IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

Review Of Artificial Intelligence And Machine Learning With The Methodology Of Fuzzing.

Prof.Vaishali N. Shelokar [1],Dr. Priti A. Khodke[2], Dr.Girish S. Thakare[3] , Prof.Nitin D. Shelokar[4]
[1] Prof Ram Meghe C.O.E.&M , Amravati, India
[2] Prof Ram Meghe C.O.E.&M , Amravati, India
[3&4] Sipna C.O.E.T., Amravati, India

Abstract

The integration of artificial intelligence (AI) and machine learning into fuzz testing, termed AI Fuzzing, presents a promising approach to enhancing software security by automating the identification of vulnerabilities through systematic input generation. However, traditional fuzz testing methods face significant limitations, particularly in their inability to encompass various attack types, leading to a critical gap in software protection prior to deployment. This study aims to explore the concept of AI Fuzzing, specifically focusing on the combination of coverage-guided and behavioral fuzzing techniques. The research employs a structured methodology utilizing OSS-Fuzz's Fuzz Introspector tool to identify under-fuzzed code segments, followed by an evaluation framework that generates and tests new fuzz targets using a large language model (LLM). The framework iteratively adjusts the fuzz targets based on observed outcomes, enhancing code coverage. The findings reveal prevalent software vulnerabilities, including memory leaks, injections, sensitive data exposure, insecure deserialization, buffer overflows, use-after-free errors, data races, and software crashes. These results underscore the effectiveness of AI-driven fuzz testing in revealing security flaws that traditional methods may overlook. The implications of this research highlight the potential for developers to leverage a variety of open-source fuzz testing tools, as well as enterprise solutions, to improve software security in diverse environments, particularly within larger development teams and DevOps settings. This study emphasizes the complementary nature of static code analysis and dynamic fuzz testing in identifying vulnerabilities, advocating for a holistic approach to software security.

Keyword –AI Fuzzing, software security, vulnerability detection, fuzz testing, machine learning, dynamic analysis code coverage

I. Introduction

Fuzz testing allows developers to deliver secure software rapidly by identifying security vulnerabilities and stability concerns during the initial phases of software development.

1. Conduct Security Assessments on the Source Code In a fuzz test, a program is run with invalid, unexpected, or random inputs to induce a crash in the application. Contemporary fuzzing tools can evaluate the code's structure they are designed to test. They are capable of generating thousands of automated test cases each second and tracking every path the inputs traverse within the program. This process provides a fuzzer with comprehensive feedback regarding the code coverage achieved during the execution of the source code.

2. Each Discovery Leads to Additional Discoveries When a fuzzer identifies an input that results in a crash, it employs mutation algorithms to create further inputs that are likely to replicate the original finding.

3. Fuzzing Safeguards Against Unforeseen Edge Cases Modern software fuzzers run a program with invalid, unexpected, or random inputs, thereby addressing unlikely or unforeseen edge cases that may not be examined through other testing methodologies.

4. Fuzzing Enhances Code Coverage Without False Positives As fuzzers execute the software under evaluation, they consistently provide inputs that can be utilized to replicate the identified bug. They can also identify the precise line of code in the repository where the problem resides.

II. LITERATURE REVIEW

Developers have access to a variety of open-source fuzz testing tools that can significantly enhance their testing processes. Many of these tools are tailored for specific applications, such as kernel fuzzing or particular programming languages. Additionally, there are enterprise-level fuzz testing solutions that prove to be highly effective for larger development teams or within DevOps frameworks. These enterprise tools typically offer a broader range of integrations and features, including automated bug reporting, continuous integration/continuous deployment (CI/CD) capabilities, integration with development tools, REST API fuzzing, and detection of OWASP vulnerabilities. It is important to note that while code analysis is a static process, fuzzing operates dynamically. Static analysis is categorized as white-box testing, whereas fuzzing is primarily considered black-box testing. Together, these two methodologies complement one another by uncovering vulnerabilities that may remain undetected if only one approach is utilized.

Contemporary fuzzing methodologies can be classified into three primary categories: black-box fuzzing, white-box fuzzing, and grey-box fuzzing. Of these categories, grey-box fuzzing stands out as the most prevalent and adaptable. Consequently, our team has chosen to concentrate on exploring grey-box fuzzing for our model. The coverage-guided fuzzing procedure of an application comprises several essential components.

What Types of Vulnerabilities Can Be Discovered Through Fuzzing?

1. Memory Leaks Improper handling of memory allocation that depletes available memory resources.

2. Injections The introduction of untrusted input into a program.

3. Sensitive Data Exposure The unintentional disclosure of personal information by an application.

4. Insecure Deserialization The injection of a manipulated object into the environment of a web application.

5. Buffer Overflow Writing data beyond the allocated buffer, potentially overwriting adjacent memory areas.

6. Use After Free Failure to reference a valid object, which may result in data corruption, segmentation faults, or general protection faults.

7. Data Races Simultaneous access to the same memory location in a multi-threaded process, leading to potential security vulnerabilities.

8. Software Crashes Malfunctions that cause a program to terminate unexpectedly, possibly resulting in critical system errors.

9. Functional Bugs The program becomes unresponsive to user inputs.

10. Uncaught Exceptions Failure to manage exceptions appropriately, which can disrupt the execution of the program.

11. Undefined Behavior Executing code that is not defined by the language specification (C/C++), potentially leading to security vulnerabilities.

III. Architectural Design.

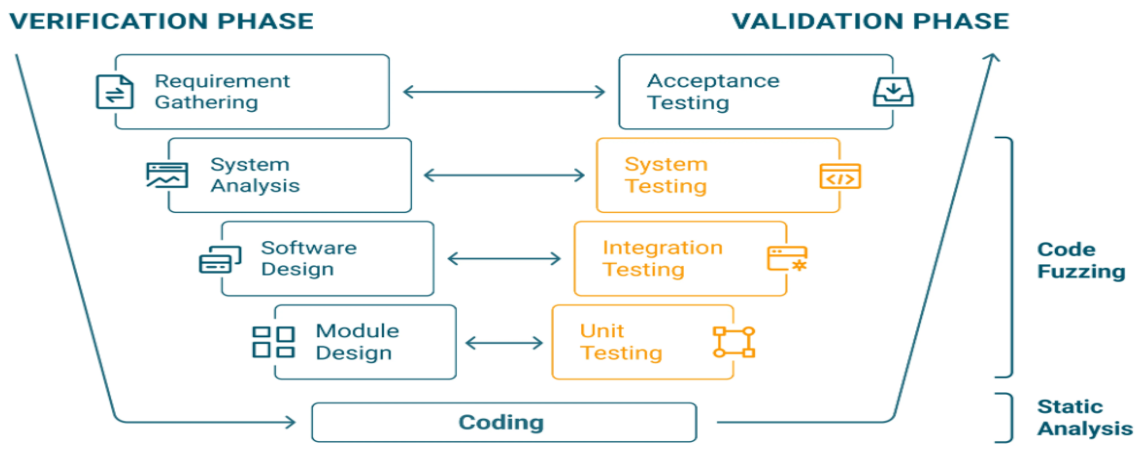
1. The Fuzz tool within OSS-Fuzz identifies a segment of the sample project's code that is under-fuzzed yet holds significant potential, subsequently forwarding this code to the evaluation framework.

2. The evaluation framework formulates a prompt for the LLM, which will be utilized to generate a new fuzz target, incorporating specific details pertinent to the project.

3. The evaluation framework executes the fuzz target produced by the LLM.

4. The evaluation framework monitors the execution for any alterations in code coverage.

5. Should the fuzz target encounter compilation issues, the evaluation framework requests the LLM to create a modified fuzz target that rectifies the compilation errors.



The most effective security strategy incorporates the use of both static and dynamic testing methods, including fuzz testing. By combining fuzz testing with Static Application Security Testing (SAST), a wider array of potential vulnerabilities can be identified early in the development process, thereby minimizing false positives and negatives while ensuring adherence to compliance standards.

Fuzzing comprises two main elements: the Verification Phase, which involves compilation-based static analysis, and the Validation Phase, which focuses on post-processing. It is capable of integrating data obtained from runtime coverage collection. To effectively employ these functionalities, fuzzing relies on external infrastructure to extract this coverage. Notably, there is a strong emphasis on utilizing OSS-Fuzz for this objective.

IV. Drawbacks in Fuzzing

Utilizing open-source fuzzing tools necessitates significant manual intervention to obtain effective testing outcomes. Furthermore, incorporating fuzzing technologies into the development workflow demands specialized expertise in IT security testing.

V. Methods and its Inclusion and Exclusion Criteria

To ensure that our survey of the current advancements remains focused and manageable, it is essential to delineate what falls within the scope of our study. The primary limitation is that we concentrate exclusively on works pertaining to machine learning-based fuzzing. This limitation leads to the establishment of the following five criteria:

1. Fuzzing methodologies that do not incorporate machine learning are excluded.

2. Approaches that utilize machine learning for vulnerability discovery without employing fuzzing are also excluded.

3. The survey is confined to papers published in peer-reviewed journals and technical reports from academic institutions. Consequently, we do not evaluate tools but rather the research that outlines their methodologies.

4. The search was restricted to publications dated between 2010 and 2020, and only studies published in the English language were considered.

5. Works that are not accessible via the internet are excluded.

VI. Source Material and Search Methodology

In order to identify potential papers, we conducted a systematic review of all publications from 2010 to 2020 in 15 leading venues related to computer security, software engineering, and artificial intelligence: IEEE S&P, ACM CCS, USENIX Security, NDSS, ACSAC, RAID, ESORICS, ASIACCS, DIMVA, ICSE, FSE, ISSTA, ASE, MSR, and AAAI. To discover additional candidate papers from other venues, we utilized specialized search engines, including Google Scholar. Furthermore, we meticulously reviewed the references within the candidate papers to uncover any additional relevant studies that may have been overlooked. The terminology associated with fuzzing encompasses: fuzz, fuzzing, fuzzer, input generation, test case, seed generation, crashes exploitability, mutation, while terms related to machine learning include: machine learning, neural network, deep learning, reinforcement learning, generative adversarial network, embedding, Bayesian network, decision tree, support vector machine, genetic algorithms, and random forest. We developed comprehensive search terms by integrating alternative terms and synonyms through the Boolean operator 'OR' and linking primary search terms with 'AND'. For instance, the following general search terms were employed to identify primary studies: Fuzzing AND machine learning, fuzzing AND neural network, fuzzer AND deep learning, fuzzing AND reinforcement learning, fuzzing AND embedding, fuzzing AND Bayesian network, fuzzing AND decision tree, fuzzing AND support vector machine, fuzzing AND genetic algorithms, fuzzing AND random forest, along with other analogous combinations.

Currently, the publicly available datasets utilized in prior research, along with their respective classifications, are outlined as follows:

• The "Big Code" dataset, developed by Veselin et al., is an open-source initiative that encompasses data from various programming languages. This includes Python abstract syntax trees (ASTs), which comprise over 100,000 Python files parsed into ASTs, and JavaScript ASTs, which consist of 150,000 JavaScript files available in both raw and parsed formats.

• The NIST SARD project dataset, released by the National Institute of Standards and Technology (NIST), has been extensively employed in research concerning vulnerabilities. It features test cases from over 100,000 programming languages, addressing numerous categories of weaknesses, including those identified in the Common Weakness Enumeration (CWE).

• The GCC test suite dataset, part of the GNU Compiler Collection, includes front-ends for languages such as C, C++, Objective-C, Fortran, Java, Ada, and Go, along with associated libraries like libstdc++ and libgcj.

• The DARPA Cyber Grand Challenge dataset consists of the DARPA Network Challenge Binaries, which includes 200 binary programs with diverse functionalities. These binaries are part of an open challenge aimed at developing tools that can automatically modify, validate, and rectify errors. Each binary is known to contain one or more bugs introduced by human programmers, as documented by the developers.

• The LAVA-M dataset, published by Dolan-Gavitt et al., comprises four programs from the GNU Coreutils suite: uniq, base64, md5sum, and who. These programs have been deliberately injected with 28, 44, 57, and 2265 errors, respectively, each assigned unique identifiers, along with some unlabeled errors. The errors are embedded deep within the programs and are triggered only when a specific offset in the program's input buffer aligns with a 4-byte "magic" value.

Machine learning has the potential to enhance fuzzing by refining various stages of the process, which include:

• Preprocessing: Gathering data regarding the target application and defining a fuzzing strategy

• Test case generation: Employing machine learning techniques to create test cases

• Input selection: Utilizing machine learning to choose appropriate inputs

• Result analysis: Applying machine learning to evaluate the outcomes

• Challenges A conflict exists between the objectives of learning and fuzzing. The learning process aims to understand the structure of valid inputs, whereas fuzzing seeks to disrupt that structure to uncover vulnerabilities.

• Techniques Several methodologies can be applied in machine learning-driven fuzzing, such as:

• TML techniques: Implementing predefined feature techniques and mathematical models to address regression and classification challenges

• Deep neural network techniques: These can be utilized to tackle issues faced in TML approaches

• Learn&Fuzz: This method employs neural-network-based statistical machine-learning techniques to automate the creation of input grammars

VII. Types of Fuzzers

A fuzzer that generates entirely random inputs and submits them to a program is referred to as a naïve fuzzer. Although naïve fuzzers are relatively simple to implement, they are unlikely to efficiently reach significant program states. Three main categories of contemporary fuzzers enhance the capabilities of naïve fuzzers: mutation-based, generation-based, and evolutionary. Mutation-based fuzzers indiscriminately alter or manipulate given inputs to test the program. Typically, mutation-based fuzzers lack awareness of the anticipated input format or specifications, which limits their ability to make informed mutation selections. Peach is an example of a fuzzer that can execute both mutation-based and generation-based fuzzing. Generation-based fuzzers, on the other hand, utilize information regarding the expected input format or protocol derived from specifications. These fuzzers create or construct inputs based on such specifications. Notable generation-based fuzzers include Peach and Sulley, a Python framework capable of generating inputs for file transfer protocols.

VIII. Evaluation of Fuzzer Performance

Metrics In assessing two distinct fuzzing methodologies, what specific metrics should be compared? What aspects do we measure? Currently, fuzzers are primarily evaluated based on their effectiveness and efficiency. When focusing on security vulnerabilities, the effectiveness of a fuzzer in a software environment is gauged by the total number of vulnerabilities it can potentially identify. Conversely, the efficiency of a fuzzer is measured by the speed at which it uncovers these vulnerabilities. Synthetic Vulnerabilities Do synthetic vulnerabilities accurately represent real issues? For the purpose of evaluation, faulty software systems can be created efficiently by introducing artificial defects into an existing framework. It is essential to empirically investigate whether these synthetic vulnerabilities truly reflect significant and genuine security flaws. If they do not, what distinguishes them from actual vulnerabilities? What measures can be taken to enhance the resemblance of synthetic vulnerabilities to real ones? Additionally, which categories of vulnerabilities are absent from synthetic bug assessments? Actual Vulnerabilities Are actual vulnerabilities, previously identified by other fuzzers, representative? An alternative method involves compiling genuine vulnerabilities discovered through various means into a benchmark. However, this approach can be labor-intensive, resulting in a relatively small sample size that may limit the applicability of the findings. Furthermore, this evaluation merely confirms that the newly introduced fuzzer can detect at least the same vulnerabilities as those previously identified, without assessing its capability to uncover new vulnerabilities. To what extent do the identified vulnerabilities represent the entirety of undiscovered vulnerabilities? One potential solution is to create a comprehensive, shared database of vulnerabilities across numerous software systems, compiled from findings by various fuzzers or auditors over time.

IX. Conclusion

This study examines the utilization of machine learning within the domain of fuzzing, drawing upon a comprehensive review of pertinent literature. Machine learning techniques are predominantly employed during the test case generation phase, where genetic algorithms successfully produce a variety of test cases to enhance both coverage and effectiveness in fuzzing. Deep learning approaches capitalize on their robust pattern recognition and feature extraction abilities to create more focused and high-quality test cases, thereby revealing potential vulnerabilities and unusual behaviors within the system. Reinforcement learning utilizes reward systems to direct the generation of test cases that facilitate the exploration and identification of abnormal behaviors, thus improving the efficiency and quality of test case generation. Additionally, machine learning has contributed significantly to the preprocessing, input selection, and result analysis stages of fuzzing, effectively enhancing the efficiency and vulnerability detection capabilities of the process. As a vital methodology in fuzzing, machine learning offers innovative ideas and technical solutions for advancing fuzzing techniques. Future investigations should delve deeper into the integration of various machine learning methods, leveraging their strengths to tackle the challenges encountered in fuzzing, thereby fostering the evolution and application of fuzzing technology. Fuzz testing is expected to maintain its essential role in future vulnerability assessments of projects. With ongoing advancements in this research area, it is anticipated that machine learning will continue to be applied to overcome obstacles in the fuzzing process. This article presents a thorough overview of the evolution of machine learning-based fuzzing techniques and associated research, aiming to provide a valuable resource for scholars in this field.

References

[1] Machine Learning-Based Fuzz Testing Techniques: A Survey AO ZHANG 1 , YIYING ZHANG 1 , YAO XU 1 , CONG WANG 1 , AND SIWEI LI2 1College of Artificial Intelligence, Tianjin University of Science and Technology, Tianjin 300457, China 2State Grid Information and Telecommunication Company Ltd., Beijing 102200, China

[2] M. Autili, I. Malavolta, A. Perucci, G.L. Scoccia and R. Verdecchia, "Software engineering techniques for statically analyzing mobile apps: research trends characteristics and potential for industrial adoption", *Journal of Internet Services and Applications*, vol. 12, pp. 1-60, 2021.

[3] S K Cha, M Woo and D Brumley, "Program-adaptive mutational fuzzing[C]", *2015 IEEE Symposium on Security and Privacy. IEEE*, pp. 725-741, 2015.

[4] M. Sutton, A. Greene and P. Amini, Fuzzing: Brute Force Vulnerability Discovery, Addison-Wesley, 2007.

[5] P. Godefroid, M.Y. Levin and D. Molnar, "Automated Whitebox Fuzz Testing", *Proceedings of NDSS2008 (Network and Distributed Systems Security)*, pp. 151-166, February 2008.

[6] FuzzerGym: A Competitive Framework for Fuzzing and Learning William Drozd, Michael D. Wagner

[7] A systematic review of fuzzing based on machine learning techniques Yan Wang,,Peng Jia,Luping Liu,Cheng Huang,,Zhonglin LiuPublished: August 18, 2020 https://doi.org/10.1371/journal.pone.0237749

[8] MITRE Corporation. CVE -Common Vulnerabilities and Exposures. 2019. [cited 17 Jul 2019]. Available from: https://www.cvedetails.com/browse-by-date.php.

[9] Höschele M, Zeller A. Mining input grammars from dynamic taints. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. ACM: 2016. p. 720–725.

[10] Wu F, Wang J, Liu J, Wang W. Vulnerability detection with deep learning. In: 2017 3rd IEEE International Conference on Computer and Communications (ICCC); 2017. p. 1298–1302.