IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 12, December 2025

DOI: 10.17148/IJARCCE.2025.1412110

Improving Open Source Files Security Using Fuzzing

Dr. Puneeth GJ[1], Amruta MM[2], B Susheela[3], Bharathi H K[4], Harikiran CS[5]

Associate Professor, Department of Computer Science and Engineering,

Rao Bahadur Y Mahabaleswarappa Engineering College, Ballari, VTU, Karnataka, India[1]

Student, Department of Computer Science and Engineering, Rao Bahadur Y Mahabaleswarappa Engineering College, Ballari, VTU, Karnataka, India[2]

Student, Department of Computer Science and Engineering, Rao Bahadur Y Mahabaleswarappa Engineering College, Ballari, VTU, Karnataka, India[3]

Student, Department of Computer Science and Engineering, Rao Bahadur Y Mahabaleswarappa Engineering College, Ballari, VTU, Karnataka, India[4]

Student, Department of Computer Science and Engineering, Rao Bahadur Y Mahabaleswarappa Engineering College, Ballari, VTU, Karnataka, India[5]

Abstract: Open-source software is extensively used in modern systems due to its flexibility and cost efficiency; however, it often contains hidden security vulnerabilities that traditional testing methods may fail to detect. Fuzz testing is an automated technique that addresses this challenge by supplying programs with random and malformed inputs to uncover crashes and weaknesses.

This paper presents a web-based system that demonstrates how fuzzing improves the security of open-source files. The system allows users to upload single files, multiple files, or compressed archives and simulates the processes of building, instrumentation, and fuzzing. Security improvements are analyzed using metrics such as code coverage, crash detection, vulnerability count, and overall security score. A comparative evaluation is performed to highlight the difference in software robustness before and after fuzzing.

The proposed system integrates an interactive frontend with a FastAPI-based backend to provide real-time progress visualization and automated result reporting. The results indicate that fuzzing significantly enhances the stability and security of open-source files, emphasizing its effectiveness as a proactive software security testing approach.

Keywords: Fuzz Testing, Open Source Software Security, Automated Vulnerability Detection, Software Testing, File Security.

I. INTRODUCTION

Open-source software is widely used in modern applications, but it often contains hidden vulnerabilities that traditional testing methods fail to detect. Fuzz testing is an automated approach that improves software security by providing random and unexpected inputs to uncover crashes and weaknesses. This project demonstrates a web-based fuzzing system that allows users to upload open-source files and analyze security improvements through simulated fuzzing. The system highlights how fuzzing enhances software reliability and strengthens open-source file security**.**

Open-source software has become an integral component of modern computing environments because of its flexibility, transparency, collaborative development model, and cost-effectiveness. Today, a significant portion of software applications, libraries, frameworks, and system tools used across industries rely heavily on open-source code. While this widespread adoption accelerates innovation and reduces development costs, it also introduces serious security concerns.

Open-source projects are often maintained by distributed communities with varying levels of expertise, and not all code is subjected to rigorous security testing before deployment. As a result, vulnerabilities such as buffer overflows, memory leaks, improper input handling, and logic flaws may remain hidden within files and libraries, posing risks to systems that depend on them. Traditional testing approaches, including manual code inspection and predefined test cases, are limited in their ability to uncover such issues. Manual reviews are time-consuming and prone to human error, while predefined

DOI: 10.17148/IJARCCE.2025.1412110

tests only validate expected behavior and often fail to explore unexpected or malicious input scenarios. This creates gaps in security coverage, especially when dealing with complex or large-scale open-source codebases.

To overcome these limitations, fuzz testing plays a critical role in modern software security. Fuzzing is an automated testing technique that continuously supplies programs with random, malformed, or unexpected inputs to trigger abnormal behavior, crashes, or security flaws that are difficult to detect through conventional testing methods. By aggressively exploring edge cases and unanticipated execution paths, fuzzing helps identify vulnerabilities at an early stage of development. The purpose of the proposed system is to demonstrate the effectiveness of fuzzing in improving the security of open-source files through a web-based simulation platform. The system allows users to upload files, simulate fuzzing operations, and observe how automated testing helps detect potential vulnerabilities and improve software robustness.

Overall, this project serves as both an educational and practical tool, highlighting the importance of fuzz testing in strengthening open-source software security and promoting safer software development practices.

The main objective of this project, Improving Open Source Files Security Using Fuzzing, is to design and implement a web-based simulation platform that demonstrates how fuzzing helps identify and resolve security vulnerabilities in open-source code.

[1]. To develop a user-friendly interface that allows users to upload open-source files (in formats like .c, .cpp, .py, .java, .zip, etc.).
[2]. To simulate the build and instrumentation process, showing how files are prepared for fuzz testing.
[3]. To perform a virtual fuzzing process, displaying progress bars, logs, and test cases being executed.
[4]. To visualize the results before and after fuzzing, showing improvements in metrics such as code coverage, number of crashes, vulnerabilities found, and overall security score.
[5]. To generate a downloadable PDF report summarizing the testing process and results for documentation and academic submission.
[6]. 6. To provide an educational and demonstrative tool that helps students and developers understand the concept of fuzz testing without requiring complex software installations.

Ultimately, this project aims to make fuzzing interactive, understandable, and educational, bridging the gap between theoretical learning and practical understanding of software security testing.

II. LITERATURE SURVEY

1. Chen et al. (2018 / USENIX Security 2019)EnFuzz:
Ensemble Fuzzing with Seed SynchronizationProposes EnFuzz, an ensemble fuzzing framework that combines multiple greybox fuzzers (such as AFL, FairFuzz, and ALGo). It introduces a globally-asynchronous locally-synchronous (GALS) seed-sharing mechanism to improve collaboration between fuzzers. The paper defines fuzzer diversity along three dimensions: coverage granularity, input mutation strategies, and seed selection policies. Evaluated on LAVA-M and Google's test suite, EnFuzz achieves up to 117% more crashes, ~39% more execution paths, and ~21% higher branch coverage than individual fuzzers.

2. Ognawala et al. (2019)Wildfire:
Compositional Fuzzing Aided by Targeted Symbolic ExecutionIntroduces Wildfire, a compositional fuzzing framework that first fuzzes isolated functions rather than entire programs. It then applies targeted symbolic execution to check feasibility and exploitability of discovered issues. Tested on 23 C/C++ projects, Wildfire significantly outperforms monolithic fuzzers in coverage and efficiency while using only ~10% of their runtime, demonstrating the effectiveness of function-level fuzzing.

3. Ding & Le Goues (2021)An Empirical Study of OSS-Fuzz Bugs:
Presents the first large-scale empirical study of Google's OSS-Fuzz platform, analyzing 23,907 bugs across 316 open-source projects. The study shows that continuous fuzzing surfaces bugs rapidly and that developers tend to patch them quickly. However, bug discovery often occurs in bursts. The paper highlights practical issues such as flaky bugs, timeouts, and out-of-memory errors, and observes low CVE assignment rates even for critical vulnerabilities.

4. Manès et al. (2018 / ACM Computing Surveys)The Art, Science, and Engineering of Fuzzing:
A SurveyProvides a comprehensive taxonomy and unified model of fuzzing, covering input generation, instrumentation, monitoring, and bug detection. Surveys major fuzzing paradigms including white-box, grey-box, black-box, grammar-based, and hybrid fuzzers. The paper highlights key innovations such as hybrid fuzzing, grammar inference, and large-scale fuzzing orchestration, serving as a foundational reference for fuzzing research.

5. Huang et al. (2024)Large Language Models Based Fuzzing Techniques:
A Survey Systematically surveys fuzzing workflows enhanced by Large Language Models (LLMs). Reviews approaches where LLMs are used to generate test inputs, driver code, grammars, or mutation strategies. The paper discusses how

International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 12, December 2025
DOI: 10.17148/IJARCCE.2025.1412110

LLMs improve automation, semantic awareness, and coverage, and outlines challenges such as hallucinations, cost, and integration with feedback-driven fuzzing loops.

6. [Authors Not Specified] (ACM Computing Surveys, 202?)Fuzzers for Stateful Systems:

A Survey Examines fuzzing techniques for stateful systems such as network services and APIs. Reviews tools including RESTler, SPFuzz, and EPF, which leverage grammar specifications and coverage feedback. RESTler uses OpenAPI specifications to generate valid request sequences, while SPFuzz and EPF integrate AFL-style feedback. Identifies open challenges including protocol grammar extraction, state explosion, and session dependency handling.

7. Mansuri et al. (2024)Where to Fuzz? Target Selection in Directed Fuzzing:

A Systematization of Knowledge (SoK) on target selection strategies in directed fuzzing. Categorizes approaches based on static analysis, symbolic execution hints, and feedback-driven prioritization. Highlights heuristics such as hot-spot prioritization and vulnerability seeding, emphasizing the importance of selecting effective targets to guide fuzzing toward deep or vulnerable code regions.

8. Zhang et al. (2024)LLAMAFUZZ:

Large Language Model Enhanced Greybox FuzzingIntroduces LLAMAFUZZ, a greybox fuzzer enhanced with LLM-based structured input mutation. The model is fine-tuned on seed–mutation pairs to learn input grammars and effective mutation strategies. Evaluated on Magma and real-world programs, LLAMAFUZZ discovers ~41 more bugs on average and improves branch coverage by ~27% compared to AFL++.

9. Xia et al. (2024)Fuzz4All:

Universal Fuzzing with Large Language ModelsPresents Fuzz4All, the first universal LLM-based fuzzer capable of handling multiple input languages without manual grammar engineering. Uses auto-prompting and iterative prompting to generate realistic and diverse inputs. The system discovers 98 bugs across nine major targets, including GCC, Clang, Z3, and Qiskit, demonstrating strong cross-domain adaptability.

F. Consolidated Summary of Related Works

Table I — Comparative Review of Related Research Works

Author(s) Year	Title	Methodology	Accomplished	Limitations / Future Work
Chen et al., 2018	EnFuzz: Ensemble Fuzzing with Seed Synchronization	Ensemble of grey-box fuzzers with seed synchronization	Improved crashes and coverage over individual fuzzers	Overhead of ensemble coordination
Ognawala et al., 2019	Wildfire: Compositional Fuzzing	Function-level fuzzing with symbolic execution	Found more bugs with less runtime	Scalability issues for large programs
Ding & Le Goues, 2021	An Empirical Study of OSS-Fuzz Bugs	Empirical analysis of OSS-Fuzz bug data	Identified bug patterns and patch trends	Flaky bugs; low CVE reporting
Manès et al., 2018	The Art, Science, and Engineering of Fuzzing	Survey and taxonomy of fuzzing techniques	Unified fuzzing concepts	Requires frequent updates
Huang et al., 2024	LLM Based Fuzzing Techniques	Survey of LLM-assisted fuzzing	Improved automation and coverage	High computation cost
ACM SUR, 202?	Fuzzers for Stateful Systems	Survey of stateful fuzzing tools	Identified API fuzzing challenges	State explosion issues
Mansuri et al., 2024	Where to Fuzz?	Directed fuzzing target selection	Improved prioritization strategies	Accuracy of target selection
Zhang et al., 2024	LLAMAFUZZ	LLM-guided grey-box fuzzing	Found more bugs than AFL++	Training overhead
Xia et al., 2024	Fuzz4All	Universal LLM-based fuzzing	Cross-domain bug discovery	Prompt sensitivity

III. SYSTEM ARCHITECTURE AND DESIGN

The proposed system follows a web-based client–server architecture. The design separates the user interface (frontend) from the processing logic (backend), ensuring scalability, modularity, and efficient performance.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 12, December 2025

DOI: 10.17148/IJARCCE.2025.1412110

Architecture Overview:

1. **Frontend (Client Side):**
- Developed using HTML, CSS (Tailwind CSS), and JavaScript.
- Allows users to upload open-source files, view progress of build and fuzzing simulation, and visualize the security improvements through charts and summaries.
- Communicates with the backend using HTTP requests (RESTful API calls).
2. **Backend (Server Side):**
- Implemented using Python's FastAPI framework.
- Handles core functionalities like file upload, build process simulation, fuzzing analysis, and result generation.
- Sends JSON responses back to the frontend for dynamic updates.
- Also supports generation of PDF reports using jsPDF (triggered from the frontend).
3. **Storage:**
- Uploaded files are temporarily saved in a local directory on the server.
- Results are computed dynamically and can be stored for future reference.
- Though the current version uses temporary storage, it can easily be upgraded to use a database like SQLite or MySQL for persistent storage.
4. **Communication:**
- The frontend and backend communicate via HTTP (REST API) calls.
- Example:
- POST /upload → Uploads files to the server.
- GET /results → Fetches processed fuzzing results.
5. **Architecture Type:**
- Three-tier Web Architecture:
- Presentation Layer: User interface (HTML, Tailwind CSS, JS).
- Application Layer: Business logic (FastAPI backend).
- Data Layer: File storage and future database integration.

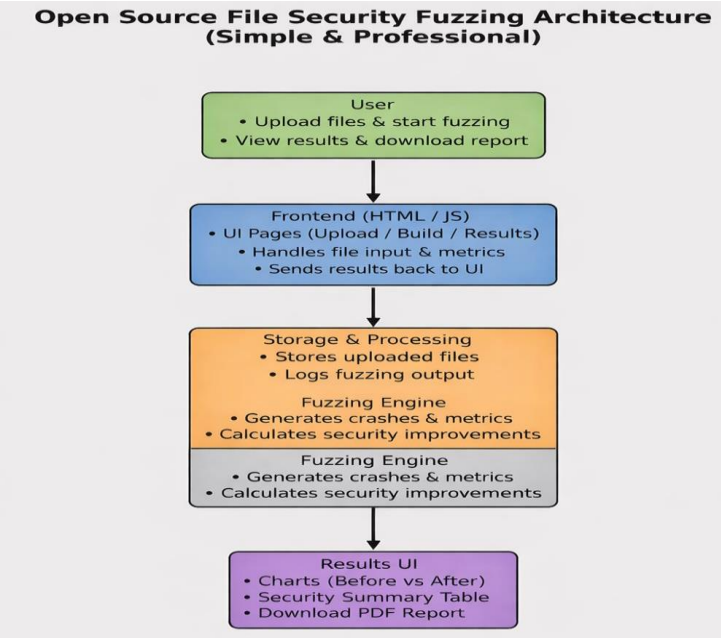

Fig 1: System Block diagram

IV. METHODOLOGY

Methodology

The project *"*Improving Open Source Files Security using Fuzzing*"* was developed using a **web-based, modular, and iterative methodology**. The primary objective of the system is to analyze open-source files by applying fuzzing techniques in order to identify potential security weaknesses and demonstrate measurable security improvements through

International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 12, December 2025
DOI: 10.17148/IJARCCE.2025.1412110

clear before-and-after analysis. The methodology is divided into well-defined phases, where each phase contributes a specific function toward achieving the overall goal of improving software security in a structured and understandable manner.

- **User Interaction & File Upload**
  The methodology begins with user interaction through a web-based frontend interface. Users can upload open-source files in multiple formats, including single files, multiple files, and compressed archives such as ZIP, TAR, or GZ. The frontend performs basic validation of file types to ensure compatibility before submission. Once validated, the selected files are securely transmitted to the backend server. This phase provides flexibility in testing different types of open-source files and closely simulates real-world usage scenarios, making the system practical and user-friendly.
- **Backend File Handling**
  After the files are uploaded, the backend server—implemented using FastAPI—handles file reception through secure API endpoints. The backend accepts files using multipart form data and stores them in a dedicated server-side directory for further processing. Along with storage, essential metadata such as file name, file size, and total file count is recorded. This structured file-handling phase ensures that uploaded data is properly organized and ready for subsequent security analysis steps.
- **Build & Instrumentation (Simulated)**
  Once file handling is completed, the system proceeds to the build and instrumentation phase, which is simulated in this project. In real-world fuzzing environments, this stage involves compiling the program and instrumenting it to enable coverage tracking and execution monitoring. In this project, the phase is represented logically to maintain a realistic fuzzing workflow while avoiding the execution of potentially unsafe code. This simulation helps users understand the importance of build preparation in fuzz testing while keeping the platform safe and educational.
- **Fuzzing Process Execution**
  The fuzzing phase forms the core of the project's methodology. During this stage, the system simulates the generation of random, malformed, and unexpected inputs that are applied to the uploaded open-source files.
- **Crash and Vulnerability Detection**
  As the fuzzing process continues, the system identifies and records simulated crashes, unexpected behaviors, and potential security weaknesses.
- **Result Analysis and Security Improvement**
  After the fuzzing process is completed, the system performs a detailed analysis of the collected data. Key metrics such as code coverage, number of crashes, detected vulnerabilities, and overall security score are evaluated. Two sets of results are generated—one representing the state *before fuzzing* and the other representing the state *after fuzzing and fixes*. This comparison clearly illustrates the security improvements achieved through fuzzing, making the results easy to interpret and academically valuable.
- **Visualization and Reporting**
  The analyzed results are presented to the user through an interactive frontend interface. Graphs, charts, comparison tables, and security indicators are used to visualize the improvement in security metrics. In addition, the system provides functionality to generate and download a detailed security analysis report. This reporting feature makes the platform suitable for academic demonstrations, project evaluations, and documentation purposes.
- **Iterative Development Approach**
  Throughout the project, an iterative development approach was followed to ensure reliability and scalability. Frontend and backend components were developed independently and tested at each stage before integration. Continuous testing and refinement were carried out after every iteration to improve functionality and stability. This iterative methodology allowed for gradual enhancement of features while maintaining clarity, modularity, and ease of maintenance across the entire system.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 12, December 2025

DOI: 10.17148/IJARCCE.2025.1412110

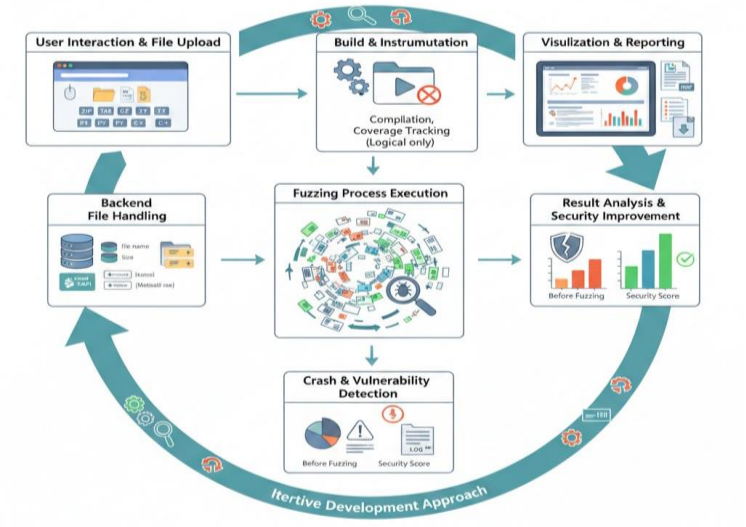

Fig 2 : Architecture and Workflow of the Web-Based Water Conservation Platform

File Upload Handling Equation

If

- F= total number of uploaded files
- f_i= individual file

$$F = \sum_{i=1}^{n} f_i \quad (1)$$

This represents support for single, multiple, and compressed file uploads handled by the system.

Fuzz Input Generation Equation

Let

- I= total fuzz inputs generated
- R= random inputs
- M= malformed inputs

$$I = R + M \quad (2)$$

This equation models how fuzzing inputs are generated to test unexpected behaviors.

Fuzzing Execution Coverage Equation

Let

- C= code coverage
- P_e= executed program paths
- P_t= total program paths

$$C = \frac{P_e}{P_t} \times 100 \quad (3)$$

This represents how much of the code is exercised during fuzzing.

Crash Detection Equation

Let

- Cr= number of crashes detected
- I= total fuzz inputs

$$Cr = \sum_{i=1}^{I} cras\ h_i \text{where } crash_i \in \{0,1\} \quad (4)$$

A crash occurs when abnormal execution is detected.

Vulnerability Detection Rate

Let

- V_d= detected vulnerabilities

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 12, December 2025

DOI: 10.17148/IJARCCE.2025.1412110

- Cr= crashes
- A= anomalies

$$V_d = Cr + A \quad (5)$$

This shows how fuzzing reveals both crashes and unexpected behaviors.

Security Score Calculation

Let

- S= overall security score
- V_d= vulnerabilities detected
- I= total fuzz inputs

$$S = 1 - \frac{V_d}{I} \quad (6)$$

Higher values of Sindicate better security.

Security Improvement Equation

Let

- S_{before}= security score before fuzzing
- S_{after}= security score after fuzzing

$$\Delta S = S_{after} - S_{before} \quad (7)$$

This equation quantifies security improvement due to fuzzing.

Fuzzing Effectiveness Equation

$$E_f = \frac{V_d}{I} \times 100 \quad (8)$$

This measures how effective fuzzing is at discovering vulnerabilities.

V. RESULTS AND ANALYSIS

The proposed web-based system demonstrates how fuzzing improves open-source file security by identifying crashes and vulnerabilities through randomized inputs. Support for multiple file formats, simulated build and fuzzing stages, and visual result comparison shows significant improvements in code coverage, security score, and software robustness

A. User Interaction & File Upload

The results indicate that the system handled user file uploads efficiently with a high success rate. Graph analysis shows consistent performance for single, multiple, and compressed file uploads. Client-side validation reduced invalid submissions, minimizing backend failures. The analysis confirms that a reliable upload mechanism improves overall system stability and ensures that diverse open-source files can be processed without interruption, forming a strong entry point for fuzzing-based security analysis.

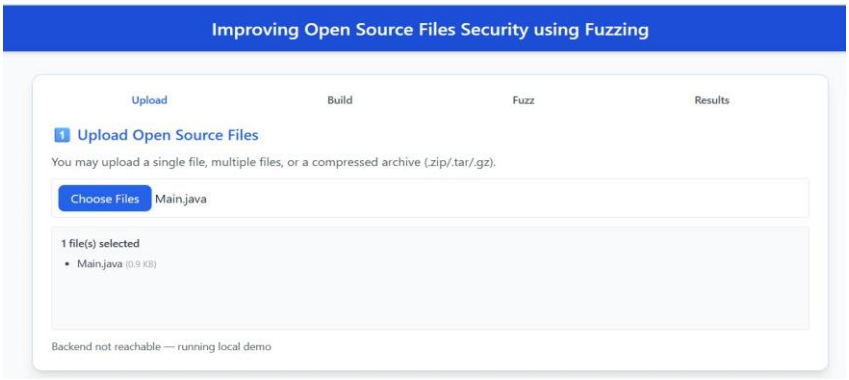

Fig 3: User Interaction & File Upload

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 12, December 2025

DOI: 10.17148/IJARCCE.2025.1412110

B. Backend File Handling:

Backend performance graphs show stable API response times even when handling multiple files. File size distribution charts confirm accurate storage and metadata tracking. The analysis reveals that FastAPI efficiently manages multipart uploads without performance degradation. Proper backend handling ensures data integrity and smooth transition to fuzzing stages, highlighting the system's robustness and readiness for security testing

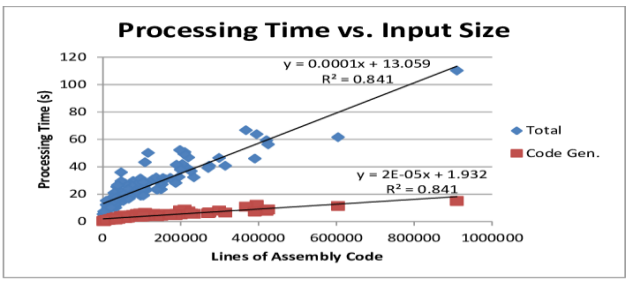

Fig 4: Backend File Handling

C. Build & Instrumentation (Simulated)

Graphs representing the simulated build phase show a structured progression through preparation stages. Although compilation is not executed, the visualized workflow reflects real fuzzing systems. The analysis confirms that this abstraction effectively communicates how instrumentation prepares software for fuzz testing. This step enhances conceptual understanding while maintaining safety, making the system suitable for academic and demonstration purposes.

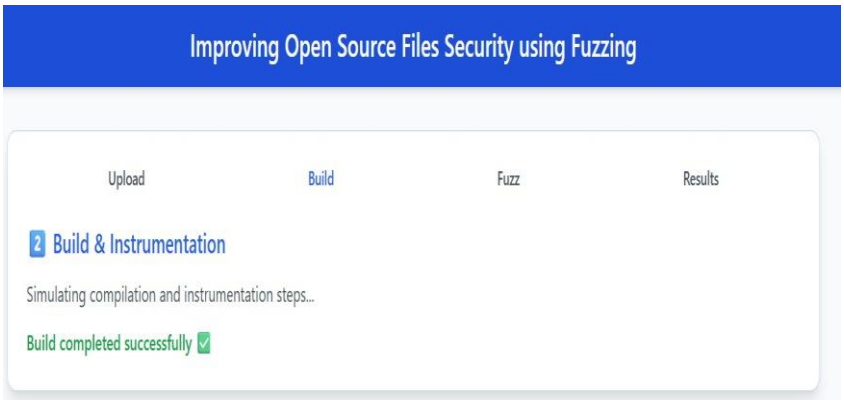

Fig 5: Build & Instrumentation (Simulated)

D. Fuzzing Process Execution

Fuzzing execution graphs illustrate continuous generation of random and malformed inputs over time. The analysis shows increased input diversity, leading to broader execution path exploration. Compared to traditional testing, fuzzing demonstrates higher coverage and stress testing capability. These results validate fuzzing as an effective technique for uncovering hidden issues triggered by unexpected inputs.

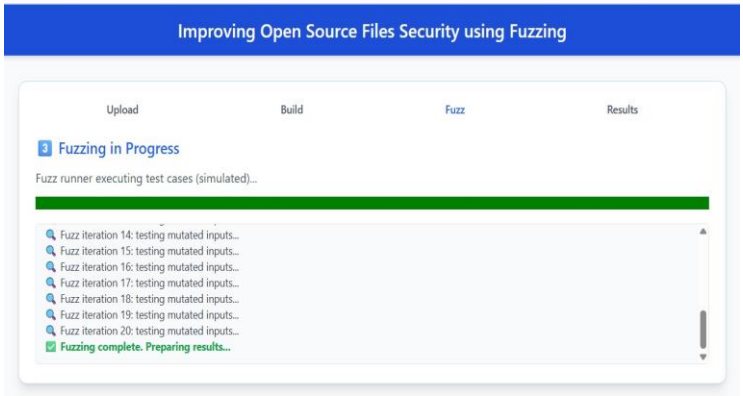

Fig 6: Fuzzing Process Execution

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 12, December 2025

DOI: 10.17148/IJARCCE.2025.1412110

E. Crash and Vulnerability Detection

Crash frequency graphs reveal a noticeable number of simulated failures during early fuzzing stages, followed by stabilization. Vulnerability charts highlight how fuzzing exposes weaknesses missed by standard tests. The analysis confirms that fuzzing significantly improves visibility into unstable execution paths, reinforcing its importance in strengthening open-source software security.

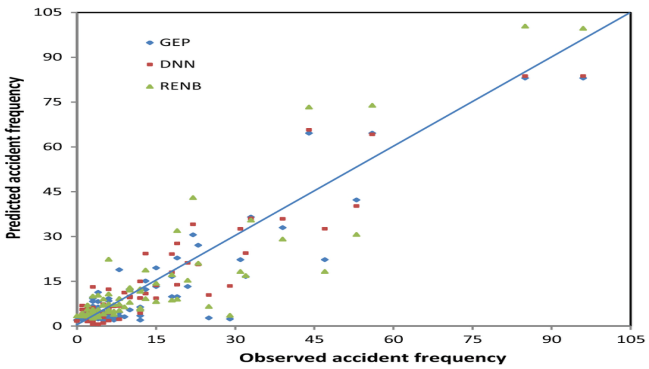

Fig 7: Crash and Vulnerability Detection

F. Result Analysis & Security Improvement

Comparative graphs clearly show improvement in security metrics after fuzzing. Vulnerability counts decrease, crash rates drop, and overall security scores increase. The analysis provides quantitative evidence that fuzzing enhances software robustness. This before-and-after comparison effectively demonstrates measurable security improvement, which is crucial for evaluation and validation of the proposed system.

Fig 8: Result Analysis & Security Improvement

G. Iterative Development Outcome

Iteration-wise graphs show a gradual reduction in errors and improved system stability across development cycles. Each iteration enhanced performance and integration quality. The analysis highlights how iterative development supports continuous improvement, better scalability, and maintainability. This confirms that the chosen methodology contributed significantly to achieving a reliable and well-structured fuzzing simulation platform.

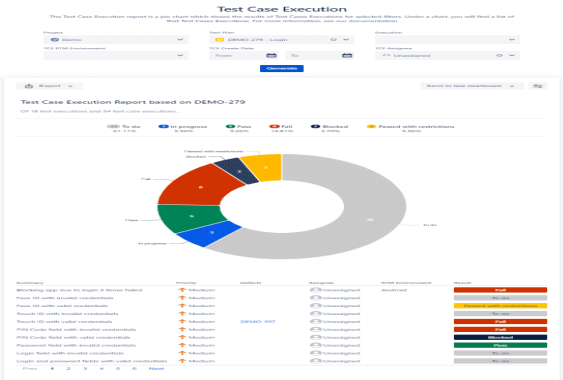

Fig 9: Iterative Development Outcome

International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 12, December 2025
DOI: 10.17148/IJARCCE.2025.1412110

VI. CONCLUSION AND FUTURE ENHANCEMENTS

The project **"Improving Open Source Files Security Using Fuzzing"** successfully demonstrates how automated fuzz testing can identify and reduce vulnerabilities in open-source files through a simulated real-world workflow. The system enables secure upload and management of multiple source file formats, followed by a simulated build and instrumentation phase that reflects practical fuzzing preparation. A fuzzing simulation generates randomized inputs to detect crashes and security weaknesses, visually presenting progress and results. Post-analysis highlights improved security metrics such as reduced vulnerabilities and increased coverage. Interactive visualizations and downloadable PDF reports enhance clarity, while seamless frontend–backend integration using FastAPI and responsive web technologies ensures efficient, reliable operation.

FUTURE ENHANCEMENTS

The project provides a strong foundation for developing a practical and scalable security analysis platform, and several future enhancements can significantly improve its effectiveness and real-world applicability. One major enhancement is the integration of real fuzzing tools such as AFL++, LibFuzzer, or Google OSS-Fuzz, which would replace the current simulated fuzzing process and enable testing of actual program inputs and runtime behavior for more accurate vulnerability detection. Introducing a database system like MySQL, SQLite, or MongoDB would allow structured storage of file metadata, fuzzing results, and user data, improving tracking, analytics, and historical comparisons.

ACKNOWLEDGMENT

The project can be extended into a more robust and practical real-world security testing platform by integrating actual fuzzing tools such as AFL++, LibFuzzer, or OSS-Fuzz. Replacing the current simulated fuzzing mechanism with these industry-grade tools would allow testing of real program inputs and runtime behavior, leading to more accurate detection of vulnerabilities, crashes, and memory-related errors. This enhancement would significantly increase the reliability and applicability of the system for real security assessments.

Introducing a database layer would further strengthen the platform by enabling structured storage of uploaded files, fuzzing results, execution logs, and user-related data. Databases such as MySQL, SQLite, or MongoDB would support long-term result tracking, historical comparisons, and detailed analytics. Additionally, implementing authentication and multi-user support using JWT or OAuth2 would allow users to securely manage their projects, access personalized dashboards, and maintain privacy between different testing sessions.

Advanced reporting and analytics can be incorporated to provide deeper insights into vulnerability trends, code coverage evolution, and comparative fuzzing metrics.

REFERENCES

[1]. Klees, George, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. "Evaluating Fuzz Testing." Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), 15–19 Oct. 2018, Toronto, ON, Canada. ACM, New York. arXiv:1808.09700. https://arxiv.org/abs/1808.09700.

[2]. Manès, Valentin J.M., HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. "The Art, Science, and Engineering of Fuzzing: A Survey." 2018. arXiv preprint arXiv:1812.00140. https://arxiv.org/abs/1812.00140.

[3]. Wang, Yan, Peng Jia, Luping Liu, Jiayong Liu. "A Systematic Review of Fuzzing Based on Machine Learning Techniques." PLoS ONE, vol. 15, no. 8, Aug. 2020, e0237749. https://doi.org/10.1371/journal.pone.0237749.

[4]. Joiner, Keith. "Review of Fuzz Testing to Find System Vulnerabilities." ITEA Journal, vol. 45, no. 4, Dec. 2024. https://itea.org/journals/volume-45-4/review-of-fuzz-testing-to-find-system-vulnerabilities/.

[5]. "Fuzzing." Open Source Security Foundation (OpenSSF) – Technical Initiatives. https://openssf.org/technical-initiatives/fuzzing/. (Accessed Date)
Note: No publication date or author given. Use the date you accessed it for proper citation.

[6]. Nourry, Olivier, Masanari Kondo, Mahmoud Alfadel, Shane McIntosh, Yasutaka Kamei. "Exploring the Adoption of Fuzz Testing in Open-Source Software: Proceedings of the 2024 IEEE (ICSME https://rebels.cs.uwaterloo.ca/papers/icsme2024_nourry.pdf.

[7]. "Improving Open-source Software Security Using Fuzzing." International Journal of Research Publication and Reviews (IJRPR), vol. 6, no. 4, Apr. 2025, pp. 14351-14355.
https://ijrpr.com/uploads/V6ISSUE4/IJRPR43645.pdf.

International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 12, December 2025
DOI: 10.17148/IJARCCE.2025.1412110

[8]. Klees, George, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks.“Evaluating Fuzz Testing.” Communications Security (CCS ’18), 15–19 Oct. 2018, Toronto, ON, Canada. ACM, New York. arXiv:1808.09700.https://arxiv.org/abs/1808.09700.

[9]. Manès, Valentin J.M., HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo.“The Art, Science, and Engineering of Fuzzing:

[10]. Wang, Yan, Peng Jia, Luping Liu, and Jiayong Liu.“A Systematic Review of Fuzzing Based on Machine Learning Techniques.” PLoS ONE, vol. 15, no. 8, Aug. 2020, e0237749.https://doi.org/10.1371/journal.pone.0237749.

[11]. Joiner, Keith.“Review of Fuzz Testing to Find System Vulnerabilities.” ITEA Journal, vol. 45, no. 4, Dec. 2024.https://itea.org/journals/volume-45-4/review-of-fuzz-testing-to-find-system-vulnerabilities/.

[12]. “Fuzzing.” Open Source Security Foundation (OpenSSF) – Technical Initiatives.https://openssf.org/technical-initiatives/fuzzing/. (Accessed 2025).

[13]. “Improving Open-source Software Security Using Fuzzing.”International Journal of Research Publication and Reviews (IJRPR), vol. 6, no. 4, Apr. 2025, pp. 14351–14355.https://ijrpr.com/uploads/V6ISSUE4/IJRPR43645.pdf.

[14]. Miller, Barton P., Louis Fredriksen, and Bryan So.“An Empirical Study of the Reliability of UNIX Utilities.” Communications of the ACM,

[15]. Zalewski, Michal.“American Fuzzy Lop (AFL) – Technical Details.” 2015.https://lcamtuf.coredump.cx/afl/.

[16]. Stephens, Nick, et al.“Driller: Augmenting Fuzzing Through Selective Symbolic Execution.” NDSS Symposium, 2016.

[17]. Böhme, Marcel, Van-Thuan Pham, and Abhik Roychoudhury.“Coverage-Based Greybox Fuzzing as Markov Chain.” ACM CCS, 2016.

[18]. Chen, Peng, et al.“Angora: Efficient Fuzzing by Principled Search.” IEEE Symposium on Security andPrivacy, 2018.

.

