A Comprehensive Review of Learning-based Fuzz Testing Techniques

Hao Cheng[1], Dongcheng Li[2,*], Man Zhao[1], Hui Li[1], and W. Eric Wong[3]
[1]School of Computer Science, China University of Geosciences, Wuhan, China
[2]Department of Computer Science, California State Polytechnic University - Humboldt, Arcata, USA
[3]Department of Computer Science, University of Texas at Dallas, Richardson, USA
haognehc@163.com, dl313@humboldt.edu, zhaoman@cug.edu.cn, huili@vip.sina.com, ewong@utdallas.edu
*corresponding author

Abstract—Fuzz testing has emerged as a dominant approach for identifying vulnerabilities, significantly improving software development and testing. Yet, traditional fuzz testing often grapples with inefficiencies and poor code coverage, relying heavily on the practitioner's expertise. With the rapid advancements in machine learning and deep learning within artificial intelligence, these technologies promise to revolutionize fuzz testing. This article critically examines learning-based fuzz testing methodologies. It starts by outlining fuzz testing's concept, core procedures, and established strategies. The discussion then shifts to the integration of machine learning and deep learning in fuzz testing, encompassing seed generation, scheduling, test case mutation, selection, target program analysis, and result evaluation. The paper concludes by addressing the current research gaps in this domain and speculating on future trends and opportunities for growth.

Keywords-fuzz testing; learning-based; machine learning; deep learning; software testing

1. INTRODUCTION

As software and applications grow in complexity, their internal code architectures become increasingly intricate, offering a broadened landscape for potential security vulnerabilities [1]. The complex code structures are likely to harbor numerous latent flaws, which hackers and malicious actors may exploit through various means in pursuit of system and application vulnerabilities. Such exploits can lead to data breaches, system crashes, service disruptions, and other security issues, thus necessitating dedicated detection methodologies to identify and rectify these vulnerabilities [2]. Current techniques for vulnerability discovery in software include static code analysis, dynamic code analysis, symbolic execution, and fuzz testing. Static code analysis [3] does not require the actual execution of the program; instead, it involves the direct examination of the source code or compiled binaries to identify potential issues. Dynamic code analysis [4] assesses a program's performance, security, and stability by executing the code and monitoring its behavior. Symbolic execution [5][6] tracks the values of symbolic variables during program execution, considering symbolic representations of inputs, thereby allowing for an analysis of the program's behavior under all possible inputs, which can aid in uncovering potential errors.

However, the aforementioned vulnerability detection methods necessitate substantial knowledge of the target program, limiting their widespread application. In contrast, fuzz testing requires minimal understanding of the target and can be easily scaled to large applications. Due to its simplicity and low performance overhead, fuzz testing has indeed achieved success in many practical applications, particularly in identifying security vulnerabilities [7]. Fuzz testing involves feeding a plethora of random or semi-random data into the target application to provoke potential errors or anomalous behaviors, thereby aiding in the identification and rectification of latent issues. This testing methodology is commonly employed in assessing various software systems, including network protocols, file formats, operating systems, and applications [8].

This article concentrates on the analysis and summarization of grey-box fuzz testing efforts utilizing machine learning [9] and deep learning [10]. It begins by examining conventional fuzz testing methodologies, proceeds to explore scholarly articles on the application of machine learning and deep learning within the domain of fuzz testing, and compiles a summary and organization of the related research work. Section 2 provides a succinct overview of fuzz testing's procedures, classifications, and respective advantages and drawbacks. Section 3 introduces current, mature, and widely implemented research related to traditional grey-box fuzz testing. In Section 4, the enhancements and optimizations that machine learning and deep learning contribute to the fuzz testing process are summarized, discussing both previous and ongoing research integrating these two technologies into fuzz testing. Finally, the article analyzes the challenges associated with the application of machine learning and deep learning to fuzz testing and anticipates future directions for its development.

2. TRADITIONAL FUZZ TESTING TECHNIQUES

Inquiry into traditional fuzz testing has been predominantly focused on gray-box fuzz testing methodologies. The quintessence of gray-box fuzz testing resides in a trifecta of mechanisms: the feedback acquisition mechanism, the feedback processing mechanism, and the sample generation mechanism [11]. The feedback acquisition mechanism is tasked with garnering feedback information from the test subject during the testing process. The feedback processing mechanism, taking cues from this information, meticulously selects high-caliber samples from the mutated specimens to constitute the corpus for the subsequent iteration of test cases. The sample generation mechanism mutates samples within the corpus, spawning new variants to furnish the testing phase with fresh inputs.

2835-2823/24/$31.00 ©2024 IEEE
DOI 10.1109/ISSSR61934.2024.00024

2.1. Definition

Fuzz testing, an automated or semi-automated technique, floods software with random or mutated inputs to detect defects or vulnerabilities, proving highly effective for dynamic vulnerability discovery in software and firmware [12][13]. Originally introduced by Miller et al [14]. to assess the reliability of Unix utilities, its relevance has grown with the expanding scale and complexity of software. Increasingly diverse applications of fuzz testing have emerged, often integrating with other software analysis methods to improve vulnerability detection and advance its complexity and functionality.

Based on the underlying objectives and principles, fuzz testing can be categorized into black-box, white-box, and grey-box fuzz testing. Black-box fuzz testing [15][16][17] focuses on the external behaviors and input-output relations of applications, while white-box fuzz testing [18][19][20][21] combines fuzz testing with code analysis to gain a deeper understanding of the application's internals. Grey-box fuzz testing [22][23][24], merging these approaches, utilizes partial internal structure information and external behavior observations to generate more targeted test cases.

Fuzz testing strategies can be delineated as generation-based and mutation-based fuzz testing. Generation-based fuzz testing crafts test cases directly from a specified model that delineates the anticipated inputs of the test program, such as the Peach Fuzzer developed by Eddington [25]. Mutation-based fuzz testing, on the other hand, generates test cases by randomly mutating a given seed file or employing predefined mutation strategies, excelling at uncovering software vulnerabilities without leveraging a priori knowledge of the target program. The American Fuzzy Lop (AFL) [26] exemplifies a state-of-the-art mutation-based grey-box fuzzer. AFL employs a selection of predefined mutation operators to create diverse inputs in an attempt to trigger latent vulnerabilities within the program under test. Following AFL, numerous descendants have emerged, adopting different techniques to augment their efficacy. For instance, Böhme et al. [27] built upon AFL to design its extension, AFLFast, and subsequently integrated directionality into grey-box fuzz testing to devise the directed grey-box fuzzer AFLGo. Yue et al. [28] further refined the AFLFast model and introduced the adaptive energy-saving grey-box fuzzer EcoFuzz.

In the context of fuzz testing approaches that navigate program exploration, fuzz testing can be categorized into coverage-based fuzz testing and directed fuzz testing. A primary objective of coverage-based fuzz testing is to achieve high code coverage in the target program, whether through methods such as those proposed by Böhme et al. [29] utilizing Markov models to construct the fuzz testing process, or the concepts advanced by Lemieux and Sen [30] that adjust mutation strategies to maintain deep coverage of the program—both aspire to attain high code coverage in the target program, i.e., coverage-guided grey-box fuzz testing (CGF). However, at times, the potentially erroneous code is known, obviating the need for increased code coverage; in such instances, directed fuzz testing techniques (DGF) can be employed for detection, thereby utilizing fuzzers generated by this technique for targeted vulnerabilities or error examination. For example, Chen et al. [31] explored the creation of precise fuzz testing memory layouts for directed fuzz testing.

2.2. Basic Process

The classical fuzz testing workflow includes input pre-processing, test case generation [32][33], seed selection and scheduling [34][35], execution, target monitoring [36][37], and result analysis. Pre-processing prepares for testing by initializing the process through analysis of inputs and program information, often involving techniques like instrumentation, symbolic execution, and taint analysis. Following this, test case generation becomes central, with prevalent methods based on generation [38][39] and mutation [40][41][42]. Generated cases enter a seed pool where fuzz testing tools prioritize those likely to reveal anomalies. The execution module tests the target program with chosen cases while monitoring for anomalies to inform continued testing decisions. Finally, tools analyze anomalies to locate and diagnose causes, assessing the target software for vulnerabilities.

2.3. Deficiencies

As fuzz testing technology evolves, it has been widely applied in the field of security vulnerability detection, encompassing a variety of domains such as operating system kernels, the Internet of Things, software applications, and network protocols. Despite numerous advantages of fuzz testing—including ease of deployment, scalability, and broad applicability—there are still some limitations, summarized as follows:

- Limited Automation: Although traditional fuzz testing is categorized as a semi-automated testing technique, the processes of analyzing target software, constructing input data formats and specifications, and generating test cases still require substantial manual effort. Therefore, augmenting the automation and intelligence of fuzz testing practices is a research priority.
- Inefficient Test Case Generation: Traditional mutation-based strategies randomly mutate normal seeds, generating a vast number of test cases, but only a few trigger exceptions, leading to suboptimal results. This inefficiency arises from the methods' failure to deeply analyze the target program's internal structure and logic, weakening the test case generation's relevance to the program.
- Uniform Test Case Selection and Scheduling: Fuzz testing often yields numerous test cases, yet many tools treat them uniformly, executing them sequentially without considering differences in execution paths or anomaly-triggering potential. This indiscriminate approach reduces fuzz testing's efficiency. Fuzz testing

tools should filter and schedule test cases based on specific criteria to improve testing outcomes.

3. Overview of Learning-Based Approaches

Learning-based methodologies enhance system performance by discerning patterns within data, primarily bifurcating into machine learning and deep learning.

3.1. Machine Learning

Machine learning [43], a branch of Artificial Intelligence (AI), is dedicated to the research and development of algorithms and models that enable computer systems to autonomously learn and enhance performance. The objective is to empower these systems to learn from experience without explicit programming. Pertinent concepts include:

- Learning: Machine learning empowers computers to make more informed decisions by identifying data patterns and structures, enhancing future performance.
- Data: Essential to machine learning, data facilitates experiential learning, with algorithms discerning patterns through extensive analysis.
- Features: These are critical data descriptors that aid model interpretation.
- Supervised and Unsupervised Learning: Supervised learning [44][45] employs annotated data to teach models to predict outcomes, while unsupervised learning [46][47] uncovers data patterns without labels.
- Overfitting and Underfitting: Overfitting [48] occurs when a model excels with training data but fails on new data; underfitting [49] reflects a model's inability to learn sufficiently, impairing its performance on unseen data.

3.2. Deep Learning

Deep Learning [50] comprises machine learning methods rooted in multi-layered neural networks, enabling abstract and complex representation learning that excels in various tasks. Key concepts include:

- Neural Networks: Arrangements of neurons (nodes) linked by edges, usually featuring input, hidden, and output layers.
- Hierarchical Structure: The organization of neurons in layers, each performing specific computational roles, with "depth" indicating layer quantity.
- Activation Functions: These determine neuron outputs and include variants like Rectified Linear Unit [51], Sigmoid [52], and Tanh [53], essential for learning complex patterns.
- Backpropagation: A core training algorithm for deep learning models, it calculates loss function gradients to update parameters using gradient descent.

In fuzz testing, prevalent learning models include RNNs, DQL, etc., as shown in Table 1.

Table 1. Common learning-based model in fuzz testing

Abbreviation	Full Name	Papers Using the model
MAB	Multi-Armed Bandit model	[58,78]
RNN	Recurrent Neural Network	[56,57,58, 66,70]
MCM	Markov chain model	[56,60,61]
TS	Thompson Sampling	[59,63]
FNN	Feed-forward Neural Networks	[65]
DQL	Deep Q-Learning	[64,68,70]
DDPG	Deep Deterministic Policy Gradient	[61]
GAN	Generative Adversarial Network	[58,62]
CNN	Convolutional Neural Networks	[67]
GEN	Graph Embedding Network	[71]

4. Learning Based Fuzz testing Techniques

Recent advancements in machine learning and deep learning have led researchers to apply these techniques to fuzz testing. Their goal is to extract insights from vast vulnerability data, using trained models to improve prediction, classification, and generation. This strategy aims to enhance the understanding of software vulnerabilities and increase the precision and efficiency of vulnerability detection, as illustrated in Figure 1.

Recent work on the application of machine learning and deep learning in fuzz testing reveals a fourfold role for these technologies. First, refining the generation and scheduling of initial seed files; second, optimizing the mutation process of test cases; third, assessing software by predicting execution paths to guide the mutation and generation of test cases, enhancing efficiency; and fourth, applying machine learning for test result analysis or integrating with fuzz testing techniques to introduce functionalities like software evaluation and vulnerability exploitation.

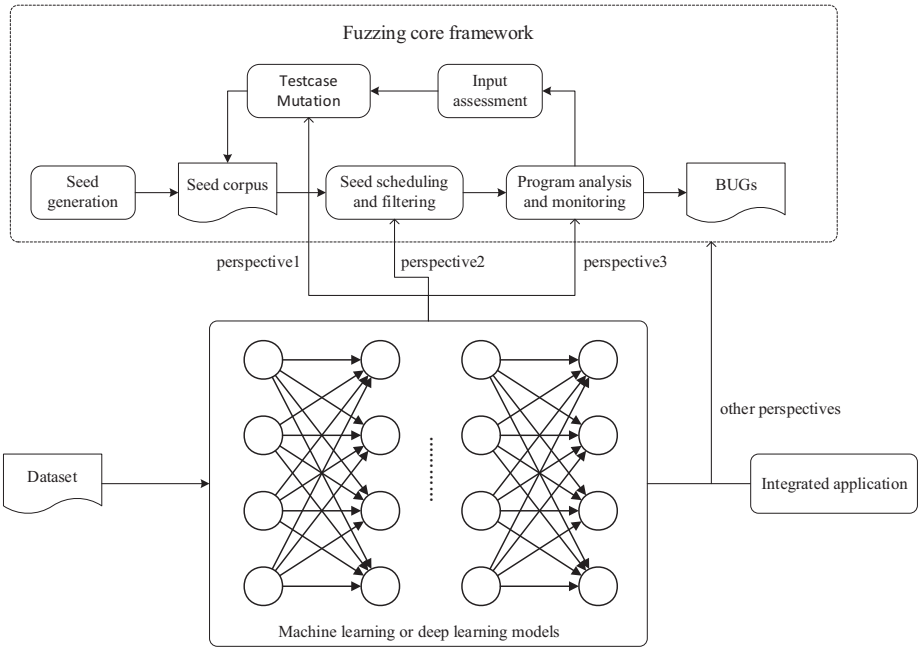

Figure 1. Learning-based fuzz testing framework

4.1. Using Learning Based Models for Seed Inputs Optimizing

Seed scheduling, which determines the initial test cases for fuzzing tools and the quantity of mutations for input seeds, significantly affects fuzz testing outcomes. Prioritizing seeds with higher crash probabilities and producing more mutations enhances the fuzzer's efficiency in exploring program paths and detecting potential errors.

Identifying the exact test cases that cause software crashes is challenging due to randomness in fuzz testing. Researchers often use heuristic strategies for selecting seed test cases, but these don't always accurately reflect the likelihood of faults or allocate resources effectively, especially with various program characteristics. Inadequate seed scheduling can delay critical seed discovery and hinder resource allocation. To address this issue, Choi et al. [54] introduced a seed scheduling algorithm based on reinforcement learning. This algorithm aims to optimize seed input order and mutation resource distribution, thereby improving crash detection efficiency. This method can be applied universally to coverage-guided fuzz testing, independent of software structure.

Fine-grained coverage metrics allow fuzzers to detect errors beyond traditional edge coverage. However, these metrics lead to larger seed pools with few effective scheduling algorithms. To address this, Wang et al. [55] introduced a multi-tiered coverage metric, integrating sensitive coverage indicators into grey-box fuzz testing, and developed a reinforcement learning-based hierarchical seed scheduling algorithm. This approach triggers more errors, achieves higher code coverage, and reaches coverage rates faster than existing methods.

In fuzz testing, to boost code coverage and induce software crashes, researchers must craft a quality mutation seed set. Current studies rarely use seed input's initial execution paths to enhance coverage. Li et al. [56] introduced a machine learning framework correlating seed inputs with their execution paths, to guide new seed generation to activate unexplored paths. For the complex PDF file structure, they employed a Markov chain-based model for PDF execution path probabilities and an improved sequence-to-sequence recurrent neural network to generate PDFs from these paths and existing content. Tests confirm this framework's effectiveness in raising code coverage.

Grammar-based fuzz testing excels for applications with structured inputs. Though creating input grammars manually is tedious and prone to errors, Godefroid et al. [57] automated this by learning input structures and semantics using a seq2seq recursive neural network. Their technique produces new seeds from seed file analysis, increasing fuzz testing efficiency and code coverage. Nichols et al. [58] used LSTM and GAN to learn from a seed corpus, generating new seeds and integrating GAN with the fuzz tester AFL, deepening path exploration and significantly raising the number of discovered paths and their length. These studies show machine learning can significantly enhance automation and efficiency in grammar-based fuzz testing.

4.2. Using Learning Based Models for Test Case Mutating

Fuzz testing excels at detecting complex program vulnerabilities by generating numerous test cases, many of which are irrelevant. To improve, recent work has integrated machine learning with fuzz testing, focusing on mutating critical input bytes or using deep learning to guide beneficial mutations.

Karamcheti et al. [59] showed that an informed sampling of mutation operators can significantly improve AFL's efficacy. They devised an adaptive method combining Thompson sampling, machine learning, and bandit optimization, which tunes mutation distributions on the fly. This results in higher code coverage and quicker, more reliable crash detection than AFL's baseline and other AFL-based methods.

Böttinger et al. [60] showed that viewing fuzz testing as a feedback-driven learning process enhances input generation quality. Using Markov Decision Process theory, they recast fuzz testing as a reinforcement learning challenge and developed a method based on deep Q-learning. This method effectively learns to select high-reward fuzzing operations. Their experiments confirmed the algorithm's ability to optimize new input generation based on ongoing feedback.

Addressing inefficient fuzz testing from random test case mutations, Zhang et al. [61] used a Markov Decision Process to describe traditional fuzz testing and implemented the Deep Deterministic Policy Gradient (DDPG) algorithm. This approach selects high-reward mutations for program inputs, creating the reinforcement learning-based fuzzing system RLFUZZ. RLFUZZ uses a reinforcement learning algorithm to choose actions that maximize rewards, guiding sample mutations. Tested with various warm-up steps and activation functions, RLFUZZ showed substantial edge coverage improvements compared to baseline mutations and DQN algorithms.

Sun et al. [62] introduced a VAE-GAN model that combines the strengths of Variational Autoencoders and Generative Adversarial Networks, using mean feature matching to quicken convergence and enhance test case variety. The VAE-GAN model outperformed traditional GANs, increasing code coverage by 11.87% and more effectively revealing unique crashes and hang-ups.

Lee et al. [63] developed SEAMFUZZ, a fuzz testing method that tailors mutation strategies to seed inputs based on their syntactic and semantic characteristics, to employ a Thompson sampling algorithm to refine test case precision. SEAMFUZZ demonstrated greater effectiveness in path exploration and bug detection than other advanced fuzzing tools.

Shao et al. [64] proposed an optimized seed mutation method for AFL that reduces redundant mutations by pinpointing effective bytes in seeds and using reinforcement learning to guide more impactful test case generation, yielding significant enhancements over traditional random mutations.

4.3. Using Learning Based Models for Target Program Analysis

Beyond using machine learning to refine seed inputs and mutation strategies, scholars have leveraged these models to predict execution paths and analyze program behavior, thereby improving fuzz testing.

She et al. [65] note that treating fuzz testing as a machine learning task is common, yet smooth program execution paths are vital for effective gradient-based searches. They developed NEUZZ, a smoothing technique using a neural network to approximate control flows and calculate gradients, to guide input mutations to increase fault detection. This neural-guided fuzz testing method has significantly enhanced testing efficiency.

Cheng et al. [66] propose a machine learning framework to mitigate traditional fuzz testing tools' reliance on predefined seed inputs for defect detection. The framework leverages neural networks to correlate input data with program execution paths, producing seed inputs that access unexplored code paths. This method not only improves code coverage but also enhances fuzz testing's effectiveness and efficiency by integrating with modern fuzzing mutation strategies. Experiments confirm its ability to boost code coverage and detect potential crashes.

Zong et al. [67] introduce FuzzGuard, a deep learning method addressing inefficiencies in directed grey-box fuzz testing, characterized by inputs failing to reach target code segments. FuzzGuard predicts whether inputs will hit the target area, filtering out ineffective ones and enhancing test performance. The team also devises solutions for data imbalance and time constraints, with results showing FuzzGuard can increase testing speed and save up to 88% of testing time. Liang and Xiao [68] developed RLF, a deep reinforcement learning-based algorithm for targeted fuzzing. RLF selects test samples by instrumenting programs and assessing distances between execution paths and targets. Using deep Q-Learning, RLF refines its strategy, making test selection deliberate and targeted, thus improving test case quality and targeted fuzz testing's efficiency.

Li et al. [69] investigated machine learning applications in fuzz testing, addressing hash collisions and using edge knowledge. They created SpeedNeuzz, which uses advanced hash techniques and neural networks for simulating program behavior. SpeedNeuzz improves mutation strategies with gradient-based methods, enhancing edge coverage more than NEUZZ.

Jeon and Moon [70] developed an algorithm to boost hybrid fuzz testing with deep reinforcement learning, creating Dr. PathFinder. This tool uses symbolic execution for test case generation and diverges from standard concolic execution by adaptively choosing paths. Its reinforcement learning-driven approach focuses on deeper paths, reducing ineffective test cases and memory usage, and controlling state space explosion. Dr. PathFinder matches or exceeds other bug fuzzers in finding deep-seated errors.

Li et al. [71] introduced V-Fuzz, an evolutionary fuzzing framework for detecting vulnerabilities. Combining a graph neural network-based prediction model with an evolution-inspired fuzzer, V-Fuzz targets and triggers vulnerabilities more effectively. Its tests confirm rapid error detection and have uncovered multiple security flaws, including three previously unknown vulnerabilities.

4.4. Using Learning Based Models for Test Results Evaluation

Beyond enhancing seed scheduling, test case mutation, and target program analysis with machine learning, researchers have optimized fuzz testing phases. They use machine learning or deep learning to evaluate evolutionary algorithm-based fuzzing outcomes, aiming to detect exceptions and improve vulnerability detection.

Ming et al. [72], following a fuzz testing campaign on Intelligent Transportation Systems (ITS), leveraged machine learning to analyze the harvested request and response messages. This analysis facilitated the automatic identification of protocol vulnerabilities and related anomalies within the ITS, enhancing the efficacy and efficiency of fuzz testing analysis.

Fuzz testing effectively uncovers memory corruption vulnerabilities, yet not all anomalies signify exploitable flaws. The intensive analysis of widespread crash data poses a major challenge to timely vulnerability discovery. To boost efficiency, researchers have turned to machine learning for parsing test characteristics, enabling the automatic identification of critical crashes and evaluation of their exploitability. For example, Tripathi et al. [73] have crafted a robust crash identification model using hardware-monitored runtime data and SVM analysis of core dumps. Simultaneously, Zhang and Thing [74] have expedited vulnerability detection by employing online classifiers to prioritize crashes, streamlining the identification and remediation of pressing vulnerabilities. These advances refine the vulnerability mining process, focusing efforts on severe crashes and hastening critical flaw resolution.

Some studies have integrated machine learning with fuzz testing to create methodologies that enhance vulnerability detection. This includes analyzing software exploitability and using code generation techniques. By integrating these methods, researchers gain a deeper understanding of fuzz testing outputs, improving vulnerability discovery. Such comprehensive approaches not only identify anomalies but also analyze and support the remediation of potential vulnerabilities. Yan et al. [75] combined machine learning with fuzz testing for software measurement, using Bayesian algorithms and dynamic fuzz testing to refine predictions of software exploitability, thereby establishing a precise framework for its quantification. You et al. [76] utilized NLP for information extraction and semantic fuzz testing to extract vulnerability insights from sources like CVE reports and Linux git logs, facilitating a better understanding of vulnerabilities and guiding the automatic generation of Proof of Concept (PoC), thus making vulnerability mining more thorough and efficient.

5. RESEARCH ISSUES

Comparative assessment is vital for learning-based fuzz testing frameworks. This section compares datasets, framework features, and additional information in fuzz testing. It elucidates dataset attributes and their use in learning-based fuzz testing. Finally, the section addresses potential challenges and opportunities in this field.

5.1. Dataset

Datasets employed in learning-based fuzz testing frameworks encompass several varieties, as illustrated in Table 2.

- Crafted by Columbia University's scholars, the LAVA-M [77] dataset, standing for Large Scale Automated Vulnerability Addition, is designed to facilitate fuzz testing and vulnerability discovery for researchers. This collection boasts a vast array of machine-crafted samples that illustrate diverse vulnerability types, including buffer overflows, format string weaknesses, and integer overflows.
- The Peach Fuzzer [25] is a fuzz testing platform that uncovers flaws in software and protocols. It offers a flexible framework for creating varied fuzzing inputs for files, networks, and APIs. Its customization options allow for detailed test scenarios, and it integrates easily with automated testing systems.
- The GNU binutils [78] are a collection of tools for creating, handling, and altering binary object files, essential in software development for transforming source code into executables and for their subsequent manipulation and debugging. Commonly paired with the GCC (GNU Compiler Collection), these utilities facilitate the compilation and linking of C and C++ programs.
- The AFL [26] (American Fuzzy Lop) is a fuzzing tool focused on increasing the efficiency of vulnerability detection in software. It employs a dynamic feedback mechanism to autonomously generate and execute test cases, pinpointing potential software vulnerabilities.
- The CGC [79] (Cyber Grand Challenge), orchestrated by DARPA, is an international cybersecurity competition promoting the advancement of automated defense systems. The challenge seeks to fortify digital security measures against escalating cyber threats by catalyzing the development and application of automated systems in cybersecurity.
- The CB-multios [80] is a fuzzing framework designed to enhance operating system diversity. It employs advanced fuzzing techniques, integrating symbolic execution and coverage-guided test generation with mutation strategies, to reinforce the integrity of OS interfaces and syscalls. The platform enables researchers to devise and implement tailored fuzzing tests, using its custom mutation engine to explore various input combinations and address potential vulnerabilities.
- Developed and maintained by Google, the Fuzzy Test Suite [81] is a suite of tools for fuzz testing that aids in detecting and fixing security vulnerabilities and bugs in software systems. It includes a range of utilities and techniques for crafting and running fuzz tests.

Table 2. Common datasets in learning-based fuzzing

Dataset Name	Brief Description	Papers Using the dataset
LAVA-M	a dataset formed by inserting bugs into programs by LAVA	[54,59,61,65,68,71]
Peach Fuzzer	A tool for generating diverse types of fuzzing inputs	[25,56,59,63,64,68]
GNU binutils	GNU's binary toolset	[27,28,28,30,31,54,62,63,65,69]
AFL	A Fuzzing tool including sample files for generating fuzzing inputs	[54,55,56,60,67,68]
Cyber Grand Challenge	Global cybersecurity race	[55,59]
CB-multios	CGC binary file	[70]
Google Fuzzy Test Suite	A collection of tools for conducting fuzzing	[11,55]

5.2. Learning Based Fuzz Testing Framework

To ensure a comprehensive survey, this article compiles a decade's fuzzy testing papers, chronologically presented in Figure 2. The trend shows an increasing publication volume, with a surge in learning-based fuzzy testing research since 2016, indicating a growing interest in applying learning methodologies to fuzzy testing.

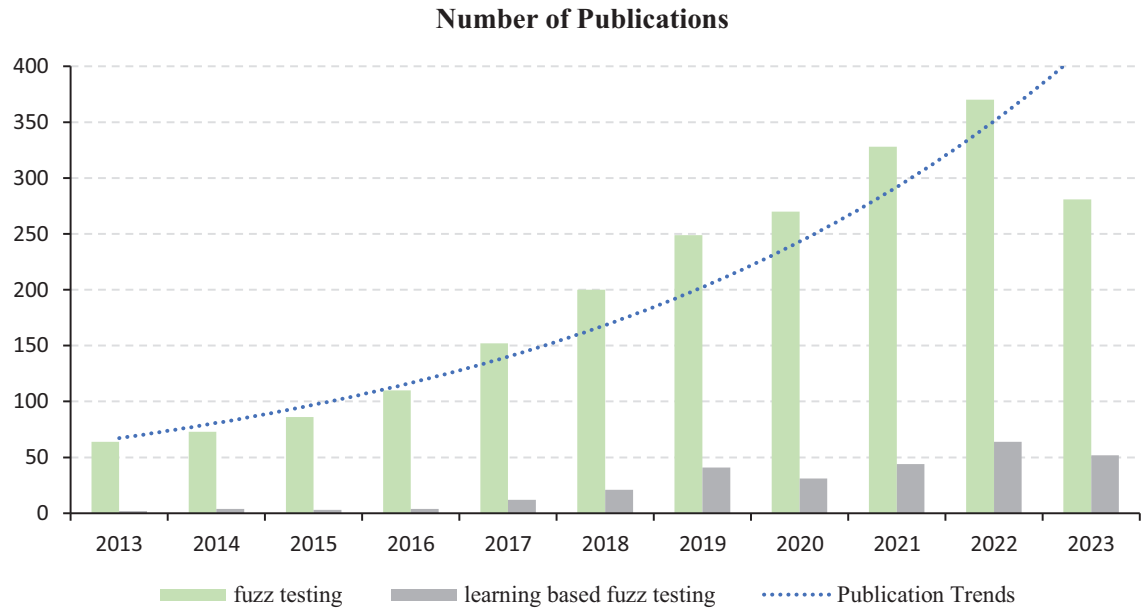

Figure 2. Papers on Fuzz Testing from 2013 to 2023

The efficiency of fuzz testing is primarily contingent upon factors such as the quality of initial seeds, seed selection and scheduling techniques, and case mutation strategies. Hence, a multitude of studies have integrated learning-based methods into these aspects, yielding an array of sophisticated learning-based fuzz testing frameworks, cataloged in Table 3. This paper segments these frameworks into four categories, each corresponding to the frameworks that utilize learning-based models to enhance the fuzz testing cycle, as discussed in the previous section.

The initial category involves frameworks that enhance seed scheduling with learning-based models, notably SampleFuzz and GAN-AFL. The core process, illustrated in Figure 3, involves the following key steps:

- Data Collection: Compile data on seed performance and test outcomes.
- Feature Extraction: Derive key attributes from seeds and their executions, such as input size, execution trajectory, code coverage, and time spent.
- Model Training: Train deep learning models using the gathered data and identified features.
- Seed Assessment: Evaluate seeds with the aid of the trained models.
- Adaptive Scheduling: Dynamically prioritize seeds based on model assessments to refine the test progression.

This methodology provides a more adaptive alternative to static rules, continuously learning from new test feedback to optimize its scheduling algorithms.

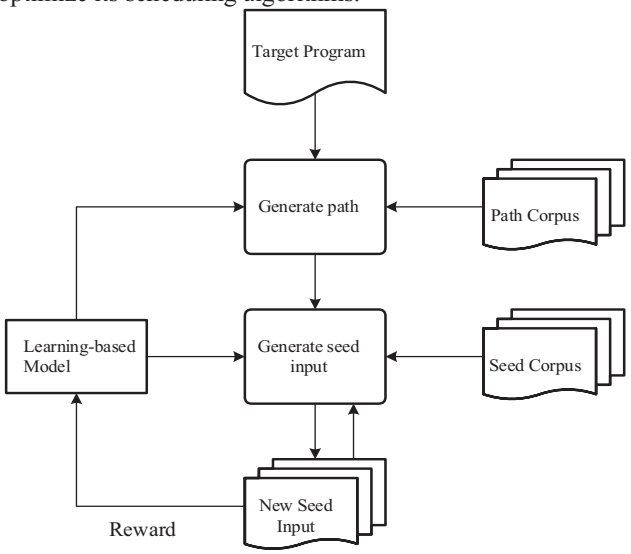

Figure 3. Generating new fuzzing seeds using learning-based framework

The second category employs learning-based methods like the RLFUZZ and VecSeeds models to refine test case mutation strategies, often through reinforcement learning, as illustrated in Figure 4.

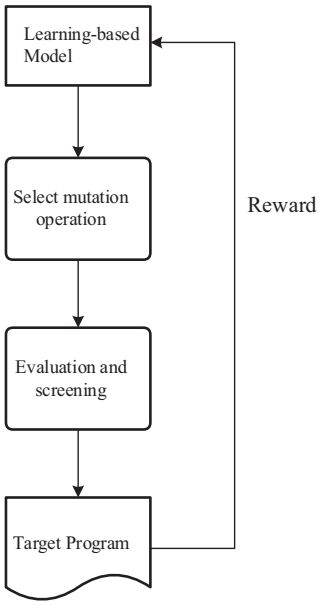

Figure 4. The architecture of testcase mutation using learning-based model

Test case mutation involves altering test inputs to reveal software defects. Machine learning optimization enhances bug detection efficiency and reduces testing costs. For example, models learn optimal mutations through trial and error, adjusting strategies based on feedback regarding whether new code paths or crashes were triggered. Clustering algorithms also group test cases to prioritize mutating those more likely to expose vulnerabilities, offering more targeted and adaptable mutations than traditional methods.

The third category adopts learning-based methods for direct analysis of the target software, exemplified by the SpeedNeuzz and V-Fuzz models which leverage machine learning to predict execution paths and behavior, guiding test case mutation and generation to improve fuzz testing results. The process, illustrated in Figure 5, starts with data collection from the target program, including inputs, responses, exceptions, system calls, and performance metrics. This is followed by feature selection—determining which data components are valuable for predicting vulnerabilities. With chosen features and collected data, a model is trained to produce new test cases, enhancing the probability of detecting unknown program errors or vulnerabilities.

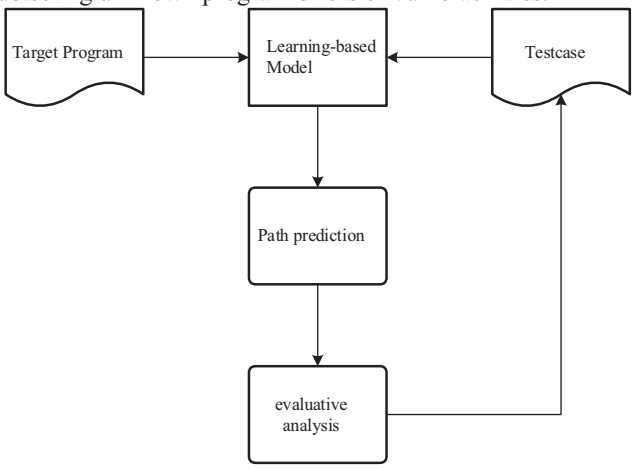

Figure 5. The framework of analyzing the diagram of the target program using learning-based model

The fourth category uses machine or deep learning to refine fuzz test analyses, detect anomalies and uncover vulnerabilities. The process begins with training the model on fuzzing results, including crash reports, exception logs, memory leaks, and code coverage. The model is then evaluated on a separate test set, and finally applied to new fuzz test data to identify potential security flaws and their origins.

This article reviews the evolution of learning-driven fuzz testing over the past decade, highlighting frameworks that incorporate learning to enhance fuzz testing. Representative frameworks and their succinct synopses are delineated in Table 3. This paper catalogs the characteristics, mechanisms, datasets, and learning models, emphasizing the last five years' technological strides due to machine learning and the demand for software security. Comparing different frameworks remains difficult without unified performance standards.

Table 3. Learning-based fuzzing framework and functionality

Model Name	Brief Description	Learning Model	Paper
SampleFuzz	Using a learned input probability distribution to intelligently guide where to fuzz inputs	RNN	[57]
GAN-AFL	Using GAN models to reinitialize AFL system with novel seed files to improve its performance	GAN	[58]
RLFUZZ	Using reinforcement learning algorithm to guide testcase variation	Markov chain, DDPG	[61]
VecSeeds	Provide inputs conforming to the input format for programs tested	GAN	[62]
SEAMFUZZ	Automatically capturing features of individual seed inputs and applying different mutation strategies to distinct seed inputs	MAB, TS	[63]
RLFuzz-IF	Training the mutator's behavior on test cases to optimize mutations	DQL	[64]
NEUZZ	Approximating and smoothing the branch behavior of the program to enhance the efficiency of fuzzing	FNN	[65]
FuzzGuard	Predicting the reachability of inputs before executing the target program, filtering out unreachable inputs to enhance the performance of fuzzing	CNN	[67]
RLF	Calculating the distance between execution paths and the target, guiding the selection of test samples	DQL	[68]
SpeedNeuzz	Reducing the randomness in test case mutations to generate high-quality inputs	DNN	[69]
Dr.PathFinder	Allowing efficient memory usage and alleviating the state explosion problem	RNN, DQL	[70]
V-Fuzz	Efficiently and rapidly discovering errors in binary programs within a limited time	GEN	[71]

5.3. Challenges And Opportunities

Machine learning and deep learning drive innovations in cybersecurity fuzzing, presenting a critical topic for discussion. The variety in software complicates vulnerability detection, especially in large systems with novel weaknesses. Challenges persist:

- Diverse data types, from integers to images, require models to adapt to dynamic software inputs, which may shift in type or format during operation.
- Models must discern complex vulnerability patterns across multiple points and layers, demanding incremental or online training to stay effective against emerging threats.
- The efficacy of machine learning in fuzz testing hinges on the quality and quantity of training data, often scarce in specialized domains. Manual data collection is laborious, and dataset quality crucially influences model training and generalization. Ideal datasets must be unbiased, with a balanced mix of positive and negative samples.

In grey-box or white-box fuzzing, limited program analysis restricts feature extraction and utilization. Future models should leverage advanced capabilities to extract and use program features more accurately and holistically, including semantic information. These improvements will enhance the precision and effectiveness of vulnerability detection, thus strengthening software security.

6. Conclusion

Fuzz testing, a key vulnerability detection method, has gained attention for its automation and scalability. The integration of machine learning has invigorated this field, bolstering efficiency and precision in identifying vulnerabilities. These technologies enable models to generate targeted test cases, increasing the likelihood of exposing vulnerabilities and adapting to evolving threats.

The paper examines the growth of learning-based fuzz testing. It starts with an introduction to fuzz testing and traditional methods, then explores learning-based enhancements at various stages, including test case development and result analysis. It underscores learning-based methods' role in refining vulnerability detection. The conclusion calls for continued research in this area to further advance security assessments and devise sophisticated solutions for emerging security challenges.

References

[1] Vilela, P., Machado, M. and Wong, W.E., 2002. Testing for security vulnerabilities in software. Software Engineering and Applications.

[2] Wei W. Billions of Android Devices Vulnerable to Privilege Escalation Except Android 5.0 Lollipop, 2014 [Online]. Available: http://thehackernews.com/2014/ 11/ billions-of-android-devices-vulnerable.html

[3] Autili, M., Malavolta, I., Perucci, A., Scoccia, G.L. and Verdecchia, R., 2021. Software engineering techniques for statically analyzing mobile apps: research trends, characteristics, and potential for industrial adoption.

Journal of Internet Services and Applications, 12, pp.1-60.
[4] Huang, L. and Song, Y.T., 2008, December. A dynamic impact analysis approach for object-oriented programs. In 2008 Advanced Software Engineering and Its Applications (pp. 217-220). IEEE.
[5] Min, Z. and Min, F., 2014, June. Automated test generation on path-based symbolic execution. In 2014 IEEE 5th International Conference on Software Engineering and Service Science (pp. 845-848). IEEE.
[6] Gaston, C., Aiguier, M., Bahrami, D. and Lapitre, A., 2009, September. Symbolic execution techniques extended to systems. In 2009 Fourth International Conference on Software Engineering Advances (pp. 78-85). IEEE.
[7] Petsios, T., Zhao, J., Keromytis, A.D. and Jana, S., 2017, October. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In Proceedings of the 2017 ACM SIGSAC conference on computer and communications security (pp. 2155-2168).
[8] Sutton, M., Greene, A. and Amini, P., 2007. Fuzzing: brute force vulnerability discovery. Pearson Education.
[9] Gupta, M.J. and Sehgal, P., 2024. Optimizing Credit Card Fraud Detection: Classifier Performance and Feature Selection Empowered by Grasshopper Algorithm. International Journal of Performability Engineering, 20(3), pp. 177-185.
[10] Bhandari, R., Singla, S., Sharma, P. and Kang, S.S., 2024. AINIS: An Intelligent Network Intrusion System. International Journal of Performability Engineering, 20(1), pp. 24-31.
[11] Herrera, A., Gunadi, H., Hayes, L., Magrath, S., Friedlander, F., Sebastian, M., Norrish, M. and Hosking, A.L., 2019. Corpus distillation for effective fuzzing: A comparative evaluation. arxiv preprint arxiv:1905.13055.
[12] Zheng, Y., Wen, H., Cheng, K., Song, Z.W., Zhu, H.S. and Sun, L.M., 2019. A survey of IoT device vulnerability mining techniques. Journal of Cyber Security, 4(5), pp.61-75.
[13] Zhou, W., Jia, Y., Peng, A., Zhang, Y. and Liu, P., 2018. The effect of IoT new features on security and privacy: New threats, existing solutions, and challenges yet to be solved. IEEE Internet of things Journal, 6(2), pp.1606-1616.
[14] Miller, B.P., Fredriksen, L. and So, B., 1990. An empirical study of the reliability of UNIX utilities. Communications of the ACM, 33(12), pp.32-44.
[15] Böttinger, K., 2017, May. Guiding a colony of black-box fuzzers with chemotaxis. In 2017 IEEE Security and Privacy Workshops (SPW) (pp. 11-16). IEEE.
[16] Woo, M., Cha, S.K., Gottlieb, S. and Brumley, D., 2013, November. Scheduling black-box mutational fuzzing. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (pp. 511-522).
[17] Aitel, D. An introduction to SPIKE, the fuzzer creation kit. in Proc. Black Hat USA, 2002. [Online]. Available: https://www.blackhat.com/presentations/bh-usa-02/bh-us-02-aitel-spike.ppt
[18] Zhao, L., Duan, Y. and XUAN, J., 2019. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. Network and Distributed System Security Symposium (NDSS).
[19] Cadar, C., Dunbar, D. and Engler, D.R., 2008, December. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In OSDI (Vol. 8, pp. 209-224).
[20] Godefroid, P., Levin, M.Y. and Molnar, D.A., 2008, February. Automated whitebox fuzz testing. In NDSS (Vol. 8, pp. 151-166).
[21] Ganesh, V., Leek, T. and Rinard, M., 2009, May. Taint-based directed whitebox fuzzing. In 2009 IEEE 31st International Conference on Software Engineering (pp. 474-484). IEEE.
[22] Chen, H., Xue, Y., Li, Y., Chen, B., Xie, X., Wu, X. and Liu, Y., 2018, October. Hawkeye: Towards a desired directed grey-box fuzzer. In Proceedings of the 2018 ACM SIGSAC conference on computer and communications security (pp. 2095-2108).
[23] Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C. and Bos, H., 2017, February. VUzzer: Application-aware Evolutionary Fuzzing. In NDSS (Vol. 17, pp. 1-14).
[24] Lemieux, C. and Sen, K., 2018, September. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In Proceedings of the 33rd ACM/IEEE international conference on automated software engineering (pp. 475-485).
[25] Eddington, M., 2011. Peach fuzzing platform. Peach Fuzzer, 34, pp.32-43.
[26] Zalewski, M., 2017. American fuzzy lop.
[27] Böhme, M., Pham, V.T., Nguyen, M.D. and Roychoudhury, A., 2017, October. Directed greybox fuzzing. In Proceedings of the 2017 ACM SIGSAC conference on computer and communications security (pp. 2329-2344).
[28] Yue, T., Wang, P., Tang, Y., Wang, E., Yu, B., Lu, K. and Zhou, X., 2020. {EcoFuzz}: Adaptive {Energy-Saving} greybox fuzzing as a variant of the adversarial {Multi-Armed} bandit. In 29th USENIX Security Symposium (USENIX Security 20) (pp. 2307-2324).
[29] Böhme, M., Pham, V.T. and Roychoudhury, A., 2016, October. Coverage-based greybox fuzzing as markov chain. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (pp. 1032-1043).
[30] Lemieux, C. and Sen, K., 2017. Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage. arxiv preprint arxiv:1709.07101.
[31] Chen, K., Zhang, Y. and Liu, P., 2016. Dynamically discovering likely memory layout to perform accurate fuzzing. IEEE Transactions on Reliability, 65(3),

pp.1180-1194.
[32] Vinesh, N. and Sethumadhavan, M., 2020. Confuzz—a concurrency fuzzer. In First International Conference on Sustainable Technologies for Computational Intelligence: Proceedings of ICTSCI 2019 (pp. 667-691). Springer Singapore.
[33] Padhye, R., Lemieux, C. and Sen, K., 2019, July. Jqf: Coverage-guided property-based testing in java. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (pp. 398-401).
[34] Chen, Y., Li, P., Xu, J., Guo, S., Zhou, R., Zhang, Y., Wei, T. and Lu, L., 2020, May. Savior: Towards bug-driven hybrid testing. In 2020 IEEE Symposium on Security and Privacy (SP) (pp. 1580-1596). IEEE.
[35] Wang, Y., Jia, X., Liu, Y., Zeng, K., Bao, T., Wu, D. and Su, P., 2020, February. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. In NDSS.
[36] Ohe, H. and Chang, B.M., 2005. An exception monitoring system for java. In Rapid Integration of Software Engineering Techniques: First International Workshop, RISE 2004, Luxembourg-Kirchberg, Luxembourg, November 26, 2004. Revised Selected Papers 1 (pp. 71-81). Springer Berlin Heidelberg.
[37] Abrantes, J., Coelho, R. and Bonifácio, R., 2015. DAEH: A Tool for Specifying and Monitoring the Exception Handling Policy. International Journal of Software Engineering and Knowledge Engineering, 25(09n10), pp.1515-1530.
[38] Serebryany, K., 2015. libfuzzer–a library for coverage-guided fuzz testing. LLVM project, p.34.
[39] Takanen, A., Demott, J.D., Miller, C. and Kettunen, A., 2018. Fuzzing for software security testing and quality assurance. Artech House.
[40] Wang, Y., Jia, P., Liu, L., Huang, C. and Liu, Z., 2020. A systematic review of fuzzing based on machine learning techniques. PloS one, 15(8), p.e0237749.
[41] Gan, S., Zhang, C., Qin, X., Tu, X., Li, K., Pei, Z. and Chen, Z., 2018, May. Collafl: Path sensitive fuzzing. In 2018 IEEE Symposium on Security and Privacy (SP) (pp. 679-696). IEEE.
[42] Nguyen, T.D., Pham, L.H., Sun, J., Lin, Y. and Minh, Q.T., 2020, June. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (pp. 778-788).
[43] Samuel, A.L., 2000. Some studies in machine learning using the game of checkers. IBM Journal of research and development, 44(1.2), pp.206-226.
[44] Cox, D.R., 1958. The regression analysis of binary sequences. Journal of the Royal Statistical Society Series B: Statistical Methodology, 20(2), pp.215-232.
[45] Cover, T. and Hart, P., 1967. Nearest neighbor pattern classification. IEEE transactions on information theory, 13(1), pp.21-27.
[46] MacQueen, J., 1967, June. Some methods for classification and analysis of multivariate observations. In Proceedings of the fifth Berkeley symposium on mathematical statistics and probability (Vol. 1, No. 14, pp. 281-297).
[47] Shi, J. and Malik, J., 2000. Normalized cuts and image segmentation. IEEE Transactions on pattern analysis and machine intelligence, 22(8), pp.888-905.
[48] Radosavljevic, A. and Anderson, R.P., 2014. Making better Maxent models of species distributions: complexity, overfitting and evaluation. Journal of biogeography, 41(4), pp.629-643.
[49] Sheela, K.G. and Deepa, S.N., 2013. Review on methods to fix number of hidden neurons in neural networks. Mathematical problems in engineering, 2013.
[50] Hinton, G.E. and Salakhutdinov, R.R., 2006. Reducing the dimensionality of data with neural networks. science, 313(5786), pp.504-507.
[51] Nair, V. and Hinton, G.E., 2010. Rectified linear units improve restricted boltzmann machines. In Proceedings of the 27th international conference on machine learning (ICML-10) (pp. 807-814).
[52] Ito, Y., 1991. Representation of functions by superpositions of a step or sigmoid function and their applications to neural network theory. Neural Networks, 4(3), pp.385-394.
[53] Fan, E., 2000. Extended tanh-function method and its applications to nonlinear equations. Physics Letters A, 277(4-5), pp.212-218.
[54] Choi, G., Jeon, S., Cho, J. and Moon, J., 2023. A Seed Scheduling Method With a Reinforcement Learning for a Coverage Guided Fuzzing. IEEE Access, 11, pp.2048-2057.
[55] Wang, J., Song, C. and Yin, H., 2021. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing.
[56] Li, Z.T., Cheng L. and Zhang Y. Seed Generation for Fuzzing Based on Deep Learning, Computer Systems Applications,2019,28(4):9–17.
[57] Godefroid, P., Peleg, H. and Singh, R., 2017, October. Learn&fuzz: Machine learning for input fuzzing. In 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 50-59). IEEE.
[58] Nichols, N., Raugas, M., Jasper, R. and Hilliard, N., 2017. Faster fuzzing: Reinitialization with deep neural models. arxiv preprint arxiv:1711.02807.
[59] Karamcheti, S., Mann, G. and Rosenberg, D., 2018, January. Adaptive grey-box fuzz-testing with thompson sampling. In Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security (pp. 37-47).
[60] Böttinger, K., Godefroid, P. and Singh, R., 2018, May. Deep reinforcement fuzzing. In 2018 IEEE Security and Privacy Workshops (SPW) (pp. 116-122). IEEE.
[61] Zhang, Z., Cui, B. and Chen, C., 2021. Reinforcement learning-based fuzzing technology. In Innovative

Mobile and Internet Services in Ubiquitous Computing: Proceedings of the 14th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS-2020) (pp. 244-253). Springer International Publishing.

[62] Sun, X., Wang, W., Liu, X., Fan, J., Li, Z., Song, Y. and Qin, Z., 2022, December. VecSeeds: Generate fuzzing testcases from latent vectors based on VAE-GAN. In 2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom) (pp. 953-958). IEEE.

[63] Lee, M., Cha, S. and Oh, H., 2023, May. Learning seed-adaptive mutation strategies for greybox fuzzing. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) (pp. 384-396). IEEE.

[64] Shao, J., Zhou, Y., Liu, G. and Zheng, D., 2023, May. Optimized Mutation of Grey-box Fuzzing: A Deep RL-based Approach. In 2023 IEEE 12th Data Driven Control and Learning Systems Conference (DDCLS) (pp. 1296-1300). IEEE.

[65] She, D., Pei, K., Epstein, D., Yang, J., Ray, B. and Jana, S., 2019, May. Neuzz: Efficient fuzzing with neural program smoothing. In 2019 IEEE Symposium on Security and Privacy (SP) (pp. 803-817). IEEE.

[66] Cheng, L., Zhang, Y., Zhang, Y., Wu, C., Li, Z., Fu, Y. and Li, H., 2019, May. Optimizing seed inputs in fuzzing with machine learning. In 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion) (pp. 244-245). IEEE.

[67] Zong, P., Lv, T., Wang, D., Deng, Z., Liang, R. and Chen, K., 2020. FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In 29th USENIX security symposium (USENIX security 20) (pp. 2255-2269).

[68] Liang, X. and Xiao, T., 2022, October. Rlf: Directed fuzzing based on deep reinforcement learning. In 2022 International Conference on Machine Learning, Control, and Robotics (MLCR) (pp. 127-133). IEEE.

[69] Li, Y., Xiao, X., Zhu, X., Chen, X., Wen, S. and Zhang, B., 2020, December. Speedneuzz: Speed up neural program approximation with neighbor edge knowledge. In 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom) (pp. 450-457). IEEE.

[70] Jeon, S. and Moon, J., 2022. Dr. PathFinder: hybrid fuzzing with deep reinforcement concolic execution toward deeper path-first search. Neural Computing and Applications, 34(13), pp.10731-10750.

[71] Li, Y., Ji, S., Lyu, C., Chen, Y., Chen, J., Gu, Q., Wu, C. and Beyah, R., 2020. V-fuzz: Vulnerability prediction-assisted evolutionary fuzzing for binary programs. IEEE transactions on cybernetics, 52(5), pp.3745-3756.

[72] Ming, L., Zhao, G., Huang, M., Pang, L., Li, J., Zhang, J., Li, D. and Lu, S., 2018, June. Remote Protocol Vulnerability Discovery for Intelligent Transportation Systems (ITS). In 2018 IEEE Third International Conference on Data Science in Cyberspace (DSC) (pp. 923-929). IEEE.

[73] Tripathi, S., Grieco, G. and Rawat, S., 2017, December. Exniffer: Learning to prioritize crashes by assessing the exploitability from memory dump. In 2017 24th Asia-Pacific Software Engineering Conference (APSEC) (pp. 239-248). IEEE.

[74] Zhang, L. and Thing, V.L., 2018, October. Assisting vulnerability detection by prioritizing crashes with incremental learning. In TENCON 2018-2018 IEEE Region 10 Conference (pp. 2080-2085). IEEE.

[75] Yan, G., Lu, J., Shu, Z. and Kucuk, Y., 2017, August. Exploitmeter: Combining fuzzing with machine learning for automated evaluation of software exploitability. In 2017 IEEE Symposium on Privacy-Aware Computing (PAC) (pp. 164-175). IEEE.

[76] You, W., Zong, P., Chen, K., Wang, X., Liao, X., Bian, P. and Liang, B., 2017, October. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In Proceedings of the 2017 ACM SIGSAC conference on computer and communications security (pp. 2139-2154).

[77] Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., Ulrich, F. and Whelan, R., 2016, May. Lava: Large-scale automated vulnerability addition. In 2016 IEEE symposium on security and privacy (SP) (pp. 110-121). IEEE.

[78] Pesch, R.H. and Osier, J.M., 1993. The GNU binary utilities. Free Software Foundation.

[79] Song, J. and Alves-Foss, J., 2015. The darpa cyber grand challenge: A competitor's perspective. IEEE Security & Privacy, 13(6), pp.72-76.

[80] Github repository cb-multios, 2018, [online] Available: https://github.com/trailofbits/cb-multios.

[81] Waymark, C., Walker, K.A., Boone, C.D. and Bernath, P.F., 2013. ACE-FTS version 3.0 data set: validation and data processing update. Annals of Geophysics, 56.