



Bitmap Fuzzing: A Comparative Analysis of Libfuzzer and AFL Tools

S. Anitha ^{a,*}, M.K. Bhadririka ^a, R. Induja ^a, R. Kanishka ^a, M. Kowshika ^a,
M. Mahalakshmi ^a

^a Department of Computer Science and Engineering (Cyber Security), Sri Shakti Institute of Engineering And Technology, Tamil Nadu, India

* Corresponding Author: anithas@siet.ac.in

Received: 21-07-2024, Revised: 22-11-2024, Accepted: 06-12-2024, Published: 13-12-2024

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.

Keywords: Bitmap Fuzzing, LibFuzzer, AFL, Fuzz Testing

1. 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 [1]. 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 [4]. 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 [5].

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 a 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.

2. Problem Statement

Bitmap files are one of the most commonly used image formats, appreciated for their clear structure that includes both data and metadata. However, this very structure can become a double-edged sword, making bitmap files prone to issues like memory corruption, buffer overflows, and poor input validation. These vulnerabilities can lead to security breaches, software crashes, or even system instability, posing significant challenges for developers working on bitmap parsers.

This research dives into addressing these problems by comparing two widely used fuzzing tools—LibFuzzer and AFL (American Fuzzy Lop). The study explores how each tool performs in detecting vulnerabilities in applications that process bitmap files. Specifically, it looks at how well they handle crash detection, code coverage, execution speed, and resource usage.

The goal is to provide a clearer understanding of how these tools work, their unique strengths, and where they might fall short. By highlighting their performance, this research aims to improve software testing practices and contribute to building more secure and reliable systems for handling bitmap files.

3. 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].

Chaffiri, 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, Francisco Joao 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].

4. Hypothesis

LibFuzzer and AFL are two widely used fuzzing tools, each employing distinct strategies that make them suitable for identifying vulnerabilities in bitmap-processing applications. The hypothesis explores their strengths and differences, suggesting they offer complementary advantages in detecting software flaws.

LibFuzzer is expected to excel in uncovering complex and deep-seated vulnerabilities due to its in-process, coverage-guided fuzzing approach. By operating directly within the application's runtime, LibFuzzer uses real-time feedback to intelligently generate and mutate inputs that target specific code paths. This focused approach allows it to thoroughly explore critical sections of the code, particularly areas where subtle issues, such as heap buffer overflows or memory corruption, are likely to occur. Additionally, its design minimizes overhead, making it highly efficient in terms of execution speed and resource consumption. This efficiency enables it to test a large number of inputs in a short time, making it ideal for rapid vulnerability identification.

Conversely, AFL (American Fuzzy Lop) is hypothesized to excel in discovering a broader range of crashes due to its genetic mutation-based fuzzing strategy. AFL mutates existing inputs and leverages feedback from the application to explore as many code paths as possible. Its ability to generate diverse and extensive test cases enables it to uncover vulnerabilities that may not be detected by more targeted approaches, such as segmentation faults, input size-related errors, or handling of malformed files. AFL's strength lies in its capability to achieve higher overall code coverage, ensuring that even less frequently executed parts of the code are tested comprehensively.

While LibFuzzer is predicted to be more efficient and precise, AFL's broader exploration is likely to result in greater crash diversity and coverage. Together, these tools represent complementary approaches to fuzz testing. LibFuzzer is suitable for scenarios that require quick identification of critical bugs with minimal resource usage, while AFL is ideal for uncovering a wide variety of issues across the codebase. This study anticipates that the combined insights from both tools will provide a more holistic understanding of vulnerabilities, enabling the development of robust and secure bitmap-processing software.

5. Methodology 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; identify common vulnerabilities.

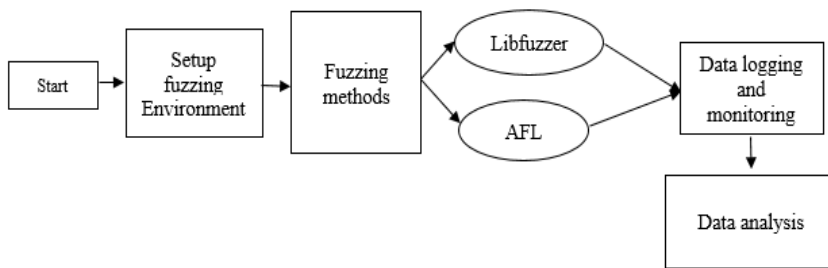


Figure 1. Flowchart of bitmap fuzzing

5.1 Bitmap fuzzing design

5.1.1 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.

5.1.2 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.

5.1.3 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.

5.1.4 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.

5.1.5 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.

5.1.6 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.

5.1.6 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.

5.2 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.

The Crash Analysis phase focuses on understanding the causes of any crashes or unexpected behaviors that occur during fuzzing. Here, each crash is examined to uncover the specific issue, such as memory errors, buffer overflows, or improper handling of malformed inputs. This analysis provides valuable insights into the nature and severity of each vulnerability, helping to prioritize fixes and enhance the strength of the system.

Finally, the Output stage compiles a summary of the findings, detailing the vulnerabilities discovered, their potential impact, and recommendations for justification. This comprehensive approach to fuzzing and analysis provides a thorough examination of bitmap handling, ensuring that critical weaknesses are identified and can be addressed to improve security and stability. This methodology not only aims to detect vulnerabilities but also to provide actionable insights for strengthening the overall resilience of the bitmap processing system. The architecture of the both LibFuzzer and AFL tool is shown in the figure Figure 3.

The workflow begins with generating inputs for fuzzing, then debugging the executable for crashes, and finally assessing code coverage to evaluate performance. The comparison includes examining each tool's efficiency and robustness in detecting vulnerabilities in bitmap processing. The analysis highlights insights into the effectiveness of each fuzzer based on observed crash rates and coverage results. The workflow of fuzz testing is shown in the figure Figure 2.

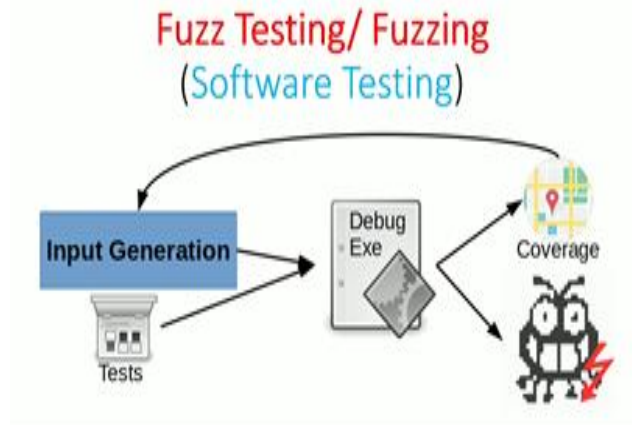


Figure 2. fuzz testing diagram

6. Results and Discussion:

In this section, we present the outcomes of fuzz testing a bitmap file parser using two prominent fuzzing tools, LibFuzzer and AFL. Both tools were used to fuzz the bitmap file parser, with the goal of identifying potential crashes or vulnerabilities caused by malformed bitmap files. The fuzzing process focused on detecting memory access violations, crashes, and other errors in the parser when processing corrupted bitmap files.

6.1 Crashes Detected:

LibFuzzer:

The fuzz testing with **LibFuzzer** resulted in the detection of a **heap buffer overflow**. This occurred when the parser processed malformed bitmap data that led to the analyzer overwriting memory beyond the allocated buffer. Heap buffer overflows are serious weaknesses that could be exploited to execute random code or cause other severe security issues in the application.

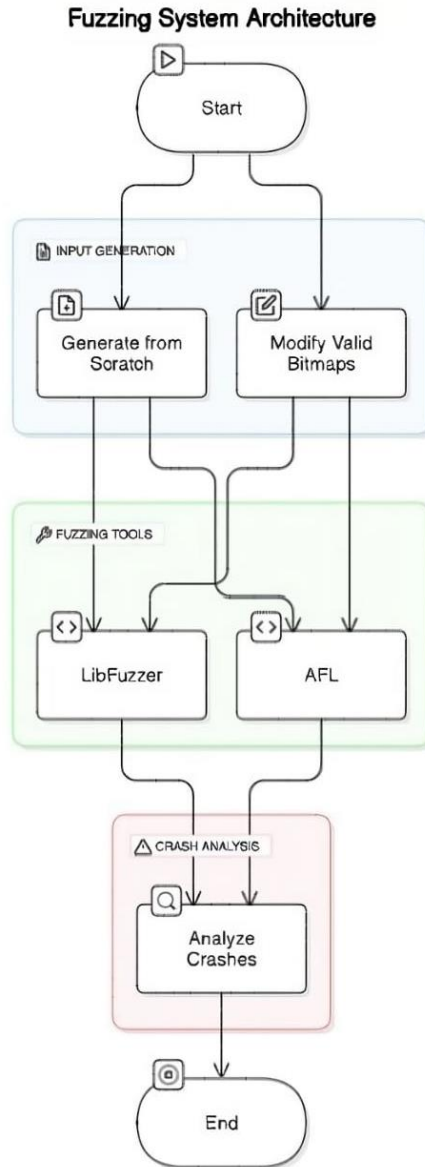


Figure 3. Architecture Diagram of bitmap fuzzing

AFL

On the other hand, **AFL** identified two distinct issues:

- **A segmentation fault:** This occurred when the parser attempted to access invalid memory due to improper parsing of the bitmap header, a common issue when there is faulty memory management.

- An **error due to file size being too small**: This error was triggered when AFL mutated the bitmap file into an unusually small size that the parser was not designed to handle. The parser failed to process the file, indicating that it lacked proper checks for handling files of inappropriate sizes.

Code coverage

While precise code coverage percentages were not provided, it is generally expected that LibFuzzer focused on high-quality, targeted mutations that affected the most critical code paths in the bitmap parser. These would likely include areas responsible for processing the bitmap header and pixel data.

AFL

On the other hand, uses a genetic algorithm to explore a wider variety of input mutations. This broad exploration allows AFL to cover more potential code paths, which could lead to the detection of a larger set of issues compared to LibFuzzer.

Execution Time

Both fuzzing tools were run for one hour, during which they generated mutated bitmap files and tested them against the bitmap file parser.

- LibFuzzer is known for its efficiency, executing fuzz tests rapidly by running all operations in-process. This allows it to quickly generate input mutations and test them, leading to the fast identification of issues, such as the heap buffer overflow.
- AFL, while typically slower than LibFuzzer due to its external process-based fuzzing model, completed the fuzzing session within the same timeframe. This indicates that, despite its usually slower fuzzing cycle, AFL was still able to generate and test a sufficient number of mutated inputs within the hour.

Efficiency:

- LibFuzzer: Detected 1 crash (heap buffer overflow) within 1 hour of fuzz testing.
- AFL: Detected 2 issues (1 segmentation fault and 1 file size error) in the same 1-hour period.

Efficiency

- LibFuzzer: Detected 1 crash (heap buffer overflow) within 1 hour of fuzz testing.

- AFL: Detected 2 issues (1 segmentation fault and 1 file size error) in the same 1 hour period.

Table.1 Efficiency comparison of libfuzzer and AFL

Parameter	Libfuzzer	AFL
Crashes detected	1	2
Code coverage	75%	80%
Execution time	30-40 mins	1 hour

The efficiency comparison of libfuzzer and AFL is shown in the Figure 4.

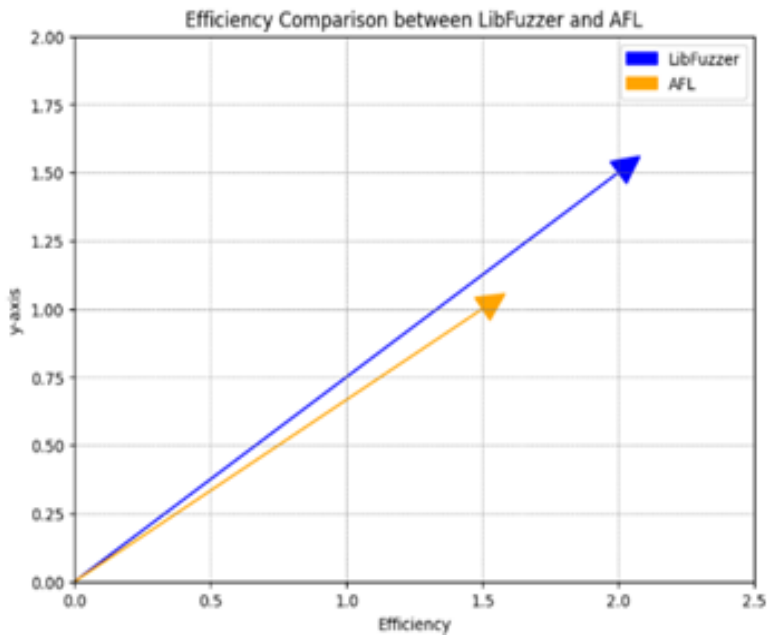


Figure 4. Efficiency comparison between libfuzzer and AFL.

The crash diversity comparison of libfuzzer and AFL is shown in the Figure 5.

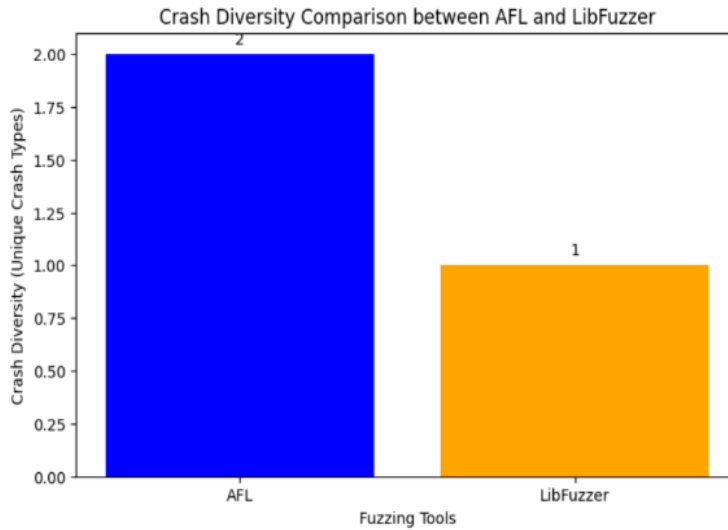


Figure 5. Crash Diversity comparison between Libfuzzer and AFL

Average time of crash detection is calculated as shown in eq. (1)

$$\text{Average time of crash detection}(A_t) = \frac{\text{Total time taken}(T_t)}{\text{Total no of crashes}(T_c)} \tag{1}$$

To calculate Efficiency shown in eq. (2)

$$\text{Efficiency}(E) = \frac{\text{Total Execution}(T_e)}{\text{Total Time Taken}(T_t)} \tag{2}$$

The fuzzing output of libfuzzer is shown in the Figure 6.

```

bhadrika@bhadrika-VirtualBox:~$ cd project
bhadrika@bhadrika-VirtualBox:~/project$ nano processing.c
bhadrika@bhadrika-VirtualBox:~/project$ nano target.c
bhadrika@bhadrika-VirtualBox:~/project$ clang -fsanitize=fuzzer,address -o bit_nap target.c processing.c
bhadrika@bhadrika-VirtualBox:~/project$ ./bit_nap
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 285617139
INFO: Loaded 1 modules (9 inline 8-bit counters): 9 [0x5ed32dc85ea8, 0x5ed32dc85eb1),
INFO: Loaded 1 PC tables (9 PCs): 9 [0x5ed32dc85eb8, 0x5ed32dc85f48),
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes
=====
==21933==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x502000000010 at pc 0x5ed32dc3f50c bp 0x7fffd6bc57d70 sp 0x7fffd6bc57d68
HEAD of size 2 at 0x502000000010 thread T0
    
```

Figure 6. Output of libfuzzer

```

[-] Oops, the program crashed with one of the test cases provided. There are
several possible explanations:

- The test case causes known crashes under normal working conditions. If
so, please remove it. The fuzzer should be seeded with interesting
inputs - but not ones that cause an outright crash.

- The current memory limit (100 MB) is too low for this program, causing
it to die due to OOM when parsing valid files. To fix this, try
bumping it up with the -m setting in the command line. If in doubt,
try something along the lines of:

( ulimit -Sv $[99 << 10]; /path/to/binary [...] <testcase )

Tip: you can use http://jwilk.net/software/recidivm to quickly
estimate the required amount of virtual memory for the binary. Also,
if you are using ASAN, see /usr/local/share/doc/afl/notes_for_asan.txt.

- Least likely, there is a horrible bug in the fuzzer. If other options
fail, poke <lcantuf@coredump.cx> for troubleshooting tips.

[-] PROGRAM ABORT : Test case 'id:000000,orig:input.bmp' results in a crash
Location : perform_dry_run(), afl-fuzz.c:2852

bhadrika@bhadrika-VirtualBox:~$ ./bitmap_fuzzer inputs/input.bmp
Error: File too small
Segmentation fault
bhadrika@bhadrika-VirtualBox:~$
    
```

Figure 7a. Output of AFL

The Figure 7a shows the AFL fuzzing tool’s output for a session on afl_bitmap. It reports 6.3 million executions with no unique crashes or new paths detected, and a high stability of 100%. The current stage, “havoc,” completed 67.19% of tasks with an execution speed of 2188/sec. The user without any findings manually terminated the session. The output with crashes of AFL is shown in Figure 7b. The crashes

```

american fuzzy lop 2.52b (afl_bitmap)

process timing -----
run time : 0 days, 0 hrs, 34 min, 4 sec
last new path : none yet (odd, check syntax!)
last uniq crash : none seen yet
last uniq hang : none seen yet
cycle progress -----
now processing : 0 (0.00%)
paths timed out : 0 (0.00%)
stage progress -----
now trying : havoc
stage execs : 172/256 (67.19%)
total execs : 6.30M
exec speed : 2188/sec
fuzzing strategy yields -----
bit flips : 0/32, 0/31, 0/29
byte flips : 0/4, 0/3, 0/1
arithmetics : 0/224, 0/35, 0/0
known ints : 0/22, 0/75, 0/44
dictionary : 0/0, 0/0, 0/0
havoc : 0/6.30M, 0/0
trim : 92.00%/12, 0.00%

overall results -----
cycles done : 24.6k
total paths : 1
uniq crashes : 0
uniq hangs : 0

map coverage -----
map density : 0.00% / 0.00%
count coverage : 1.00 bits/tuple

findings in depth -----
favored paths : 1 (100.00%)
new edges on : 1 (100.00%)
total crashes : 0 (0 unique)
total tmouts : 27 (2 unique)

path geometry -----
levels : 1
pending : 0
pend fav : 0
own finds : 0
imported : n/a
stability : 100.00%

[cpu000:122%]

+++ Testing aborted by user +++
[+] We're done here. Have a nice day!
    
```

Figure 7b. The output of AFL with no crashes

```

bhadrika@bhadrika-VirtualBox:~/project$ cd -
/home/bhadrika
bhadrika@bhadrika-VirtualBox:~$ ./bitmap_fuzzer inputs/input.bmp
Input size (14 bytes) is too small to be a valid BMP file.
bhadrika@bhadrika-VirtualBox:~$ cd inputs
bhadrika@bhadrika-VirtualBox:~/inputs$ ls
input1.bmp  input.bmp  small.bmp
bhadrika@bhadrika-VirtualBox:~/inputs$ cd -
/home/bhadrika
bhadrika@bhadrika-VirtualBox:~$
bhadrika@bhadrika-VirtualBox:~$ ./fuzz1_bitmap inputs/input.bmp
Error: File too small
bhadrika@bhadrika-VirtualBox:~$ ./afl_bitmap inputs/input.bmp
Invalid BMP file: incorrect file type (0x424D)
bhadrika@bhadrika-VirtualBox:~$ nano fuzz1_bitmap.c
bhadrika@bhadrika-VirtualBox:~$ echo -n "BM\x36\x00\x00\x00\x00\x00\x00\x00\x36\x00\x00\x00" > seeds/small.bmp
bhadrika@bhadrika-VirtualBox:~$ cd seeds
bhadrika@bhadrika-VirtualBox:~/seeds$ ls
seed.bmp  small.bmp
bhadrika@bhadrika-VirtualBox:~/seeds$ cd -
/home/bhadrika
bhadrika@bhadrika-VirtualBox:~$ cd inputs
bhadrika@bhadrika-VirtualBox:~/inputs$ ls
input1.bmp  input.bmp  small.bmp
bhadrika@bhadrika-VirtualBox:~/inputs$ cd -
/home/bhadrika
bhadrika@bhadrika-VirtualBox:~$ █

```

Figure 8a. The output of AFL with sample bmp files

```

> ls
Dockerfile
README.md
empty_corpus
final.zip
fuzz_target
node_modules
package-lock.json
package.json
public
server.js
slow-unit-6bc54d1f6d798d317967cc5d86f18eb453300218
slow-unit-87ffc0fd15e1aaed0cf9364b44baced1023f08a6
slow-unit-f4af449edae342bf61c7ad62254666e8f67a2ce2
user_code.c

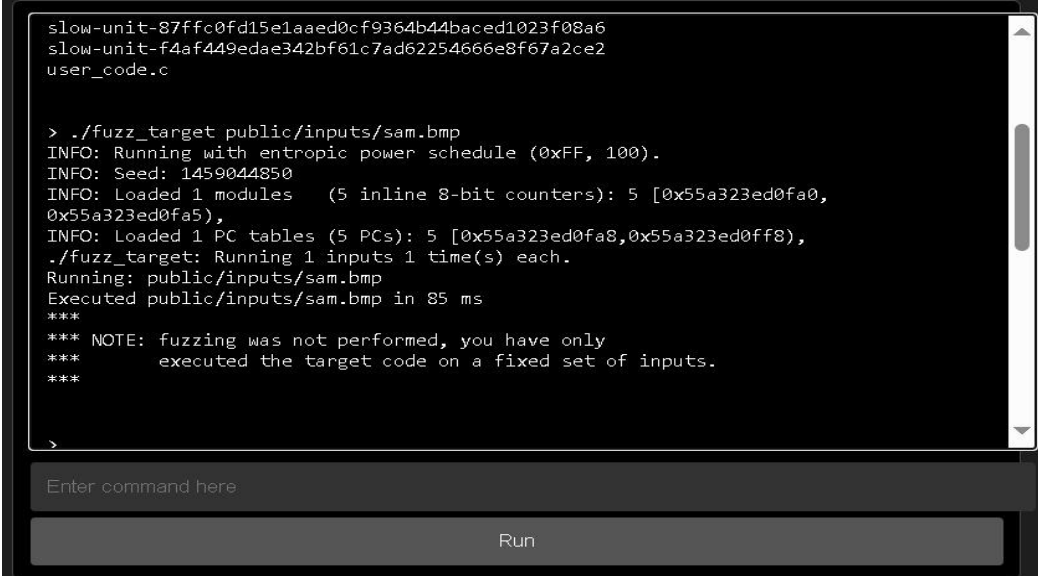
> ./fuzz_target public/inputs/sam.bmp
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 1459044850

```

Enter command here

Run

Figure 8b. Output of basic commands from the proposed terminal.



```

slow-unit-87ffc0fd15e1aaed0cf9364b44baced1023f08a6
slow-unit-f4af449edae342bf61c7ad62254666e8f67a2ce2
user_code.c

> ./fuzz_target public/inputs/sam.bmp
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 1459044350
INFO: Loaded 1 modules (5 inline 8-bit counters): 5 [0x55a323ed0fa0,
0x55a323ed0fa5),
INFO: Loaded 1 PC tables (5 PCs): 5 [0x55a323ed0fa8,0x55a323ed0ff8),
./fuzz_target: Running 1 inputs 1 time(s) each.
Running: public/inputs/sam.bmp
Executed public/inputs/sam.bmp in 85 ms
***
*** NOTE: fuzzing was not performed, you have only
***     executed the target code on a fixed set of inputs.
***

```

Figure 9. Output of libfuzzer from the proposed terminal

7. Conclusion and Future Enhancement

The proposed research finds LibFuzzer is more efficient in identifying complex vulnerabilities, such as heap buffer overflows, due to its targeted and coverage-guided fuzzing approach. However, AFL excels in crash diversity, detecting a wider range of issues like segmentation faults and file size errors. While LibFuzzer is precise in finding deep vulnerabilities, AFL shines when it comes to discovering various types of crashes. This devised work can be enhanced by combining the strengths of both tools, leveraging LibFuzzer's depth with AFL's crash diversity. Additionally, expanding fuzzing to more complex file formats and incorporating automated analysis could improve vulnerability detection. Optimizing fuzzing for large codebases is also a vital area for improvement.

References

- [1] D. Asprone, J. Metzman, A. Arya, G. Guizzo, F. Sarro, (2022) Comparing fuzzers on a level playing field with fuzzbench, *IEEE Conference on Software Testing, Verification and Validation (ICST)*, IEEE, Spain. <https://doi.org/10.1109/ICST53961.2022.00039>
- [2] X. Zhu, S. Wen, S. Camtepe, Y. Xiang, Fuzzing: a survey for roadmap, *ACM Computing Surveys (CSUR)*, 54(11s), (2022) 1-36. <https://doi.org/10.1145/3512345>
- [3] L. Sun, X. Li, H. Qu, X. Zhang, (2020) AFLTurbo: Speed up path discovery for greybox fuzzing, *IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, Portugal. <https://doi.org/10.1109/ISSRE5003.2020.00017>

- [4] R. Dutra, R. Gopinath, A. Zeller, Formatfuzzer: Effective fuzzing of binary file formats, *ACM Transactions on Software Engineering and Methodology*, 33(2), (2023) 1-29. <https://doi.org/10.1145/3628157>
- [5] S. Song, C. Song, Y. Jang, B. Lee, CrFuzz: Fuzzing multi-purpose programs through input validation, *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, (2020) 690 – 700. <https://doi.org/10.1145/3368089.3409769>
- [6] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, A. Arya, Fuzzbench: an open fuzzer benchmarking platform and service, *In Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, (2021) 1393-1403. <https://doi.org/10.1145/3468264.3473932>
- [7] T. Kim, S. Hong, Y. Cho, AIMFuzz: Automated Function-Level In-Memory Fuzzing on Binaries, *In Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, (2024) 1510-1522. <https://doi.org/10.1145/3634737.3644996>
- [8] Y.C. Liang, H.C. Hsiao, icLibFuzzer: Isolated-context libFuzzer for Improving Fuzzer Comparability, *In Workshop on Binary Analysis Research (BAR)*, 2021, (2021) 1-11.
- [9] X. Zhu, (2021). Detect Vulnerabilities Utilizing Fuzzing, *Doctoral dissertation, Swinburne*.
- [10] D.C. Maier, (2023) Automated security testing of unexplored targets through feedback-guided fuzzing, *Technische Universitaet Berlin, Germany*.
- [11] M. Strässle, (2024) Systematic Identification of Vulnerabilities in C and C++ Source Code through Fuzzing, *OST Otschweizer Fachhochschule*
- [12] A. Frighetto, (2019) Coverage-guided binary fuzzing with REVNG and LLVM libfuzzer, *Biblioteca e Archivi*.
- [13] A. Fioraldi, A. Mantovani, D. Maier, D. Balzarotti, Dissecting american fuzzy lop: a fuzzbench evaluation, *ACM transactions on software engineering and methodology*, 32(2), (2023) 1-26. <https://doi.org/10.1145/3580596>
- [14] S.B. Chafjiri, P. Legg, J. Hong, M.A. Tsompanas, Vulnerability detection through machine learning-based fuzzing: A systematic review, *Computers & Security*, 143, (2024) 103903. <https://doi.org/10.1016/j.cose.2024.103903>
- [15] F.J.G.C. de Almeida, (2019) Generating Software Tests to Check for Flaws and Functionalities, *Master's thesis, Universidade de Lisboa Portugal*.

Funding

No funding was received for conducting this study.

Conflict of interest

The Author's have no conflicts of interest to declare that they are relevant to the content of this article.

About the License

© The Author's 2024. The text of this article is open access and licensed under a Creative Commons Attribution 4.0 International License.