Bitmap Fuzzing: A Comparative Analysis of Libfuzzer and AFL Tools

ANITHA S[1], BHADRIKA M K[2], INDUJA R[3], KANISHKA R[4], KOWSHIKA M[5], MAHALAKSHMI M[6]
[1, 2, 3, 4 ,5, 6]*Department of Computer Science and Engineering (Cyber Security), Sri Shakthi Institute of Engineering and Technology, Tamilnadu, India*

Abstract*— This research work compares the effectiveness of bitmap fuzzing between two prominent algorithms exploited in LibFuzzer and AFL, aimed at identifying vulnerabilities in software handling bitmap images. Bitmap fuzzing generates varied image-based test cases that rigorously test software, revealing potential security flaws. In this analysis, LibFuzzer, an in-process guided fuzzer, and AFL, an external fuzzer driven by coverage feedback, are both utilized to evaluate their accuracy in detecting errors within bitmap¬processing applications. The performance is assessed based on the types and frequency of errors found, offering a layered perspective on error-handling strength in image processing contexts. By recording software crashes and categorizing the faults, this proposed research provides important insights into the comparative strengths and limitations of these fuzzing tools. The experimental results aid significant improvements in fuzzing practices, enhancing security frameworks by enabling early identification of vulnerabilities in multimedia-focused applications. The devised comparative research highlights the critical role of fuzzing tools in building robust, resilient software defenses.

Indexed Terms- Bitmap Fuzzing, LibFuzzer, AFL, Fuzz Testing

I. INTRODUCTION

Fuzz testing has emerged as a key technique for identifying software vulnerabilities, particularly in file format parsers where complex data structures are common. Bitmap file formats, like BMP, are often targeted due to their widespread use and the complexity involved in parsing various pixel structures and metadata, making them prone to errors that could lead to security vulnerabilities. Bitmap fuzzing operates by inputting malformed or randomly generated data into bitmap parsers to detect potential flaws that could otherwise remain hidden during traditional testing.

LibFuzzer and AFL are two widely used fuzzing tools, each employing unique strategies to identify software vulnerabilities. LibFuzzer works as an in-process fuzzer, repeatedly executing the target program with various inputs to explore diverse code paths, making it particularly effective for analyzing specific file formats by leveraging instrumentation for precise memory detection [5]. In contrast, AFL utilizes a coverage-guided approach, employing genetic algorithms to optimize test case generation based on feedback from code coverage, which enhances its efficiency in uncovering vulnerabilities across different applications [2~3]. Tools like FormatFuzzer build on LibFuzzer's format-specific capabilities, enabling it to handle structured inputs such as bitmaps effectively [1]. Similarly, AFL's versatility is augmented by innovations like CrFuzz, which improves its performance in multi-purpose programs by optimizing input validation and coverage, highlighting the complementary strengths of both fuzzers in addressing diverse fuzzing challenges [4].

This section discusses the significance of fuzz testing in enhancing software security. Fuzz testing is a dynamic approach that helps identify potential vulnerabilities by providing a program with random, unexpected, or malformed input data. It is particularly effective at uncovering hard-to-detect issues, such as memory corruption, buffer overflows, or improper input validation, that can be exploited by attackers. In the context of file parsers, especially for complex formats like bitmap images, fuzz testing is essential for discovering flaws in how data is handled. Malformed files, if not properly validated, can lead to critical vulnerabilities such as remote code execution or crashes, and fuzz testing helps to expose these issues by simulating a wide range of abnormal inputs.

Additionally, fuzz testing is valued for its scalability and efficiency. Unlike traditional manual testing methods, fuzzing can automatically generate large volumes of diverse test cases, covering edge scenarios that are difficult to anticipate. This scalability allows it to run continuously, providing comprehensive coverage and identifying issues faster than conventional approaches. By detecting vulnerabilities early in the development process, fuzz testing also reduces the cost and effort associated with strengthening security errors later on. As a result, fuzz testing has become an vital tool for software developers and security professionals, ensuring that applications are more strong to cyber threats and reducing the risk of security breaches.

This study provides a comparative analysis of LibFuzzer and AFL by examining their respective concerts in bitmap fuzzing. Through systematic testing, this study assesses which tool is more effective in uncovering vulnerabilities specific to bitmap processing, a crucial consideration in secure image-handling applications. The structure of this work is as follows: Section 1 underscores the importance of fuzz testing in security, Section 2 reviews related literature, Section 3 outlines the experimental setup, Section 4 presents the comparative findings, and Section 5 discusses conclusions and potential areas for future research.

II. LITERATURE SURVEY

This section reviews the related research works and tools in fuzzing.

Metzman, Jonathan, et al.FuzzBench provides an open platform for comparing the effectiveness of general-purpose fuzzers such as LibFuzzer, AFL, and honggfuzz. The study evaluates their ability to detect bugs and achieve code coverage across various applications. LibFuzzer excels in memory-related bugs, while honggfuzz leads in coverage metrics. This platform fosters standardized evaluation of fuzzing tools in different domains [6].

Kim et al. (2024) introduce AIMFuzz, an automated in-memory fuzzing tool designed for binary programs. It identifies and mutates critical memory regions and functions using dynamic taint tracking. AIMFuzz eliminates the need for manual intervention, improving efficiency and effectiveness in vulnerability detection. The tool demonstrates high success in detecting bugs within binary-only applications [7].

Liang and Hsiao (2021) propose icLibFuzzer, a version of LibFuzzer that isolates contexts to ensure consistent results. This approach resolves challenges in comparing fuzzing tools by reinitializing program states after crashes. Their experiments show icLibFuzzer achieves higher bug discovery rates and better coverage than competing tools. This research enhances the reliability of fuzzing in complex scenarios [8].

Zhu (2021) addresses challenges in fuzzing, such as resource efficiency and test prioritization. The study proposes synthesized benchmarks and optimized scheduling to improve bug detection and reduce overhead. These methods enhance fuzzing performance by focusing on critical areas of the code. This research provides a foundation for more effective vulnerability detection techniques [9].

Maier (2023) explores feedback-guided fuzzing to uncover vulnerabilities in untested software. By adapting inputs based on program feedback, this technique improves bug discovery rates. The study demonstrates its effectiveness in revealing flaws in complex applications. Feedback-driven fuzzing is shown to significantly strengthen software security [10].

Strassle (2024) explores advanced fuzzing frameworks to identify vulnerabilities in C and C++ source code. The study emphasizes the importance of high-performance fuzzing to address increasing software complexity and associated security risks. By focusing on real-world open-source projects, the research highlights how systematic fuzzing improves software robustness. This work provides a foundation for enhancing vulnerability detection in critical programming environments [11].

Frighetto (2019) introduces a framework combining REVNG and LLVM LibFuzzer for coverage-guided binary fuzzing. The study tackles the challenges of fuzzing binary-only software by leveraging static binary translation to enable instrumentation and

analysis. The proposed method achieves semantic preservation and effective bug discovery in executable programs. This approach underscores the potential of coverage-based fuzzing for securing memory-unsafe applications [12].

Fioraldi et al. (2023) provide an in-depth evaluation of American Fuzzy Lop (AFL) using the FuzzBench platform. Through nine experiments, the research examines AFL's mutation strategies, feedback encoding, and scheduling methods. The findings reveal how design choices influence bug detection and code coverage, offering insights into improving modern fuzzing tools. This study advocates for refining AFL's framework to enhance security testing practices [13].

Chafjiri, Sadegh Bamohabbat, et al. (2024) This research reviews the use of machine learning techniques in enhancing fuzzing for vulnerability detection. The paper highlights how machine learning can improve fuzzing efficiency by refining input generation and coverage prioritization. It shows that integrating machine learning with fuzzing techniques can significantly increase the discovery of security vulnerabilities [14].

De Almeida, FranciscoJoao Guimarães Coimbra (2019) This thesis focuses on methods for creating software tests that verify both functionality and the presence of flaws. It underscores the role of automated testing in boosting software reliability while shortening development cycles. The work advocates for systematic approaches to test generation to detect vulnerabilities early and ensure robust software development [15].

III. PROPOSED WORK FOR BITMAP FUZZING

This study compares the effectiveness of two widely used fuzzing tools, LibFuzzer and AFL, in identifying vulnerabilities in bitmap processing functions. By setting up both tools within a controlled environment, we analyze their abilities to generate varied inputs, detect crashes, and manage resource consumption. The goal is to understand each fuzzer's strengths and limitations by observing how they handle bitmap files, focusing on their efficiency, memory use, and bug detection capability. This section includes the following processes namely setup, testing, data collection, and comparative analysis steps, offering insights into which fuzzer is better suited for bitmap file testing.

Step 1: Setup Fuzzing Environment: Install and configure LibFuzzer and AFL on an Ubuntu server. Prepare a bitmap processing application as the fuzzing target.
Step 2: LibFuzzer Bitmap Fuzzing: Run LibFuzzer on bitmap processing functions, generating inputs and logging crashes, memory use, and crash details.
Step 3: AFL Bitmap Fuzzing: Instrument the target application with AFL, using a bitmap seed for mutations, and capture any crash data and memory/resource use.
Step 4: Data Logging and Monitoring: Log all inputs, crash data, and memory/CPU usage for both fuzzers to analyze efficiency and performance.
Step 5: Data Analysis: Compare each fuzzer on crash count, memory use, and code coverage; identity common vulnerabilities.

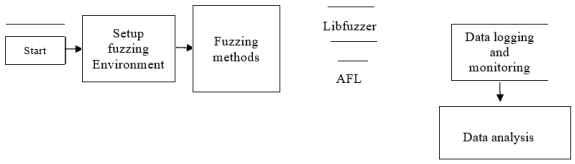

Fig. 1 Flowchart of bitmap fuzzing.

3.1 Bitmap fuzzing design:
3.1.1 Objective:
The goal is to assess the capabilities of two fuzzing tools, LibFuzzer and AFL, in testing the stability of bitmap processing functions. This includes detecting vulnerabilities and evaluating the performance of each tool in identifying errors.

3.1.2 System Overview:
The design involves two key components: the Fuzzing Engine and the Bitmap Processing Target. The fuzzing engines generate mutated bitmap files, which are then processed by the target application designed to simulate the processing of bitmap images.

LibFuzzer: An in-process tool that targets specific code paths in bitmap processing functions. AFL

(American Fuzzy Lop): A binary-level fuzzer that utilizes coverage feedback to guide input mutations.

Bitmap Processing Target: A simple application designed to load, parse, and manipulate bitmap files, acting as the vulnerable system being tested.

3.1.3 Input Generation:
Bitmap Seed Files: Both fuzzers begin with a set of initial bitmap images to mutate and generate new test cases.

Mutation Techniques: These include altering byte sequences, flipping bits, and introducing random data changes to stress the bitmap processing functions.

3.1.4 Fuzzing Execution:
LibFuzzer:
1) Uses internal feedback to mutate the bitmap data and direct testing toward different execution paths in the target code.
2) The mutated data is processed using a function like LLVMFuzzerTestOneInput. AFL:
1) Relies on coverage-guided fuzzing, where it generates bitmap variations and uses execution feedback to guide the process.
2) Logs any crashes or unusual behavior during execution, providing a record of errors for later analysis.

3.1.5 Crash Detection:
Both fuzzers capture crashes such as buffer overflows, invalid memory access, or other runtime exceptions.
All relevant details, including the inputs, error logs, and system resource usage, are recorded for further review.

3.1.6 Performance Evaluation:
The performance evaluation compares LibFuzzer and AFL based on several metrics, including crash detection, code coverage, execution speed, and memory usage. Crash detection measures how many vulnerabilities each tool uncovers. Code coverage evaluates how thoroughly each tool tests the bitmap processing functions. Execution speed and memory usage determine how efficient and resource-intensive each fuzzer is during testing.

3.1.6 Data Analysis:
Data analysis involves comparing the results from LibFuzzer and AFL to assess their effectiveness. The types of vulnerabilities found by each tool are examined to identify which fuzzer performs better. Code coverage is analyzed to see how much of the bitmap processing code is tested. Insights gained from the comparison help identify improvements for the fuzzing process.

3.2 Bitmap fuzzing Architecture:
In this section input Generation phase, a diverse set of test cases is created to challenge the bitmap processing system. This is achieved using two approaches: creating random inputs from scratch and modifying valid bitmap files. Generating inputs from scratch allows the system to encounter highly unpredictable data, testing its resilience to unexpected structures. Modifying existing bitmap files, on the other hand, provides slight alterations of valid files, which simulates more realistic but varied scenarios, helping to identify vulnerabilities that might arise under real-world usage.

Once inputs are generated, they are passed through two popular Fuzzing Tools: LibFuzzer and AFL. LibFuzzer applies coverage-guided fuzzing, which tracks code coverage and adjusts inputs dynamically to explore unexplored code paths, enhancing the chance of finding hidden vulnerabilities. AFL (American Fuzzy Lop) uses a genetic mutation-based approach to create diverse inputs that explore different states of the program. The combination of LibFuzzer's coverage-guided approach and AFL's mutation strategy ensures a broad exploration of the system's capabilities and limitations.