchive/libarchive-3.1.2/libarchive/archive\_read\_support\_format\_rar.c:877:14  
IDs: id:000372, id:000233, id:000280  
Count: 3

Error Type: SEGV, Function: read\_CodersInfo, Location: /root/fuzzing\_libarchive/libarchive-3.1.2/libarchive/archive\_read\_support\_format\_7zip.c:1956:30  
↪ 2/libarchive/archive\_read\_support\_format\_7zip.c:1956:30  
IDs: id:000218, id:000288  
Count: 2

Error Type: SEGV, Function: read\_StreamsInfo, Location: /root/fuzzing\_libarchive/libarchive-3.1.2/libarchive/archive\_read\_support\_format\_7zip.c:2190:7  
↪ .2/libarchive/archive\_read\_support\_format\_7zip.c:2190:7  
IDs: id:000218, id:000288  
Count: 2

## 10.8.5 ASAN Report headline

```
=====
==330078==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x606000000120 at pc
↳ 0x5572b09dcb9e bp 0x7fff49ef0d30 sp 0x7fff49ef04f8
READ of size 3 at 0x606000000120 thread T0
#0 0x5572b09dcb9d in __asan_memmove (/root/fuzzing_libarchive/install/bin/bsdtar+0x14fb9d)
↳ (BuildId: 49b28e667647cc1c)
#1 0x5572b0b9ed44 in readline /root/fuzzing_libarchive/libarchive-3.1.2/libarchive/archive_
↳ read_support_format_mtree.c:1861:5
#2 0x5572b0b9ed44 in read_mtree /root/fuzzing_libarchive/libarchive-3.1.2/libarchive/archiv
↳ e_read_support_format_mtree.c:937:9
#3 0x5572b0b9ed44 in read_header /root/fuzzing_libarchive/libarchive-3.1.2/libarchive/archi
↳ ve_read_support_format_mtree.c:1007:7
#4 0x5572b0a714b0 in _archive_read_next_header2
↳ /root/fuzzing_libarchive/libarchive-3.1.2/libarchive/archive_read.c:636:7
#5 0x5572b0a70d91 in _archive_read_next_header
↳ /root/fuzzing_libarchive/libarchive-3.1.2/libarchive/archive_read.c:676:8
#6 0x5572b0a2a05f in archive_read_next_header
↳ /root/fuzzing_libarchive/libarchive-3.1.2/libarchive/archive_virtual.c:144:10
#7 0x5572b0a2a05f in read_archive /root/fuzzing_libarchive/libarchive-3.1.2/tar/read.c:235:7
#8 0x5572b0a2ea7f in tar_mode_x /root/fuzzing_libarchive/libarchive-3.1.2/tar/read.c:104:2
#9 0x5572b0a22852 in main /root/fuzzing_libarchive/libarchive-3.1.2/tar/bsdtar.c:804:3
#10 0x7eff40a2bd8f (/lib/x86_64-linux-gnu/libc.so.6+0x29d8f) (BuildId:
↳ c289da5071a3399de893d2af81d6a30c62646e1e)
#11 0x7eff40a2be3f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x29e3f) (BuildId:
↳ c289da5071a3399de893d2af81d6a30c62646e1e)
#12 0x5572b0943654 in _start (/root/fuzzing_libarchive/install/bin/bsdtar+0xb6654)
↳ (BuildId: 49b28e667647cc1c)
```

0x606000000120 is located 0 bytes after 64-byte region [0x6060000000e0,0x606000000120) allocated by thread T0 here:

```
#0 0x5572b09dd8a5 in realloc (/root/fuzzing_libarchive/install/bin/bsdtar+0x1508a5)
↳ (BuildId: 49b28e667647cc1c)
#1 0x5572b0c0ce3a in archive_string_ensure
↳ /root/fuzzing_libarchive/libarchive-3.1.2/libarchive/archive_string.c:307:14
```

SUMMARY: AddressSanitizer: heap-buffer-overflow

```
↳ (/root/fuzzing_libarchive/install/bin/bsdtar+0x14fb9d) (BuildId: 49b28e667647cc1c) in
↳ __asan_memmove
```

Shadow bytes around the buggy address:

```
0x605ffffffe80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x605ffffff00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x605ffffff80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x606000000000: fa fa fa fa 00 00 00 00 00 00 00 00 fa fa fa fa
0x606000000080: 00 00 00 00 00 00 00 fa fa fa fa fa 00 00 00 00
=>0x606000000100: 00 00 00 00[fa]fa fa fa fa fa fa fa fa fa fa fa
0x606000000180: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x606000000200: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x606000000280: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x606000000300: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x606000000380: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
```

Shadow byte legend (one shadow byte represents 8 application bytes):

```
Addressable: 00
Partially addressable: 01 02 03 04 05 06 07
Heap left redzone: fa
Freed heap region: fd
Stack left redzone: f1
Stack mid redzone: f2
Stack right redzone: f3
Stack after return: f5
Stack use after scope: f8
Global redzone: f9
Global init order: f6
```

```

Poisoned by user:      f7
Container overflow:    fc
Array cookie:         ac
Intra object redzone: bb
ASan internal:        fe
Left alloca redzone:  ca
Right alloca redzone: cb
==330078==ABORTING

```

## 10.9 Appendix ImageMagick

### 10.9.1 Build

```
docker-compose -p imagemagick up -d
```

```

cd $HOME
mkdir -p fuzzing_imagemagick && cd fuzzing_imagemagick

#ImageMagick
wget https://github.com/ImageMagick/ImageMagick/archive/refs/tags/7.1.1-4.tar.gz
tar -xvf 7.1.1-4.tar.gz
cd ImageMagick-7.1.1-4

AFL_USE_ASAN=1 CC=afl-clang-lto CXX=afl-clang-lto++ ./configure
↪ --prefix="$HOME/fuzzing_imagemagick/install/"
AFL_USE_ASAN=1 make -j$(nproc)
AFL_USE_ASAN=1 make install

```

```

docker exec -it fuzzcptestthesis aflplusplus_1 screen -S afl_fuzz_session bash -c 'afl-fuzz -m
↪ none -i $HOME/fuzzing_imagemagick/ImageMagick-7.1.1-4/tests/ -o
↪ $HOME/fuzzing_imagemagick/out/ -- $HOME/fuzzing_imagemagick/install/bin/magick @@"

```

### 10.9.2 Build With known Vuln-Files as Input

```

AFL_USE_ASAN=1 CC=afl-clang-fast CXX=afl-clang-fast++ ./configure
↪ --prefix="$HOME/fuzzing_imagemagick/install/"
AFL_USE_ASAN=1 make -j$(nproc)
AFL_USE_ASAN=1 make install

```

```

mkdir -p ImageMagick/lcantdump-coredump-images/
mkdir -p gif
cd gif
wget -O index.html "https://lcamtuf.coredump.cx/afl/demo/gif_im/full/images/"

grep -o 'href="[~]*.gif"' index.html | cut -d'"' -f2 | while read line; do wget
↪ "https://lcamtuf.coredump.cx/afl/demo/gif_im/full/images/$line"; done

mkdir -p bmp
cd bmp
wget -O index.html "https://lcamtuf.coredump.cx/afl/demo/bmp/full/images/"

grep -o 'href="[~]*.bmp"' index.html | cut -d'"' -f2 | while read line; do wget
↪ "https://lcamtuf.coredump.cx/afl/demo/bmp/full/images/$line"; done

mkdir -p ico
cd ico
wget -O index.html "https://lcamtuf.coredump.cx/afl/demo/ico/full/images/"

grep -o 'href="[~]*.ico"' index.html | cut -d'"' -f2 | while read line; do wget
↪ "https://lcamtuf.coredump.cx/afl/demo/ico/full/images/$line"; done

```

```
docker exec -it imagemagick_aflplusplus_1 screen -S afl_fuzz_session bash -c 'afl-fuzz -m none
↳ -i /src/fuzzing_projects/ImageMagick/lcamtuf-coredump-images/ -o
↳ $HOME/fuzzing_imagemagick/out/ -- $HOME/fuzzing_imagemagick/install/bin/magick @@'
```

## 10.10 Appendix Analysis

### 10.10.1 Python Analysis Script for Metrics

```
import os
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import locale
import yaml
import logging

from dataclasses import dataclass
from typing import List, Dict
from pandas import DataFrame
from scipy.cluster.hierarchy import linkage, leaves_list
from scipy.spatial.distance import pdist, squareform

logging.basicConfig(
    level=logging.INFO, format="%(asctime)s - %(levelname)s - %(message)s"
)

@dataclass
class Metric:
    label: str
    title: str
    plot: bool

@dataclass
class Config:
    numeric_columns: List[str]
    percentage_columns: List[str]
    metrics: Dict[str, Metric]

    @staticmethod
    def from_dict(config_dict: Dict[str, any]) -> "Config":
        metrics = {k: Metric(**v) for k, v in config_dict["metrics"].items()}
        return Config(
            numeric_columns=config_dict["numeric_columns"],
            percentage_columns=config_dict["percentage_columns"],
            metrics=metrics,
        )

class FuzzerStatsAnalyzer:
    def __init__(self, directory: str, config_file: str) -> None:
        self.directory = directory
        self.config = self.load_config(config_file)

    def load_config(self, config_file: str) -> Config:
        try:
            with open(config_file, "r") as file:
                config_dict = yaml.safe_load(file)
            return Config.from_dict(config_dict)
        except Exception as e:
            logging.error(f"Error loading configuration file: {e}")
            raise
```

```

def read_fuzzer_stats(self, file_path: str) -> Dict[str, str]:
    """Read a fuzzer_stats file and extract the relevant data."""
    data = {}
    try:
        with open(file_path, "r") as file:
            for line in file:
                if ":" in line:
                    key, value = line.split(":", 1)
                    key, value = key.strip(), value.strip()
                    # Strip percentage sign for percentage values
                    if key in self.config.percentage_columns:
                        value = value.rstrip("%")
                    data[key] = value
    except Exception as e:
        logging.error(f"Error reading fuzzer_stats file {file_path}: {e}")
    return data

def aggregate_data(self) -> DataFrame:
    """Aggregate data from all fuzzer_stats files in a directory into a DataFrame."""
    all_data = []
    for root, _, files in os.walk(self.directory):
        for file in files:
            if file == "fuzzer_stats":
                file_path = os.path.join(root, file)
                data = self.read_fuzzer_stats(file_path)
                if data:
                    data["Program"] = self.get_program_path(root)
                    all_data.append(data)
    return pd.DataFrame(all_data)

@staticmethod
def get_program_path(root: str) -> str:
    """Get the program path from the root directory."""
    path_parts = root.split(os.sep)
    return os.sep.join(path_parts[-3:]) if len(path_parts) > 2 else root

def convert_data_types(self, df: DataFrame) -> DataFrame:
    """Convert data types of specific columns to numeric."""
    for column in self.config.numeric_columns:
        if column in df.columns:
            df[column] = pd.to_numeric(df[column], errors="coerce")

    # Convert run_time to hours
    if "run_time" in df.columns:
        df["run_time"] = df["run_time"] / 3600.0

    return df

@staticmethod
def filter_non_zero_columns(df: DataFrame) -> DataFrame:
    """Filter out columns that have all zero values."""
    return df.loc[:, (df != 0).any(axis=0)]

@staticmethod
def clean_program_names(program: str) -> str:
    """Clean program names by removing specific parts."""
    if program.startswith("fuzzer_run-"):
        program = program[len("fuzzer_run-") :]
    return program.split("/") [0] if "/" in program else program

def aggregate_slaves(self, df: DataFrame) -> DataFrame:
    """Aggregate data by summing values for each program's master and slaves."""

```

```

df["BaseProgram"] = (
    df["Program"]
    .str.replace(r"/slave\d+$", "", regex=True)
    .str.replace(r"/master$", "", regex=True)
    .apply(self.clean_program_names)
)
numeric_columns = [
    col for col in self.config.numeric_columns if col in df.columns
]
numeric_df = df[numeric_columns + ["BaseProgram"]]

aggregated_df = numeric_df.groupby("BaseProgram").mean().reset_index()

aggregated_df["Program"] = aggregated_df["BaseProgram"]
return aggregated_df.drop(columns=["BaseProgram"])

def calculate_averages(self, df: DataFrame) -> DataFrame:
    """Calculate averages for all numerical columns for each program."""
    df["BaseProgram"] = (
        df["Program"]
        .str.replace(r"/slave\d+$", "", regex=True)
        .str.replace(r"/master$", "", regex=True)
        .apply(self.clean_program_names)
    )
    numeric_columns = [
        col for col in self.config.numeric_columns if col in df.columns
    ]
    numeric_df = df[numeric_columns + ["BaseProgram"]]
    avg_df = numeric_df.groupby("BaseProgram").mean().reset_index()
    avg_df["Program"] = avg_df["BaseProgram"]
    return avg_df.drop(columns=["BaseProgram"])

def plot_metrics(self, df: DataFrame, exclude_zeros: bool = False) -> None:
    """Generate plots for key metrics and save them as PNG files."""
    sns.set(style="whitegrid")
    locale.setlocale(locale.LC_ALL, "de_CH.UTF-8")

    for metric, properties in self.config.metrics.items():
        if properties.plot and metric in df.columns:
            self.create_plot(df, metric, properties, exclude_zeros)

def create_plot(
    self, df: DataFrame, metric: str, properties: Metric, exclude_zeros: bool
) -> None:
    """Create and save a bar plot for a specific metric."""
    df_plot = df[df[metric] != 0] if exclude_zeros else df
    plt.figure(figsize=(12, 7))
    ax = sns.barplot(
        x="Program",
        y=metric,
        data=df_plot,
        palette="viridis",
        hue="Program",
        legend=False,
    )
    ax.set_xlabel("")
    ax.set_ylabel(properties.label, fontsize=12)
    ax.set_title(properties.title, fontsize=14)

    tick_labels = ax.get_xticklabels()
    ax.set_xticks(range(len(tick_labels)))
    ax.set_xticklabels(tick_labels, rotation=45, ha="right")

```

```

for p in ax.patches:
    value = p.get_height()
    formatted_value = locale.format_string("%.2f", value, grouping=True)
    ax.annotate(
        formatted_value,
        (p.get_x() + p.get_width() / 2.0, p.get_height()),
        ha="center",
        va="center",
        xytext=(0, 9),
        textcoords="offset points",
    )

plt.tight_layout()
os.makedirs("./plots", exist_ok=True)
plt.savefig(os.path.join("./plots", f"{metric}.png"))
plt.close()

def plot_correlation_heatmap(self, df: DataFrame) -> None:
    """Generate an improved heatmap showing the correlation between metrics, grouped by
    ↪ hierarchical clustering."""
    numeric_columns = [
        col for col in self.config.numeric_columns if col in df.columns
    ]
    plt.figure(figsize=(24, 22))

    # Calculate the correlation matrix
    corr = df[numeric_columns].corr()

    # Perform hierarchical clustering on the correlation matrix
    distance_matrix = 1 - corr
    condensed_distance_matrix = pdist(distance_matrix)
    linkage_matrix = linkage(condensed_distance_matrix, method="average")
    leaves_order = leaves_list(linkage_matrix)

    # Reorder the correlation matrix
    ordered_corr = corr.iloc[leaves_order, leaves_order]

    # Plot the reordered correlation heatmap
    sns.heatmap(ordered_corr, annot=True, fmt=".2f", cmap="coolwarm", square=True)
    plt.title("Correlation Heatmap of Metrics", fontsize=16)
    plt.tight_layout()
    os.makedirs("./plots/correlation", exist_ok=True)
    plt.savefig(os.path.join("./plots/correlation", "correlation_heatmap.png"))
    plt.close()

def print_high_correlations(self, df: DataFrame, threshold: float = 0.90) -> None:
    """Print pairs of metrics that have a correlation greater than the specified
    ↪ threshold."""
    numeric_columns = [
        col for col in self.config.numeric_columns if col in df.columns
    ]
    corr = df[numeric_columns].corr()

    # Find pairs with correlation greater than the threshold
    high_corr_pairs = [
        (i, j, corr.at[i, j])
        for i in corr.columns
        for j in corr.columns
        if i != j and corr.at[i, j] > threshold
    ]

    # Print the high correlation pairs
    if high_corr_pairs:

```

```

        print(f"Pairs of metrics with correlation greater than {threshold}:")
        for i, j, value in high_corr_pairs:
            print(f"{i} and {j}: {value:.2f}")
    else:
        print(f"No pairs of metrics have a correlation greater than {threshold}.")

def plot_distributions(self, df: DataFrame) -> None:
    """Generate distribution plots for key metrics."""
    sns.set(style="whitegrid")
    locale.setlocale(locale.LC_ALL, "de_CH.UTF-8")

    for metric, properties in self.config.metrics.items():
        if properties.plot and metric in df.columns:
            plt.figure(figsize=(10, 6))
            sns.histplot(df[metric].dropna(), kde=True, bins=30, color="blue")
            plt.xlabel(properties.label, fontsize=12)
            plt.ylabel("Frequency", fontsize=12)
            plt.title(f"Distribution of {properties.title}", fontsize=14)
            plt.tight_layout()
            os.makedirs("./plots/distributions", exist_ok=True)
            plt.savefig(
                os.path.join("./plots/distributions", f"{metric}_distribution.png")
            )
            plt.close()

def plot_trends(self, df: DataFrame) -> None:
    """Generate trend plots over time for select metrics."""
    sns.set(style="whitegrid")
    locale.setlocale(locale.LC_ALL, "de_CH.UTF-8")

    # Ensure datetime format
    df["run_time"] = pd.to_numeric(df["run_time"], errors="coerce")
    df = df.dropna(subset=["run_time"])
    df["run_time"] = df["run_time"].astype(int)

    for metric, properties in self.config.metrics.items():
        if properties.plot and metric in df.columns:
            plt.figure(figsize=(12, 7))
            sns.lineplot(data=df, x="run_time", y=metric, hue="Program", marker="o")
            plt.xlabel("Time [h]", fontsize=12)
            plt.ylabel(properties.label, fontsize=12)
            plt.title(f"Trend of {properties.title} Over Time", fontsize=14)
            plt.xticks(rotation=45)
            plt.tight_layout()
            os.makedirs("./plots/trends", exist_ok=True)
            plt.savefig(os.path.join("./plots/trends", f"{metric}_trend.png"))
            plt.close()

def plot_coverage_vs_runtime(self, df: DataFrame) -> None:
    """Plot the relationship between coverage percentage and run time."""
    plt.figure(figsize=(12, 7))
    sns.scatterplot(
        data=df, x="run_time", y="bitmap_cvlg", hue="Program", marker="o"
    )
    plt.xlabel("Run Time [h]", fontsize=12)
    plt.ylabel("Coverage Percentage", fontsize=12)
    plt.title("Coverage vs. Run Time", fontsize=14)
    plt.tight_layout()
    os.makedirs("./plots/relations", exist_ok=True)
    plt.savefig(os.path.join("./plots/relations", "coverage_vs_runtime.png"))
    plt.close()

def plot_crashes_vs_executions(self, df: DataFrame) -> None:

```



```

"""Plot the relationship between the number of crashes and the total executions."""
plt.figure(figsize=(12, 7))
sns.scatterplot(
    data=df, x="execs_done", y="saved_crashes", hue="Program", marker="o"
)
plt.xlabel("Total Executions", fontsize=12)
plt.ylabel("Number of Crashes", fontsize=12)
plt.title("Crashes vs. Executions", fontsize=14)
plt.tight_layout()
os.makedirs("./plots/relations", exist_ok=True)
plt.savefig(os.path.join("./plots/relations", "crashes_vs_executions.png"))
plt.close()

def plot_pairplot_key_metrics(self, df: DataFrame) -> None:
    """Generate pair plots for key metrics."""
    key_metrics = [
        "run_time",
        "bitmap_cvg",
        "execs_done",
        "saved_crashes",
        "saved_hangs",
    ]
    sns.pairplot(df[key_metrics])
    plt.suptitle("Pair Plot of Key Metrics", y=1.02, fontsize=16)
    plt.tight_layout()
    os.makedirs("./plots/relations", exist_ok=True)
    plt.savefig(os.path.join("./plots/relations", "pairplot_key_metrics.png"))
    plt.close()

def plot_execution_speed_by_program(self, df: DataFrame) -> None:
    """Generate box plots for execution speed by program."""
    plt.figure(figsize=(14, 7))
    sns.boxplot(data=df, x="Program", y="execs_per_sec")
    plt.xlabel("Program", fontsize=12)
    plt.ylabel("Executions per Second", fontsize=12)
    plt.title("Execution Speed by Program", fontsize=14)
    plt.xticks(rotation=45, ha="right")
    plt.tight_layout()
    os.makedirs("./plots/relations", exist_ok=True)
    plt.savefig(os.path.join("./plots/relations", "execution_speed_by_program.png"))
    plt.close()

def plot_coverage_over_time(self, df: DataFrame) -> None:
    """Plot the coverage percentage over time."""
    plt.figure(figsize=(12, 7))
    sns.lineplot(
        data=df, x="last_update", y="bitmap_cvg", hue="Program", marker="o"
    )
    plt.xlabel("Time", fontsize=12)
    plt.ylabel("Coverage Percentage", fontsize=12)
    plt.title("Coverage Percentage Over Time", fontsize=14)
    plt.xticks(rotation=45)
    plt.tight_layout()
    os.makedirs("./plots/relations", exist_ok=True)
    plt.savefig(os.path.join("./plots/relations", "coverage_over_time.png"))
    plt.close()

def plot_execution_speed_distribution(self, df: DataFrame) -> None:
    """Generate a histogram for the distribution of executions per second."""
    plt.figure(figsize=(10, 6))
    sns.histplot(df["execs_per_sec"].dropna(), kde=True, bins=30, color="green")
    plt.xlabel("Executions per Second", fontsize=12)
    plt.ylabel("Frequency", fontsize=12)

```

```

plt.title("Distribution of Execution Speed", fontsize=14)
plt.tight_layout()
os.makedirs("./plots/relations", exist_ok=True)
plt.savefig(
    os.path.join("./plots/relations", "execution_speed_distribution.png")
)
plt.close()

def create_summary_table(self, df: DataFrame) -> None:
    """Create and save a summary table of all metrics for all projects."""
    numeric_columns = [
        col for col in self.config.numeric_columns if col in df.columns
    ]
    summary_df = df.groupby("Program")[numeric_columns].mean().reset_index()
    summary_file_path = os.path.join("./plots", "summary_table.csv")
    summary_df.to_csv(summary_file_path, index=False)
    logging.info(f"Summary table saved to {summary_file_path}")

def analyze_cycles_and_coverage(self, df: DataFrame) -> None:
    """Analyze cycles, code coverage, and other termination conditions."""
    plt.figure(figsize=(14, 7))
    sns.scatterplot(
        data=df, x="cycles_done", y="bitmap_cvg", hue="Program", marker="o"
    )
    plt.xlabel("Cycles Done", fontsize=12)
    plt.ylabel("Code Coverage (%)", fontsize=12)
    plt.title("Cycles Done vs. Code Coverage", fontsize=14)
    plt.tight_layout()
    os.makedirs("./plots/abbruchbedingungen", exist_ok=True)
    plt.savefig(os.path.join("./plots", "cycles_vs_coverage.png"))
    plt.close()

def run_analysis(self, exclude_zeros: bool = False) -> None:
    """Run the complete analysis pipeline."""
    logging.info("Starting analysis pipeline...")
    df = self.aggregate_data()
    if df.empty:
        logging.warning("No data found. Exiting analysis.")
        return
    df = self.convert_data_types(df)
    df = self.filter_non_zero_columns(df)
    df = self.aggregate_slaves(df)
    self.create_summary_table(df)
    self.analyze_cycles_and_coverage(df)
    self.plot_metrics(df, exclude_zeros)
    self.plot_correlation_heatmap(df)
    self.print_high_correlations(df)
    self.plot_distributions(df)
    self.plot_trends(df)
    self.plot_coverage_vs_runtime(df)
    self.plot_crashes_vs_executions(df)
    self.plot_pairplot_key_metrics(df)
    self.plot_execution_speed_by_program(df)
    self.plot_coverage_over_time(df)
    self.plot_execution_speed_distribution(df)
    logging.info("Analysis pipeline completed.")

# Directory containing the fuzzer_stats files
directory = "./afl_stats"
config_file = "analyzer_config.yaml"

# Initialize the analyzer and run the analysis

```

```
analyzer = FuzzerStatsAnalyzer(directory, config_file)
analyzer.run_analysis(exclude_zeros=True)
```

### 10.10.2 Config for Python Analysis Script for Metrics

```
numeric_columns:
- start_time
- last_update
- run_time
- fuzzer_pid
- cycles_done
- cycles_wo_finds
- time_wo_finds
- execs_done
- execs_per_sec
- execs_ps_last_min
- corpus_count
- corpus_favored
- corpus_found
- corpus_imported
- corpus_variable
- max_depth
- cur_item
- pending_favs
- pending_total
- stability
- bitmap_cvg
- saved_crashes
- saved_hangs
- last_find
- last_crash
- last_hang
- execs_since_crash
- exec_timeout
- slowest_exec_ms
- peak_rss_mb
- edges_found
- total_edges
- var_byte_count
- havoc_expansion
- auto_dict_entries
- testcache_size
- testcache_count
- testcache_evict

percentage_columns:
- stability
- bitmap_cvg

metrics:
start_time:
  label: "Start Time"
  title: "Start Time per Program"
  plot: false
last_update:
  label: "Last Update"
  title: "Last Update per Program"
  plot: false
run_time:
  label: "Run Time [h]"
  title: "Run Time per Program"
  plot: true
fuzzer_pid:
```

```

    label: "Fuzzer PID"
    title: "Fuzzer PID per Program"
    plot: false
cycles_done:
    label: "Cycles Done"
    title: "Cycles Done per Program"
    plot: true
cycles_wo_finds:
    label: "Cycles Without Finds"
    title: "Cycles Without Finds per Program"
    plot: true
time_wo_finds:
    label: "Time Without Finds"
    title: "Time Without Finds per Program"
    plot: true
execs_done:
    label: "Total Executions"
    title: "Total Executions per Program"
    plot: true
execs_per_sec:
    label: "Executions per Second"
    title: "Executions per Second per Program"
    plot: true
execs_ps_last_min:
    label: "Executions per Second Last Minute"
    title: "Executions per Second Last Minute per Program"
    plot: false
corpus_count:
    label: "Corpus Count"
    title: "Corpus Count per Program"
    plot: true
corpus_favored:
    label: "Corpus Favored"
    title: "Corpus Favored per Program"
    plot: false
corpus_found:
    label: "Corpus Found"
    title: "Corpus Found per Program"
    plot: true
corpus_imported:
    label: "Corpus Imported"
    title: "Corpus Imported per Program"
    plot: false
corpus_variable:
    label: "Corpus Variable"
    title: "Corpus Variable per Program"
    plot: false
max_depth:
    label: "Max Depth"
    title: "Max Depth per Program"
    plot: true
cur_item:
    label: "Current Item"
    title: "Current Item per Program"
    plot: false
pending_favs:
    label: "Pending Favorites"
    title: "Pending Favorites per Program"
    plot: false
pending_total:
    label: "Pending Total"
    title: "Pending Total per Program"
    plot: false

```

```

stability:
  label: "Stability Percentage"
  title: "Stability Percentage per Program"
  plot: true
bitmap_cvg:
  label: "Coverage Percentage"
  title: "Coverage Percentage per Program"
  plot: true
saved_crashes:
  label: "Crashes"
  title: "Crashes per Program"
  plot: true
saved_hangs:
  label: "Hangs"
  title: "Hangs per Program"
  plot: true
last_find:
  label: "Last Find"
  title: "Last Find per Program"
  plot: true
last_crash:
  label: "Last Crash"
  title: "Last Crash per Program"
  plot: true
last_hang:
  label: "Last Hang"
  title: "Last Hang per Program"
  plot: true
execs_since_crash:
  label: "Executions Since Crash"
  title: "Executions Since Crash per Program"
  plot: true
exec_timeout:
  label: "Execution Timeout"
  title: "Execution Timeout per Program"
  plot: true
slowest_exec_ms:
  label: "Slowest Execution (ms)"
  title: "Slowest Execution (ms) per Program"
  plot: false
peak_rss_mb:
  label: "Peak RSS (MB)"
  title: "Peak RSS (MB) per Program"
  plot: true
edges_found:
  label: "Edges Found"
  title: "Edges Found per Program"
  plot: true
total_edges:
  label: "Total Edges"
  title: "Total Edges per Program"
  plot: false
var_byte_count:
  label: "Variable Byte Count"
  title: "Variable Byte Count per Program"
  plot: true
havoc_expansion:
  label: "Havoc Expansion"
  title: "Havoc Expansion per Program"
  plot: false
auto_dict_entries:
  label: "Auto Dictionary Entries"
  title: "Auto Dictionary Entries per Program"

```

```

    plot: true
testcache_size:
  label: "Test Cache Size"
  title: "Test Cache Size per Program"
  plot: false
testcache_count:
  label: "Test Cache Count"
  title: "Test Cache Count per Program"
  plot: false
testcache_evict:
  label: "Test Cache Evict"
  title: "Test Cache Evict per Program"
  plot: false

```

### 10.10.3 Sample Penetration Test Report for vt-parser

#### ### Sample Penetration Test Report for vt-parser

**Project Name:** VT-Parser

**Report Date:** [Date]

**Testing Period:** [Start Date] - [End Date]

#### ### 1. Chosen Fuzzer

- **Fuzzer:** AFL++
- **Reason:** AFL++ was chosen due to its efficiency in finding memory safety issues and its ↪ high execution speed, which is crucial for CI/CD environments.

#### ### 2. Approach Tried

- **Setup:** The `vt-parser` was instrumented and prepared for fuzzing with AFL++. Appropriate ↪ initial seed files were selected to cover a wide range of inputs.
- **Fuzzing Parameters:** AFL++ was configured with sanitizers such as ASAN to detect memory ↪ errors.
- **Environment:** The fuzzing process was integrated into a Docker container to ensure ↪ consistency and reproducibility.

#### ### 3. Fuzzer Run Results

- **Run Time:** The fuzzer was run for 140.5 hours.
- **Number of Cycles:** Completed 293.7 cycles.
- **Crashes Found:** Identified 3 unique crashes.
- **Code Coverage Achieved:** Reached 36.1% code coverage.

#### ### 4. Findings

- **Vulnerabilities Discovered:**
  - **Buffer Overflow:** Found in the parsing function when handling specific malformed inputs.
  - **Use-After-Free:** Discovered in memory management when parsing certain sequences of data.
- **Crash Analysis:**
  - **Buffer Overflow:** The buffer overflow can lead to arbitrary code execution if exploited. ↪ Detailed analysis showed that the issue is triggered by input exceeding buffer limits ↪ without proper checks.
  - **Use-After-Free:** This vulnerability can cause undefined behavior and potential crashes, ↪ resulting from the parser freeing memory that is still in use.

#### ### 5. Further Actions

- **Alternative Tools:** Suggested trying libFuzzer for targeted function-level fuzzing to ↪ uncover deeper logical bugs.
- **Extended Duration:** Recommended running the fuzzer for a longer period to increase the ↪ chances of discovering more subtle bugs.
- **Parameter Adjustments:** Adjusting parameters such as the mutation strategy and memory ↪ limits to explore deeper code paths.

#### ### 6. Types of Vulnerabilities Tested

- **Memory Safety Issues**: Focused on detecting buffer overflows, use-after-free errors, and  
↳ null pointer dereferences.
- **Input Validation**: Ensured that the application correctly handles malformed or unexpected  
↳ input data.
- **Resource Management**: Checked for issues like memory leaks and improper resource  
↳ deallocation.

**Conclusion**: The fuzzing run uncovered critical vulnerabilities in `vt-parser`. It is  
↳ recommended to address these issues promptly and consider further testing with additional  
↳ tools and extended durations to ensure comprehensive coverage.