Received 4 November 2023, accepted 19 December 2023, date of publication 26 December 2023, date of current version 31 January 2024.

Digital Object Identifier 10.1109/ACCESS.2023.3347652

SURVEY

Machine Learning-Based Fuzz Testing Techniques: A Survey

AO ZHANG[1], YIYING ZHANG[1], YAO XU[1], CONG WANG[1], AND SIWEI LI[2]
[1]College of Artificial Intelligence, Tianjin University of Science and Technology, Tianjin 300457, China
[2]State Grid Information and Telecommunication Company Ltd., Beijing 102200, China

Corresponding author: Yiying Zhang (yiyingzhang@tust.edu.cn)

ABSTRACT Fuzz testing is a vulnerability discovery technique that tests the robustness of target programs by providing them with unconventional data. With the rapid increase in software quantity, scale and complexity, traditional fuzzing has revealed issues such as incomplete logic coverage, low automation level and insufficient test cases. Machine learning, with its exceptional capabilities in data analysis and classification prediction, presents a promising approach for improve fuzzing. This paper investigates the latest research results in fuzzing and provides a systematic review of machine learning-based fuzzing techniques. Firstly, by outlining the workflow of fuzzing, it summarizes the optimization of different stages of fuzzing using machine learning. Specifically, it focuses on the application of machine learning in the preprocessing phase, test case generation phase, input selection phase and result analysis phase. Secondly, it mentally focuses on the optimization methods of machine learning in the process of mutation, generation and filtering of test cases and compares and analyzes its technical principles. Furthermore, it analyzes the performance gains brought by applying machine learning techniques to fuzzing, mainly including coverage, vulnerability detection capability, efficiency and effectiveness of test cases. Lastly, it concludes by summarizing the challenges and difficulties in combining machine learning with fuzzing and presents prospects for future trends in this field.

INDEX TERMS Vulnerability discovery, fuzzing, machine learning.

I. INTRODUCTION

In recent years, there has been a proliferation of network attacks and a rapid increase in the number of vulnerabilities, leading to potential risks such as information leakage or loss. Vulnerability discovery techniques aim to identify and patch vulnerabilities before they are exploited by attackers [1], effectively reducing security threats and maintaining the secure operation of networks. Fuzz testing, as an effective method for vulnerability discovery, attempts to trigger program anomalies by automatically or semi-automatically generating test cases, monitoring target program execution and providing feedback to adjust the generation of test cases. It offers the advantages of easy deployment and broad applicability. The concept of fuzz testing was initially proposed by Miller in 1990 [2], who designed a tool called Fuzz to test the robustness of target programs using unconventional data. The value of fuzzing has been explored, black-box [3], white-box [4] and gray-box fuzzers [5] have appeared one after another. Countless scholars have carried out continuous improvement and enhancement, and the coverage rate and anomaly triggering ability have been improved to different degrees. However, traditional fuzzing still faces several challenges, such as an insufficient number of existing test cases, weak ability of generated test cases to trigger vulnerabilities, the lack of differentiation between test case weights during input selection, and a relatively high degree of blindness during the testing process.

With the remarkable performance of machine learning techniques in statistical learning, natural language processing and pattern recognition, researchers have applied these techniques to the field of cybersecurity, including the detection of malicious code [6] and intrusion detection [7]. Machine learning can automatically learn grammar rules that conform to syntax specifications from a large number of samples, effectively addressing classification problems in fuzzing, such as

The associate editor coordinating the review of this manuscript and approving it for publication was Xinyu Du.

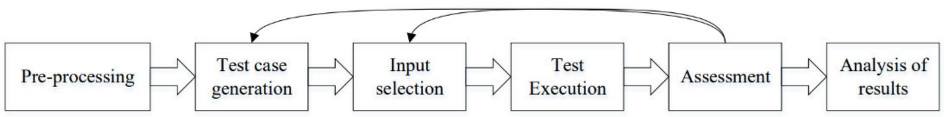

FIGURE 1. **Basic flow of fuzzing.**

determining the validity of generated test cases and the usability of seeds for mutation. Furthermore, machine learning can reduce manual effort and minimize the time overhead of fuzzing. Therefore, combining machine learning with fuzzing provides new ideas and methods for alleviating the bottlenecks of traditional fuzzing techniques. How to balance the advantages of both to better enhance vulnerability detection is still an area that requires further research. This paper focuses on the background of machine learning, analyzes and reviews a large body of literature on the combination of machine learning and fuzzing. Taking the basic process of fuzzing as a vein, it introduces various improved methods of fuzzing implementation based on different machine learning models, comparing and analyzing their enhancements and improvements. Furthermore, it introduces the performance gain of different machine learning methods for fuzzing and demonstrates the effectiveness of machine learning for fuzzing improvement. Moreover, it identifies existing issues in applying machine learning techniques to fuzzing and provides insights into future development trends.

The primary contributions of our work can be summarized as follows:

(1) This paper refers to and examines a large amount of relevant literature and highlights the latest research results in the past five years, which can better grasp the future direction of the fuzzing field. Not only that, this paper analyzes and organizes research on fuzzing in different areas, such as fuzzing in the Internet of Things, web applications, compilers and deep learning models, which encompasses common areas where fuzzing can be used.

(2) This paper focuses on the workflow of fuzzing and introduces the application of machine learning methods in four different stages: preprocessing, test case generation, input selection and result analysis. It compares and contrasts various improvement techniques, explaining their underlying technical principles and the resulting optimization enhancements. Finally, it provides a comprehensive summary of the performance gains achieved through the utilization of machine learning algorithms. It facilitates readers to better understand the overall workflow of fuzzing and helps them to carry out in-depth research.

(3) By comparing different improvement methods, the problems and challenges in this field are analyzed and summarized, and the possible hot research directions in the field of fuzzing in the background of machine learning are put forward.

Section II provides a brief overview of the basic process of fuzzing. Section III introduces the application of machine learning techniques at different stages of fuzzing, comparing and analyzing the strengths and technical principles of different fuzzing tools. Furthermore, in Section IV, the performance gains to fuzzing from different machine learning approaches are theorized. Next, the challenges faced by existing fuzzing techniques are analyzed, and in Section V, the existing liberation schemes are presented as well as an insight into the future trends in the field. Finally, Section VI summarizes and concludes the work presented in this paper.

II. OVERVIEW OF FUZZING

A. BASIC FLOW OF FUZZING

Fuzzing involves constructing a large number of illegal test inputs, fuzz testing the target program, monitoring its execution, observing and recording any abnormal behavior, analyzing the cause of abnormality or crashes, and finally detecting vulnerabilities. The basic flow of fuzzing can be divided into six parts: pre-processing, test case generation, input selection, test execution, evaluation and result analysis [8], as shown in Figure 1.

The preprocessing stage primarily involves collecting relevant information about the target program and specifying the strategy for fuzzing to assist the fuzzing tool in detecting or observing the target program. This stage typically relies on program analysis techniques such as instrumentation [9] [10], symbolic execution [11], [12] and taint analysis [13], [14]. Existing research efforts have focused on integrating one or more of these techniques into hybrid fuzzing to improve overall performance. For example, Risk-AFL [10] proposes a risk-guided seed selection method based on AFL. During program operation, the risk fitness of the seeds is calculated based on the risky functions and function calls on the program execution path by means of the instrument, and the seed selection strategy of AFL is improved accordingly. Intriguer [11] optimizes symbolic execution by utilizing field-level knowledge to more effectively simulate symbolically relevant instructions. TaintPoint [14] applies to the seed mutation stage of general fuzzing and obtains more accurate taint analysis results to guide mutation.

The test case generation phase is mainly to obtain a large number of test inputs, and based on the relevant information obtained in the preprocessing phase, select the appropriate generation method or mutation strategy to construct a large number of test cases which are suitable for the target program. The test case generation phase consists of seed selection, mutation strategy scheduling and test case generation. Seed selection is a process of evaluating the likelihood that a seed could trigger a program anomaly and prioritizing

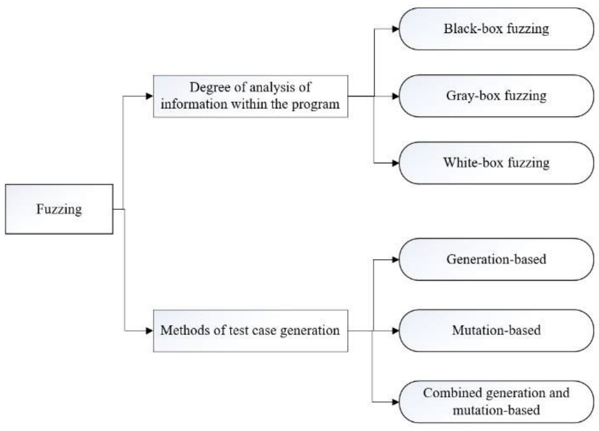
Fuzzing
Degree of analysis of information within the program
Methods of test case generation
Black-box fuzzing
Gray-box fuzzing
White-box fuzzing
Generation-based
Mutation-based
Combined generation and mutation-based

FIGURE 2. Classification of fuzzing.

the higher quality seeds for mutation, so as to reduce the number of invalid test cases generated. Mutation strategy scheduling is similar to the idea of seed selection, prioritizing near-excellent mutation strategies to improve test case bypass and reduce mutation blindness. Test case generation can be categorized based on the generation method as generation-based, mutation-based and based on a combination of the two. There are more studies on the application of machine learning techniques to this phase, which are described in detail below in the context of existing studies.

The input selection phase screens constructed test cases before execution, eliminates invalid cases, and reduces computation time.

The test execution phase involves entering the constructed test cases into the target program, monitoring the execution of the program, and identifying and recording abnormal state changes.

The evaluation phase selects suitable indicators to assess fuzzing effectiveness and vulnerability mining ability. The results are fed back to the test case generation phase to optimize the fuzzer.

The result analysis phase analyzes output results after fuzzing execution. Based on abnormal program states, causes and defect categories are identified to better detect vulnerabilities.

B. CLASSIFICATION OF FUZZING

Fuzzing can be classified according to different classification bases. As shown in Figure 2. black-box fuzzing, gray-box fuzzing and white-box fuzzing can be classified according to the degree of analysis of the information inside the program. According to the way of test case generation, they can be classified as generation-based fuzzing, Mutation-based fuzzing and combined generation and Mutation-based fuzzing.

1) BLACK-BOX, GRAY-BOX AND WHITE-BOX

Black-box fuzzing cannot analyze the internal state and structure of the target, but only obtains internally irrelevant information such as the input data format of the target. In addition, during the testing process, black-box fuzzing cannot track the execution status inside the target and can only determine the status of the target by detecting the output data of the target [8]. Black-box fuzzing tools are simple to implement and fast to test, and are more suitable for target programs with highly structured input data, as well as complex and difficult to analyze target programs. However, its detection is not satisfactory.

White-box fuzzing is the opposite of black-box fuzzing in that it obtains sufficient internal information about the target to generate high-quality test inputs. White-box fuzzing has better performance in the coverage of programs and in the detection of deep vulnerabilities. However, the method can seriously affect the efficiency of fuzzing because a detailed and comprehensive analysis of the target program consumes a lot of resources.

Gray-box fuzzing is between black-box and white-box, and only part of the in-ternal information of the target is obtained for fuzzing. The method aims to obtain satisfactory test results with limited internal information and a good testing strategy. Compared to both black-box and white-box, gray-box is more flexible and has more advantages. Gray-box fuzzing can find a suitable balance between detection capability and resource consumption to obtain the best detection results.

2) GENERATION-BASED, MUTATION-BASED AND COMBINATION OF GENERATION-BASED AND MUTATION-BASED

The generation-based test case generation approach is mainly based on the known input case format or protocol syntax to generate new test cases. The method needs to generate and process inputs according to the specification of the expected input format or protocol.

Mutation-based test case generation is based on existing test cases with certain mutation methods (e.g., bit-flip, byte-inversion, arithmetic increment/decrement and splicing operations) [15]. In general, blind mutation or manipulation of data generates a multitude of invalid test cases. The introduction of machine learning techniques can guide mutation operations and improve the quality of generated test cases.

The test case generation approach based on a combination of generation and mutation considers both variation and generation approaches for different test scenarios to maximize their advantages and better guide test case generation.

III. FUZZING IN THE CONTEXT OF MACHINE LEARNING

Existing fuzzing tools have problems such as low degree of automation and weak vulnerability triggering ability of fuzzing test cases, the excellent data processing and classification prediction ability of machine learning technology is utilized and applied to different stages of fuzzing, which can realize the quality optimization of fuzzing test cases and the efficiency improvement of vulnerability detection.

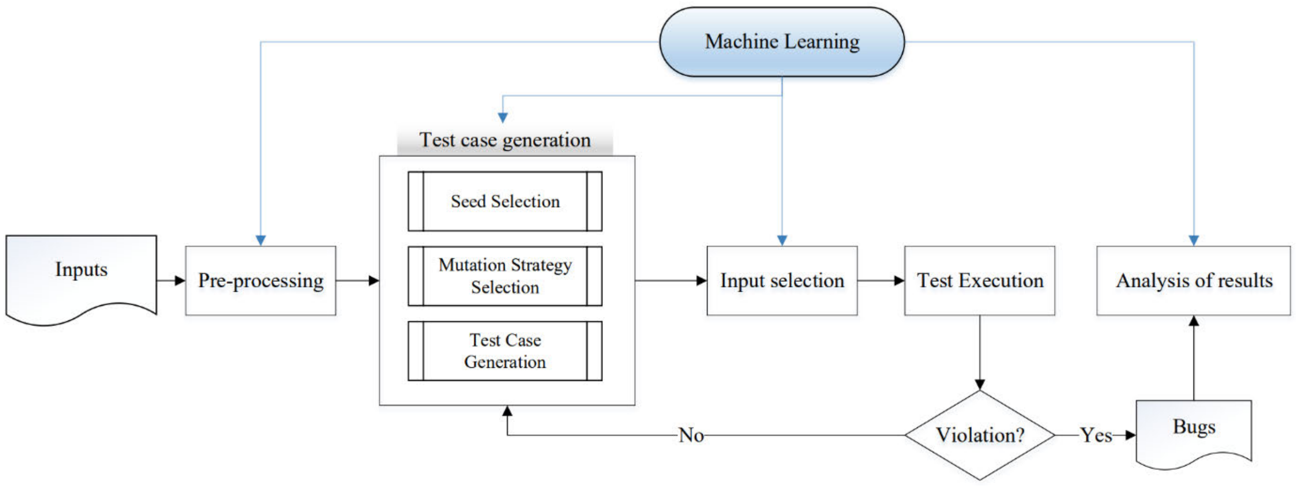

FIGURE 3. Schematic diagram of machine learning-based fuzzing process.

Currently, machine learning is primarily applied in the preprocessing stage, test case generation stage, input selection stage and result analysis stage of fuzzing. The basic process of fuzzing combined with machine learning algorithms is illustrated in Figure 3. In the preprocessing stage, machine learning algorithms can analyze and predict the program information obtained during preprocessing, enhancing the effectiveness of program analysis techniques combined with fuzzing. In the test case generation stage, machine learning algorithms can be used to optimize seed selection, guide mutation strategies and mutation point selection, facilitating seed and test case generation. In the input selection stage, machine learning algorithms can filter and select test inputs. For example, machine learning algorithms can be used for vulnerability prediction and classification processing of test inputs, prioritizing the selection of test inputs that are more likely to trigger vulnerabilities when passed into the target program. In the result analysis stage, machine learning can efficiently and reasonably analyze the numerous test results, enabling the identification of true vulnerabilities within a large number of crashes and anomalies.

Based on the general process of fuzzing, this section provides a detailed introduction to the application and improvements of machine learning algorithms in different stages of fuzzing. It systematically elucidates fuzzing methods based on different machine learning techniques and intuitively presents the performance of different fuzzing models in tabular form.

A. PRE-PROCESSING

The preprocessing stage of fuzzing can utilize program analysis techniques such as instrumentation and symbolic execution to extract program features or runtime information, providing support for generating subsequent test cases. For instance, Pangolin introduces the concept of incremental fuzzing, which involves a polyhedral path abstraction method to accelerate constraint solving in cooperative execution [16]. The results of constraint solving are then used to guide the subsequent fuzzing process, improving the efficiency of vulnerability discovery. Liu et al. propose SiCsFuzzer, a tool that adopts a sparse instrumentation-based tracing strategy combined with “warm-up” optimization to improve the efficiency of fuzzing for closed-source programs by eliminating the redundancy overhead in the coverage tracking process of closed-source software without compromising coverage calculation accuracy [17]. Meanwhile, Xiao et al. leverage runtime information obtained through instrumentation as rewards in a deep reinforcement learning network, guiding the generation of more targeted and directed test cases [18].

Many researchers have dedicated efforts to combine these techniques with fuzzing, complementing each other. The hybrid fuzzing methods overcome difficulties by employing one technique when the other encounters bottlenecks, leading to higher coverage and the exploration of deep program regions to discover deep-seated vulnerabilities. MPFuzz proposes a hybrid fuzzing technique that combines symbolic simulation and grammar-based [19]. Symbolic simulation is used to guide the testing process for achieving high coverage, while grammar-based fuzzing generates test instructions conforming to the syntax specifications of microprocessor RTL designs. The combination of both techniques efficiently generates test instructions for microprocessor RTL designs. Furthermore, the utilization of deep learning techniques can learn code space features suitable for both techniques before program execution, serving as guidance for hybrid fuzzing. This approach effectively enhances code coverage and significantly improves defect detection capabilities. Gao et al. introduce a hybrid testing method based on deep learning [20]. The algorithm flow is illustrated in Figure 4. Given a program, a graphical representation of its paths is constructed, and a gated graph neural network (GGNN) model is

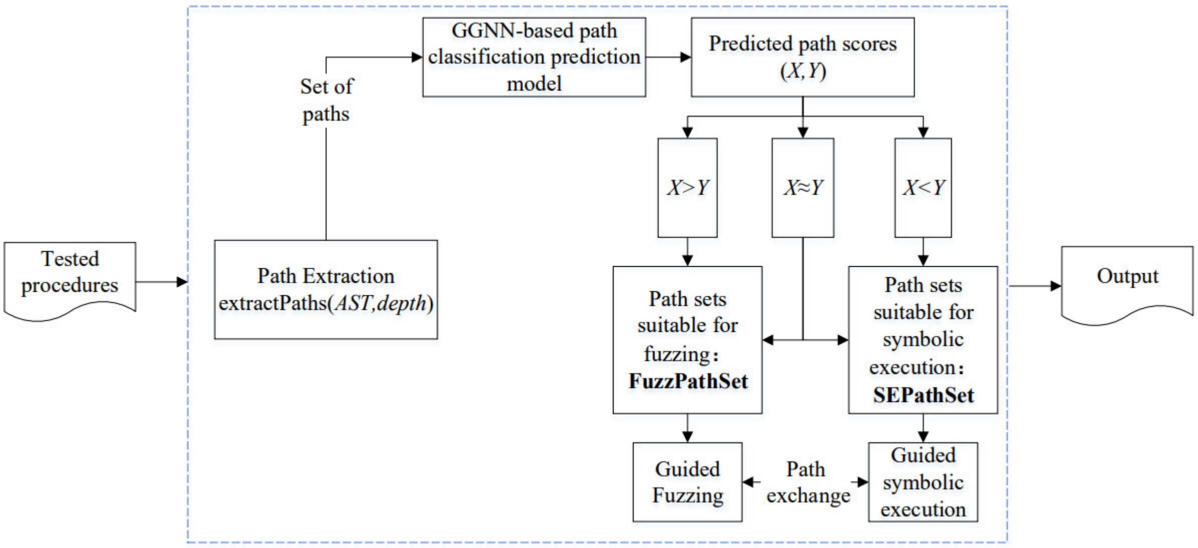
Tested procedures
Path Extraction extractPaths(AST,depth)
Set of paths
GGNN-based path classification prediction model
Predicted path scores (X,Y)
X>Y
X≈Y
X<Y
Path sets suitable for fuzzing: FuzzPathSet
Path sets suitable for symbolic execution: SEPathSet
Guided Fuzzing
Path exchange
Guided symbolic execution
Output

FIGURE 4. The overall framework of deep learning-based hybrid fuzzing [20].

employed to predict whether a path is suitable for fuzzing or symbolic execution. This leads to the development of Smart-FuSE, which guides symbolic execution or fuzzing tools attempt to execute the set of paths preferentially. Moreover, considering the inaccuracies of model predictions, Smart-FuSE proposes a hybrid mechanism that, among the set of predicted paths suitable for executing fuzzing, passes the paths uncovered by the fuzzing to symbolic execution, and uses the symbolic execution technique to traverse the paths for further improvement of the overall coverage.

Where AST denotes abstract syntax tree, *depth* denotes path depth, X denotes the score suitable for fuzzing, and Y denotes the score suitable for symbolic execution.

Hybrid fuzzing that incorporates one or more techniques has become a new research branch aiming to combine the advantages of multiple techniques and enhance vulnerability detection capability. Many studies have improved combinatorial strategy for hybrid fuzzing using "optimal strategy," "discriminative dispatch strategy," and "Priority Based Path Searching method" [21], [22], [23]. However, in the face of large software vulnerability mining, the operational costs of hybrid fuzzing tools are invariably high due to path explosion problems caused by program branches. Applying machine learning techniques to mitigate inherent shortcomings of a single technique in hybrid fuzzing and thus improve fuzzing performance is also a possible research direction for future fuzzing development.

B. TEST CASE GENERATION

In the field of test case generation, machine learning can be applied to scenarios such as mutation position selection, mutation strategy schedule and structured test case generation. It effectively overcomes the limitations of traditional fuzzing techniques, including blind mutation, ineffective sample generation and reliance on manual involvement, thereby greatly improving the quality of generated samples.

Machine learning has been widely used in this stage in existing research. Therefore, in this section, we will divide the discussion based on the problems addressed by machine learning algorithms, and specifically introduce their applications in mutation strategy scheduling, seed selection and test case generation problems. Not only that, we will also analyze how different machine learning models contribute to the improvement and enhancement of the efficiency and capability of fuzzing.

1) SEED SELECTION

Seeds can be mutated using various mutation operations to generate test cases. The quality of seeds is one of the important factors that influence the effectiveness of fuzzing. Selecting well-formed seeds can significantly save CPU time, and mutations based on well-formed seed inputs are more likely to generate test cases that reach deeper levels of the program.

Wei Xiao et al. proposed a test case classification method based on LSTM neural networks, where the test cases are passed through LSTM and linear layers, resulting in two output nodes. The activation function is applied to obtain the probability of the input belonging to a certain class in the label set [24]. The model is trained using the test cases and their coverage states, and after multiple training iterations, an accurate prediction model for input categories is obtained. This model is used to learn high-level features of the input file structure and assess the value of seeds. By prioritizing the mutation of high-value seeds, the seed selection process is guided. NeuFuzz proposes a hidden pattern learning approach for vulnerable program paths based on LSTM models. Firstly, the seed files are subjected to vulnerability detection. Then,

TABLE 1. Seed selection.

Literature	Model	Improvement Points	Test Target	Year
[24]	LSTM	Use models to learn high-level features of documents and determine seed values.	Documents	2022
NeuFuzz [25]	LSTM	Vulnerability detection of seed files using a hidden model of model learning vulnerable program paths.	LAVA-M and real-world applications	2019
V-Fuzz [26]	GNN	The model is used to predict the fragility probability of each function of the target program and, in turn, to predict the inputs that are more likely to arrive at a fragile location.	Juliet Test Suite and real-world applications	2022

the fuzzer is instructed to prioritize the vulnerable paths identified by the trained model and allocate more mutation energy to them. This method achieves maximum efficiency in defect discovery [25]. With the increasing research on graph embedding networks, V-Fuzz applies it to fuzzing and proposes a fuzzing framework that combines graph embedding networks and evolutionary algorithms. This framework enables efficient testing of binary programs without requiring source code [26]. V-Fuzz proposes a vulnerability detection model based on graph embedding networks, which outputs predicted vulnerability probability values for each function in the target program. These probabilities are subsequently used to calculate fitness scores. During the test execution using user-defined initial seeds, evolutionary algorithms compute fitness scores for each seed and select seeds with high fitness scores and triggering crashes as new seed inputs. Subsequently, these seeds are mutated to generate more test inputs that have the potential to discover vulnerabilities.

2) MUTATION STRATEGY SCHEDULING

The random selection of mutation strategies and the sequential selection of mutation positions in existing fuzzers are important factors that affect the quality of test cases and vulnerability detection. In this section, we will focus on the application of different machine learning algorithms in the problem of mutation strategy scheduling.

Genetic algorithms simulate the natural process of gene recombination and evolution. Based on the principles of biological evolution, they perform selection, crossover and mutation operations on test cases to enhance their ability to trigger exceptions. To address the issue of high inefficiency of test cases in fuzzing for industrial control protocol vulnerability discovery and to automate and streamline the fuzzing process, Zhang et al. designed a protocol fuzzer called GA-Fuzzer that combines genetic algorithms with fuzzing [27]. The model structure is shown in Figure 5. In the paper, a dynamic fitness function is constructed. Through monitoring the presence of danger point use cases within the test case population, different fitness functions are selected. Additionally, by introducing dynamic mutation and crossover probabilities, the diversity of test cases within the population can be adjusted based on the population's state, aiming to improve both the test hit rate and the test case coverage as much as possible.

Zhou proposed an improved mutation method that imitates mutation. It enhances the ability of poor individuals to bypass defense mechanisms by imitating the mutation strategy of good individuals, thereby improving the aggressiveness of test cases and the efficiency of genetic algorithms [28]. Zhang et al. also proposed an improved mutation strategy [29]. They set a threshold as a criterion, where individuals with fitness values higher than the threshold still undergo random mutation to maintain population diversity. Individuals with fitness values lower than the threshold randomly select an individual with a fitness value higher than the threshold, learn its mutation method, and mutate themselves accordingly, guiding the population towards high aggressiveness evolution.

DARWIN proposed a mutation scheduling optimization method based on evolutionary strategies. It systematically optimizes and updates the probability distribution of mutation methods using evolutionary strategies, selects an approximate optimal mutation strategy, and guides seeds to mutate towards high-quality directions [30]. AMSFuzz introduced an adaptive mutation scheduling framework [31]. It adaptively adjusts the probability distribution of mutation operators using a multi-armed bandit model to determine the capabilities of the mutation operators. It also utilizes a seed slicing mechanism to select the mutation positions and mutation area sizes for seeds, thereby improving the efficiency of fuzzing. SEAMFUZZ also proposed a fuzzing method for adaptive selection of variation strategies. By learning the individual characteristics of different seeds, different mutation strategies are applied to different seeds. SEAMFUZZ clusters seeds into clusters based on their grammar properties and uses Thompson sampling variants to learn the probability distribution of selecting different mutation strategies for each cluster, customizing effective mutation strategies for each seed cluster [32].

Reinforcement learning is the process of adjusting agent behavior during interaction with a system, aiming to maximize the received rewards based on executed actions and system state transitions. Böttinge et al. formalized fuzzing as reinforcement learning problems using Markov Decision Processes (MDP) [33]. The fuzzy agent learns a policy by

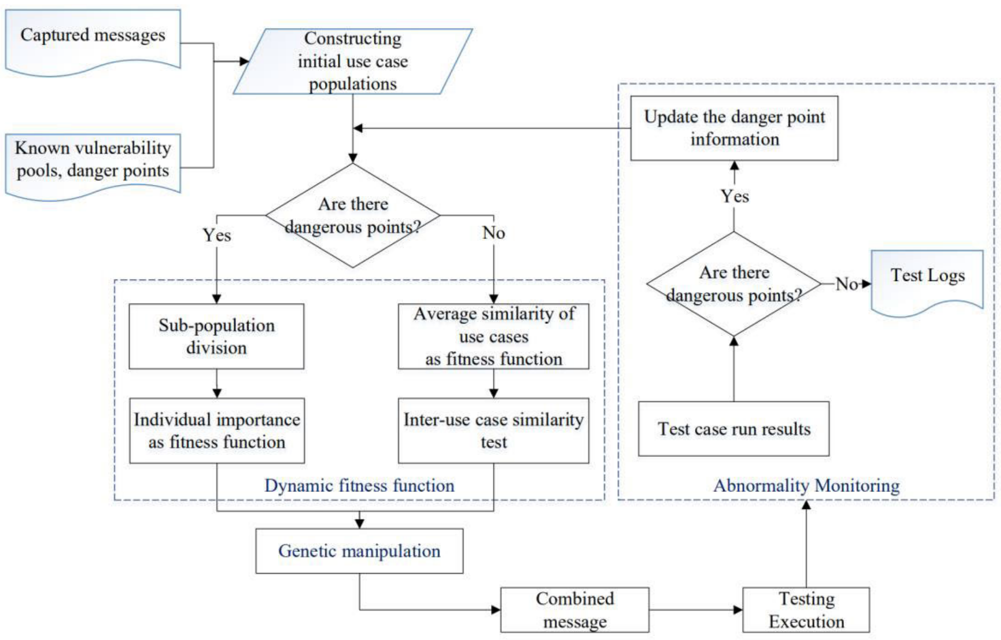
Captured messages
Known vulnerability pools, danger points
Constructing initial use case populations
Are there dangerous points?
Yes
No
Sub-population division
Individual importance as fitness function
Average similarity of use cases as fitness function
Inter-use case similarity test
Dynamic fitness function
Genetic manipulation
Combined message
Testing Execution
Update the danger point information
Yes
Are there dangerous points?
No
Test Logs
Test case run results
Abnormality Monitoring

FIGURE 5. Flow chart of test case generation algorithm based on genetic algorithm.

TABLE 2. Mutation strategy scheduling.

Literature	Model	Improvement Points	Test Target	Year
GA-Fuzzer [27]	genetic algorithm	More efficient dynamic fitness functions are constructed; Dynamic mutational operators and crossover operators are designed to optimize the test cases.	Modbus protocol	2021
[28]	genetic algorithm	Imitation Mutation mode.	Web Applications	2020
Zokfuzz [29]	genetic algorithm	A web application fuzzy script is proposed that enables automatic monitoring of web application state changes, while allowing more accurate determination of whether a test case is aggressive or not; A mutation method that enables populations to quickly evolve more aggressive test cases is proposed.	Web Applications	2022
DARWIN [30]	evolutionary strategy	The probability distribution of the mutation operator is optimized by using the evolution strategy.	real-world applications, LAVA-M and MAGMA	2023
AMSFuzz [31]	multi-armed bandit	Determining the ability of the mutation operator by adaptively adjusting the probability distribution of the mutation operator through the multi-armed bandit model. The seed slicing mechanism was used to select the mutation location and mutation region size of the seeds.	real-world programs and LAVA-M	2022
SEAMFUZZ [32]	Clustering; Thompson Sampling	Cluster seeds into clusters based on their syntactic and semantic properties. Thompson sampling is used to learn the probability distribution of selecting effective mutation methods for each seed cluster.	real-world programs	2023
[33]	MDP; Deep Q_learning	Deep Q_learning-based fuzzy algorithm for optimizing rewards.	PDF processing program	2018
[34]	MDP; Deep Q_learning	Selection of high-return actions given an input seed; Integration with hardware mechanisms during program execution.	Binary Programs	2021
[35]	DDPG	The DDPG reinforcement learning algorithm that combines the value function and the policy function is chosen to solve the process, from which the optimal action selection policy is learned.	LAVA-M dataset;	2021

observing the reward induced by mutations of a particular set of actions performed on the initial program input. The learned policy is used to generate new higher-rewarded inputs, which in turn improves the quality of test cases. DRLFuzzer uses

MDP and deep Q_learning algorithms to help the fuzzer automatically select mutation operations [34]. Furthermore, it realizes the combination with hardware mechanisms during program execution to effectively improve the path coverage and execution efficiency of fuzzing. A method based on the DDPG reinforcement learning algorithm was proposed by RLFUZZ to improve traditional fuzzing techniques [35]. After modeling traditional fuzzing as a Markov decision process, the DDPG reinforcement learning algorithm with an integrated value function and policy function is used to select an optimal action selection strategy for the process. This helps to reduce the blindness of sample mutation, enables mutation samples to obtain maximum code coverage reward, reduces the generation of invalid samples, and thus improves the efficiency of traditional fuzzing techniques.

3) TEST CASE GENERATION

Test cases can be generated through mutation operations on seeds or automatically generated based on the known specification format of test inputs. The content of test cases serves as the payload for attacking the target program, directly influencing the effectiveness of vulnerability detection. Therefore, constructing effective test cases with high code coverage can enhance the efficiency of fuzzers in vulnerability detection. In this section, we introduce three techniques for fuzzing based on test case generation: generation-based fuzzing, mutation-based fuzzing, and combination of generation and mutation-based fuzzing.

- **Generation-Based**

In the generation-based test case generation approach, machine learning algorithms are primarily used to learn the format specifications of the target program. By learning from well-structured corpus features, these algorithms generate a large number of high-quality test cases that adhere to the specifications.

For fuzzing against the PDF file format, Godefroid et al. proposed Learn&fuzz, which considers the use of deep learning algorithms to enhance the syntax-based fuzzing case generation process [36]. Learn&fuzz introduces a generation model based on Char-RNN to learn PDF objects and a SampleFuzz algorithm that can conduct fuzzy processing when sampling new objects, intelligently guiding the generation of well-formed PDF input files. While their experiment did not achieve better results, it was still a commendable effort. In 2021, Liu and Yang proposed an automatic test case generation model based on BLSTM and attention mechanism, along with an improved sampling algorithm based on Learn&fuzz [37]. BLSTM models were employed to extract and preserve information in the training samples considering both forward and backward factors. The attention mechanism highlights key positions of sample sequences and prevents information loss. The sampling algorithm's performance was improved by adding mutations that better predict character sequences. This paper proposed test case generation model can learn the intrinsic format of test cases and automate the generation of more well-formed test cases. Wang et al. also introduced a machine learning framework that can generate a large number of seed files [38]. They used the Transformer model to learn the internal formatting document syntax of PDF files and guide the generation of a new sequence of objects. These objects were then assembled to form a new PDF file with complete formatting for subsequent fuzzing. Experiments showed that for the mupdf software (version 1.4.0), this approach not only achieves faster coverage growth but also increases the upper limit of code coverage.

GANFuzz proposed an automated test case generation method that enables the generation of test cases without relying on protocol-specific format specifications [39]. Firstly, a real protocol message corpus is used as training data and is partitioned using three clustering strategies. Then, a test case generation model is built using Generative Adversarial Networks (GANs) [39]. By using the generated model based on real protocol messages, fake protocol messages with different degrees of similarity to real protocols can be generated. The SeqGan algorithm is introduced to update the generator's parameters using reinforcement learning techniques, addressing the issue of the inability to apply backpropagation during the training process of protocol messages. Finally, this approach generates a large number of effective test cases. When these test cases are applied to the Modbus TCP protocol, the experiment confirms their good vulnerability detection ability. However, overall, GAN-based fuzzing methods may somewhat reduce the efficiency of fuzzing.

Security of industrial control protocols is a crucial aspect in overall industrial safety. To overcome the limitations of traditional fuzzing heavily reliant on industrial protocol specifications, Wang et al. proposed a pointer-generated network (PGN)-based approach to handle the generation of fuzz testing data [40]. The aim is to intelligently learn the real sequences of industrial control protocol messages using a pointer-generated network and generate well-structured synthetic test cases similar to actual data frames without detailed protocol specifications. The architecture of this model combines three components: a seq2seq model, an attention mechanism, and a pointer network model. It also incorporates a coverage mechanism, as illustrated in Figure 6. Firstly, a hierarchical LSTM unit is employed as both the encoder and decoder of the seq2seq model to retain the temporal dimensionality information and feature vector dimensionality information. The encoder consists of bidirectional LSTM units that learn the character probability distribution within protocol messages. The decoder utilizes a unidirectional LSTM unit to predict the learned protocol sequence and generate test cases for fuzzing with semantically valid data fields and well-formed sequential grammar. Secondly, a coverage mechanism is introduced to address the issue of message repetition in the seq2seq model generation.

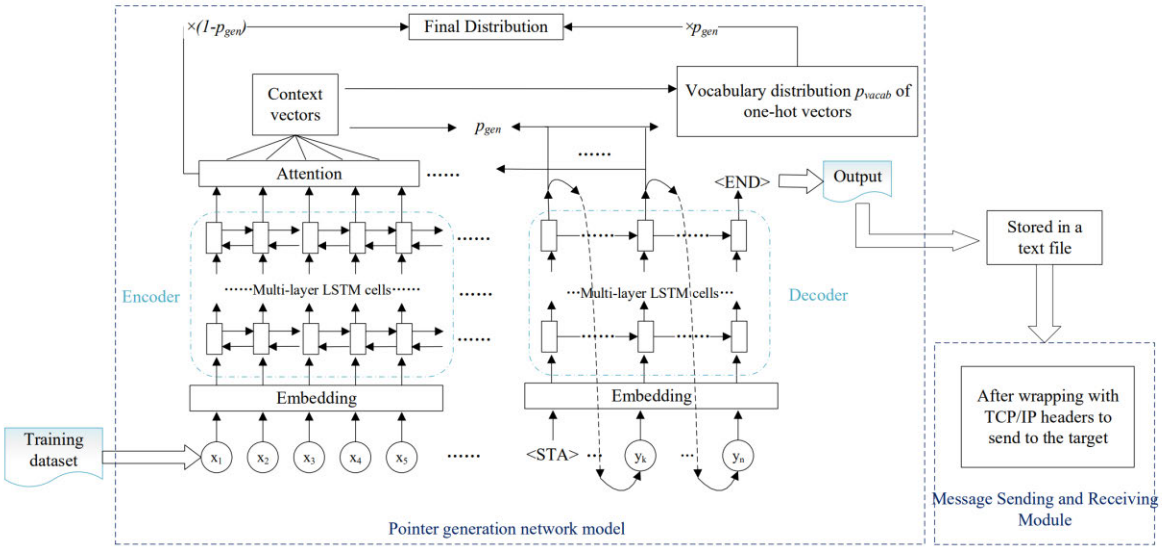

FIGURE 6. The specific generative model structure [40]. Where the generation probability of time step t $p_{gen} \in [0, 1]$.

Additionally, a general-purpose intelligent industrial control protocol fuzzing framework called PGNFuzz is proposed based on this method. Experimental results demonstrate that PGNFuzz outperforms GAN-based and LSTM-based seq2seq model fuzzers in industrial control protocol fuzzing scenarios.

Stateful protocols can pose testing difficulties for fuzzing, and in the case of unknown industrial protocols, seqfuzzer proposes a fuzzing method based on the seq2seq structure [41]. Real traffic in industrial networks is captured, pre-processed, and passed into the seq2seq model as a dataset. The LSTM model is used as both encoder and decoder of seq2seq to automatically learn temporal features of stateful protocols. By learning the syntax of the real protocol sequence, spurious protocol messages similar to real messages are generated as test cases for fuzzing. The experiments verify that seqfuzzer can generate test cases conforming to the EtherCAT protocol format with unknown protocol structure and detect various vulnerabilities. However, there are many protocols in industrial networks, and building a generalized protocol fuzzing method for industrial networks needs to face the challenge of various different private protocols. Future research targeting generalized fuzzing frameworks will also provide significant value to industrial safety.

In fuzzing for web applications vulnerabilities, A test case generation method for Web applications based on an improved LeakGAN algorithm has been proposed by Liu [42]. In the optimized LeakGAN algorithm model, the generator consists of two LSTM models acting as the manager module and the worker module, respectively. Additionally, batch normalization is introduced to process the input test cases, preventing extreme data distribution. The discriminator applies an attention mechanism to guide the generator in generating test cases. This method enables the generation of a large number of valid test cases that adhere to the syntactic structure.

Despite their widespread use, compilers and interpreters are still prone to defects that can cause abnormal behavior in programs. DeepFuzz [43] has proposed a test input generation method based on the seq2seq model to implement fuzzing of compiler test suites. The seq2seq model uses LSTM as the encoder and decoder and is trained to learn the language patterns of C programs. More syntactically correct C programs are generated as test inputs by employing insertion, replacement, and deletion strategies to fuzz the compiler. To address the problems of insufficient syntactic correctness and low generation efficiency in existing methods for generating test cases, a feedforward neural network-based compiler fuzzing case generation method is proposed in FAIR [44]. FAIR captures the widespread long-distance syntactic dependencies existing in the source code. Subtrees are extracted from the abstract syntax tree to form a sequence of code snippets. A self-attention-based feedforward neural network is used to capture the syntactic correlations between code snippets. By learning a series of context-aware feature representations in the input sequence, it predicts subsequent code sequences. For JavaScript engines, Montage utilizes LSTM to learn syntactic and semantic relationships between segments in a regression test set to guide the reconstruction of a given regression JavaScript test case and generate more effective test cases for use in JS engine fuzzing [45]. Similarly, COMFORT [46] has designed a test input generation model based on the GPT-2 model, which can generate more syntactically correct JS programs using the specification rules defined in the ECMAScript standard.

The application of deep learning models in the test case generation phase is widely studied. Seq2seq is a typical encoder-decoder model that can be used to generate more

TABLE 3. **Generation-based test case generation.**

Literature	Model	Improvement Points	Test Target	Year
Learn&fuzz [36]	char-rnn	The seq2seq model learns the syntax of PDF objects. Char-RNN trains statistical probability models to guide the generation of test inputs.	Microsoft in the Edeg browser PDF parser	2017
[37]	BLSTM; attention	BLSTM extracts and preserves more information from the training samples to improve the prediction accuracy. The Attention mechanism effectively prevents information loss by calculating the attention allocation probability and highlighting the key locations of the samples.	yaml-cp and mupdf	2021
[38]	Transformer	Learn the syntax of formatting documents inside PDF files and guide the generation of new PDF files.	mupdf PDF Reader	2021
GANFuzz [39]	Clustering; GAN; SeqGan	Clustering operations divide the training data; reinforcement learning techniques update the parameters of the generator; generative adversarial networks construct the generative model.	Industrial Network Protocols	2018
PGNFuzz [40]	seq2seq; attention; PGN	Learn the format of the original network packets and generate well-structured test cases similar to real data frames.	Industrial control protocols	2022
Seqfuzzer [41]	seq2seq; LSTM	Learn the structure of the protocol framework and handle automatic learning of the temporal characteristics of the state-based protocol.	Protocol	2019
[42]	LeakGAN; CNN; LSTM; attention	The generator model consists of two LSTM models as the manager module and the worker module, respectively, as well as the introduction of batch normalization for the input test cases. The discriminator applies an attention mechanism to guide the generator to generate test cases.	Web Application	2021
DeepFuzz [43]	seq2seq	Continuously generate syntactically correct C programs based on the learned syntax.	Compiler Test Suite	2019
FAIR [44]	FNN; Self-Attention	Learning a series of context-aware feature representations in the input sequence to predict subsequent code sequences.	Compiler Test Suite	2022
Montage [45]	LSTM	Transform an AST into a string of AST subtrees, train the LSTM, and guide the fuzzing generation process.	JavaScript engine	2020
COMFORT [46]	GPT-2	The GPT-2 model is fine-tuned and trained to generate valid test cases.	JavaScript engine	2021

and higher quality test cases by selecting LSTM, BLSTM, RNN, and their variants as encoders and decoders based on data characteristics. The Transformer model is an attention-based encoder-decoder model that dynamically selects all inputs through attention mechanisms, effectively preventing information loss and highlighting key positions in training samples. Therefore, attention mechanisms can be applied to various deep learning models. Although the basic RNN model can effectively handle long sequence data, it suffers from problems such as long-range dependencies and vanishing gradients. In contrast, the LSTM model can effectively address these issues by introducing cell states on top of the RNN and using gate structures to update and delete cell states. The LSTM model uses the output information from the previous part as input for the current training, thus achieving test case generation. Many studies currently focus on designing and implementing test case generation models based on the LSTM model, which have shown good performance in handling PDF files, various protocol information and compilers. BLSTM, considering both forward and backward factors on top of LSTM, extracts and retains more information from the training samples, thereby improving the prediction accuracy of the model. Deep learning models possess powerful learning and data processing capabilities, giving them an advantage in the generation of fuzz testing test cases. They can generate more high-quality test cases that conform to format specifications based on specification information such as syntactic format.

- **mutation-based**

Mutation-based fuzzing focuses on generating new test cases by modifying certain fields of valid inputs. The main optimization directions are selecting appropriate mutation positions and improving the fitness function. Typically, a fuzzer can guide the mutation process based on the evaluation results of test input performance to effectively generate inputs [47].

Rajpal et al. [48] proposed a technique that uses information from training data for mutation coverage to predict a heatmap of complete input files. This heatmap corresponds to the mutation probabilities for each file location that leads to new code coverage. It guides the generation of effective test cases, reducing time wasted on invalid test cases and improving the overall efficiency of fuzzing. DeltaFuzz [49] is a fuzzing technique based on historical version information. By analyzing the differences between historical versions and the target program, it locates the points of change. Then, it identifies the affected basic code blocks based on impact analysis. Finally, it calculates the fitness value of test cases based on execution traces and iteratively generates new test cases using a genetic algorithm to improve test case quality. Experimental results have shown that DeltaFuzz reaches the target faster compared to existing fuzz testing tools.

DYNFuzz [50] proposes a neural network-based directed grey-box fuzzing method. It uses an LSTM neural network to learn mutation patterns at different positions in previous input files and predicts the mutation gains at different positions in the current input file. This optimization helps in guiding future fuzz search. The entire DGF process consists of two stages: exploration and exploitation. The seed inputs in fuzzing are divided into two groups: coverage seeds for path exploration and directed seeds for exploitation. In the exploration stage, the fuzzer queries a trained neural network model before mutating the seed. The model returns a coverage heatmap for the corresponding complete input file, indicating probabilities of mutation-induced new code coverage for each position in the file. The mutation positions are sorted based on the probabilities from the coverage heatmap, giving priority to positions with a higher likelihood of mutation gains. In the exploitation stage, the distance between basic blocks and target points is calculated using the LLVM method, and each directed fuzzing seed is assigned a distance value for priority sorting and seed mutation optimization.

In protocol-oriented fuzzing, Xiang and Ma [51] aimed to avoid generating a large amount of redundant data. They utilized the returned error codes to indicate the code coverage of test cases and optimized the calculation method of individual fitness function based on two aspects: the similarity between individuals and the seed queue, as well as the error codes of seeds. This approach adjusted the evolution direction of genetic algorithms in a timely manner based on the fuzzing results, effectively improving the efficiency of fuzzing for the Modbus TCP protocol.

In the field of web application vulnerability detection, Qu et al. proposed a test case optimization method based on genetic algorithm to improve the effectiveness of fuzzing [52]. They analyzed different types of attack elements through traffic analysis systems and created a weighted web attack feature database, which was then passed to the genetic algorithm. The construction of the fitness function is based on the analysis of the response information of the web sites to calculate the actual fitness function value. By repeatedly iterating through the selection, crossover and mutation operations, the best test case is generated. Experimental results have shown that this method performs well in web vulnerability mining.

In addition to improving the fitness function and designing optimized mutation strategies during the test case mutation generation phase, some research has also focused on the problem of refining the initial sample set. Wang et al. utilized a heuristic genetic algorithm to optimize chromosome selection methods by eliminating redundancy caused by duplicate genes and selecting chromosomes that contain more genes and richer gene combinations [53]. This optimization allows for improved search conditions and enhanced efficiency of fuzzing without needing to change the working process of the genetic algorithm.

In summary, within the domain of mutation-based test case generation methods, most studies rely on genetic algorithms to generate diverse test cases. By improving the fitness function and optimizing search conditions, the quality of test cases is enhanced, effectively exploring the input space to identify potential vulnerabilities and errors. It should be noted, however, that the search process of genetic algorithms may require numerous iterations and computational resources, leading to potentially reduced efficiency when dealing with large-scale complex systems. Moreover, for target programs with complex software structures and varying variable types, further research is needed to address the handling of non-numeric variables and the construction of appropriate fitness functions. On another front, machine learning algorithms possess exceptional predictive capabilities that can address the issue of selecting suitable mutation positions. Through model training, these algorithms forecast mutation gains for different mutation positions, guiding the fuzzing tool to prioritize mutation at positions with higher gains. This enables faster traversal of the target program and minimizes the time required to trigger exceptions.

- **Combination of Generation and Mutation Based**

The combined approach of generation and mutation in fuzzing aims to leverage the strengths of both techniques, generating higher-quality and effective test cases.

When applied to network protocols, Wang et al. [54] proposed an adaptive fuzzing method based on transformers. Utilizing transformers, they learned semantic information of the Modbus TCP protocol and generated test cases. By comparing the semantic similarity between protocols, they guided the generated test cases to undergo byte-level mutations, reducing the similarity among test cases and enhancing the probability of triggering exceptions. This method combines both generation and mutation approaches, dynamically adjusting the mutation frequency of bytes that are prone to triggering vulnerabilities. It not only ensures compliance with the protocol's syntax format but also improves the ability to generate test cases that effectively trigger exceptions. RapidFuzz proposes a combination of an improved WGAN model based on the gradient penalty and a mutated gene detection algorithm for test case generation [55]. The WGAN-GP model is utilized to learn the numerical distribution of seed samples and generate numerous new samples. RapidFuzz proposes a mutated gene detection algorithm that sorts training set samples obtained from AFL based on the frequency

TABLE 4. **Mutation-based test case generation.**

Literature	Model	Improvement Points	Test Target	Year
[48]	LSTM; seq2seq;	Utilizes neural network models to predict good (and bad) locations in seeded input files to guide fuzzing mutations.	Readpng, readelf, mupdf and libxml	2017
DeltaFuzz [49]	genetic algorithm	Locate change points based on historical version information. A path-sensitive fitness function is proposed, and genetic algorithm iteratively generates test cases.	open-source software programs	2022
DYNFuzz [50]	LSTM	The model predicts the mutation gain at different locations of the input file and performs the corresponding fuzzy mutation based on the results.	Three programs for handling common file formats	2022
[51]	genetic algorithm	When calculating the fitness of an individual in the genetic algorithm, it is calculated from the similarity between the individual and the seed in the seed queue and the anomaly code of the seed.	Modbus TCP protocol	2020
[52]	genetic algorithm	Creating a Web Attack Signature Database with Weights; Using genetic algorithms to randomly pre-generate the test cases of the fuzzing test; Using the response of the Web service repeatedly iterate the weights of different attack signatures in the Web attack signature database.	Web Applications	2021
[53]	Peach-based heuristic genetic algorithm	Heuristic improvement of chromosome evolution and quality chromosome selection using 0-1 matrices and ensemble coverage.	jpg, PDF, AVI data	2022

TABLE 5. **Test case generation based on the combination of mutation and generation.**

Literature	Model	Improvement Points	Test Target	Year
[54]	Transformer	Learn protocol semantic information using Transformer network; compare semantic similarities between protocols and guide the generated test cases for byte mutation.	Modbus TCP protocol	2023
RapidFuzz [55]	WGAN	Combining with mutation gene detection algorithm and advanced learning model WGAN-GP.	9 different open-source programs with 6 types of highly-structured data	2021

of sensitive mutation sites. To combine samples generated by the GAN model with high-frequency mutation points with those generated by the original AFL, a semi-random method is employed. Through the rational combination of generation-based fuzzing with mutation-based fuzzing, RapidFuzz achieves significantly faster fuzzing speeds while obtaining higher coverage.

Generation-based test case generation techniques often require known specifications and relevant information. However, in most cases, fuzzing is a black-box testing technique where limited known information is available for training and optimizing the test case generation model. Additionally, existing mutation-based test case generation techniques exhibit strong randomness and often do not differentiate between seed files that have different vulnerabilities, resulting in wasted time generating ineffective test cases. As a result, research on the combined approach of generation and mutation in test case generation has gained significant attention, aiming to leverage the advantages of both approaches and improve the efficiency and vulnerability detection capability of fuzzing.

C. *INPUT SELECTION*

In the real world, due to the presence of a large number of invalid test cases in the generated input and various constraints protecting the target program, the efficiency of fuzzing processing is greatly affected. To improve the efficiency of fuzzing, machine learning techniques can be utilized for input selection by classifying a large number of test cases before testing, thus prioritizing and filtering out test cases that are expected to trigger new paths or specific types of vulnerabilities, as determined by testers.

Input selection involves the direct selection and elimination of test cases. Hu and Pan proposed a Quasi-Recurrent Neural Network (QRNN)-based fuzzing case filtering method for network protocols that combines the processing and prediction capabilities of the QRNN model for sequential data [56]. This method effectively learns the structural features of network protocols to automatically filter invalid test cases, thus improving the efficiency of network protocol fuzzing. Karamcheti et al. proposed a gray-box fuzzing method based on machine learning that directly models program behavior [57]. The learned forward prediction model maps program inputs to execution traces, and the entropy of the distribution of execution traces is used to assess the model's uncertainty about the input. A higher entropy indicates higher uncertainty, suggesting that the input may cover new code areas during execution. This method filters out deterministic test inputs, significantly reducing unnecessary executions and improving the efficiency of fuzzing. Zong et al. developed a directed gray-box fuzzer called FuzzGuard [58], which predicts the reachability of test inputs without executing the target program. By learning from previous execution inputs, it predicts whether a program can execute the target error code with newly generated inputs. If the prediction result is unreachable, the input will not be executed. This method, built upon the mature directed gray-box fuzzing tool AFLGo, improves overall efficiency by filtering out unreachable inputs, thereby saving actual execution time.

While research on input selection is limited, machine learning techniques have demonstrated their ability to enhance the quality of test inputs, reduce unnecessary resource waste, and outperform traditional fuzzing tools in terms of efficiency.

D. ANALYSIS OF RESULTS

The result analysis phase, which follows the completion of fuzzing, focuses on analyzing and processing the output information. In cases of abnormal output states, manual identification and analysis are typically required to determine the cause of the anomaly, a process heavily reliant on domain knowledge and the ability to perform vulnerability analysis and reproduction.

To improve the automation of fuzzing and reduce the influence of subjective experience on analysis results, machine learning techniques can be used for output classification, facilitating the identification of abnormalities and their underlying causes. Harsh et al. utilized four methods - supervised, unsupervised, unsupervised + supervised and semi-supervised - with various techniques such as decision trees, support vector machines, K-Means clustering and Nave Bayes, to experimentally address the root cause analysis problem [59]. Given the lack of labeled data, Harsh et al. proposed a semi-supervised method that is best suited for most real-world scenarios and evaluated the feasibility of the method on eclipse.

However, the application of machine learning techniques to the post-fuzzing result analysis phase requires further research due to the limited availability of labeled data sets suitable for training and the predictive nature of machine learning results, making it challenging to analyze and interpret the output of fuzzing.

IV. PERFORMANCE GAINS EVALUATION

Different machine learning methods have their own characteristics and can be applied to a variety of target scenarios. So far, machine learning techniques have been applied in the four main stages of fuzzing. In the preprocessing stage, deep learning and reinforcement learning techniques can be used to guide fuzzing in better utilizing program information. In the test case generation stage, evolutionary algorithms can be applied to guide seed mutation, and machine learning techniques can learn effective structural features, optimize seed selection, and generate a large number of test cases in unknown format specifications. Reinforcement learning can also be applied to select the approximate optimal mutation strategy to improve the efficiency of test cases. In the input selection stage, machine learning techniques can be applied to predict the effectiveness and accessibility of test inputs. In the result analysis stage, machine learning techniques can be utilized to automatically classify and identify output results, reducing manual cost. When applying machine learning to improve fuzzing, it is necessary to balance multiple factors and consider practical needs.

This section provides a further summary of existing research, analyzing the gain effects of different machine learning algorithms on fuzzing, mainly including coverage, vulnerability detection capability, efficiency and test case effectiveness.

A. COVERAGE

As one of the important indicators for evaluating fuzzing performance, coverage reflects the possibility of triggering crashes. Existing research on coverage indicators mainly includes statement coverage, branch coverage, number of triggered paths, edge coverage and basic block coverage. MPFuzz [19] improves coverage by a factor of 4 over conventional Fuzzer, while SmartFuSE [20] improved statement coverage by 2.8%-3.4%, branch coverage by 20.7%-26.9%, and increased the number of paths by 0.9-13.5 times. In LAVA-M, a total of 929 program defects were discovered. QYSM [21] explored over 20% of the code paths in libpng, increasing the code coverage by about 3%. NeuFuzz [25] can achieve more than 1000 new edge coverages in an hour, which is about 4 times better than AFL. All five types of errors in six real programs were detected at least 64 more bugs than other compared fuzzer in LAVA-M. DARWIN [30] averaged a 6.77% improvement in edge coverage in MOPT and a 1.73% improvement in edge coverage in AFL. The seeds generated by literature [38] covered up to 9914 paths, which is much higher than traditional methods. It can be seen that machine learning algorithms applied in fuzzing can effectively improve coverage and greatly enhance the ability to trigger crashes. Figure 7 shows the coverage ranges of different literature. The figure mainly displays the coverage interval implemented by different methods.

B. VULNERABILITY DETECTION CAPABILITY

The ability to detect vulnerabilities is the most intuitive reflection of fuzzer's performance. It mainly includes the number of triggered crashes, errors and CVEs. Existing literature mostly evaluates fuzzer performance by testing on public datasets and real-world applications. For example, QYSM [21] detected 13 unknown errors in eight real programs. Literature [26] discovered 10 CVEs in real programs; Literature [27], Literature [29], and Literature [42] conducted experimental verification in a web target environment and detected 4, 3 and 21 vulnerabilities respectively. AMSFuzz [31] discovered an average of 226.2 bugs in LAVA-M, detected 17 previously unknown bugs in real programs, 15 of which were assigned CVE IDs. Compared with the baseline, AMSFuzz triggered the most bugs in the same amount of time. SEAMFUZZ [32] generated 56.4%-57.1% more crash inputs, triggering a total of 606 crashes; discovered 99 unique bugs, including 27 bugs that other baselines did not detect.

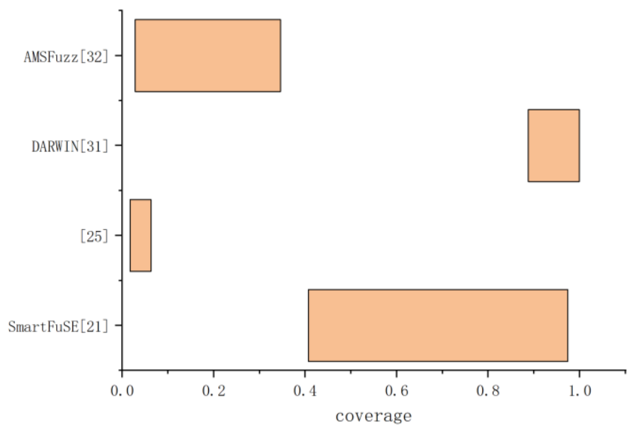

FIGURE 7. Comparison of coverage across different literature.

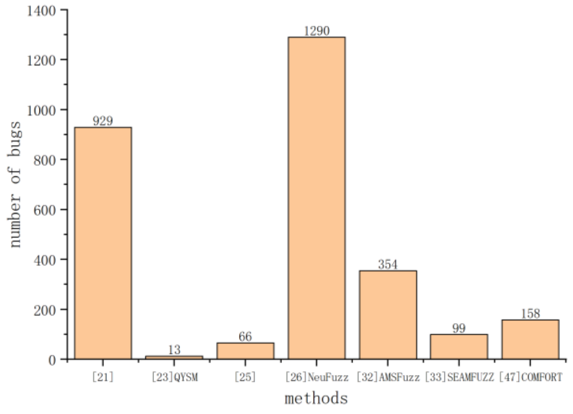

FIGURE 8. Comparison of the number of vulnerabilities detected by different methods.

Figure 8 shows a comparison of the number of vulnerabilities detected by different methods. Due to the different target datasets used in different literature, the number of bug detections also varies. Therefore, the bug quantity in this figure is selected as the maximum number of vulnerabilities detected in the literature.

C. EFFICIENCY

Efficiency mainly refers to the total number of covered paths or the number of test cases that trigger program crashes in the same amount of time. The more the quantity, the higher the efficiency. The average discovery time of vulnerabilities in [18] is 1.5 times higher than AFL. In an average runtime of 30 minutes, QSYM [21] generated hundreds of test cases, exceeding the number of test cases generated by other fuzzers by 10 times. DARWIN [30] is 48.26% faster than MOPT while, in the MAGMA benchmark test, after 5 hours of fuzzing, DARWIN was able to find 15 bugs (a total of 21). [39] discovered 5 bugs per 10,000 test cases. COMFORT [46] detected 158 unique vulnerabilities by automatically running on 250k self-generated test cases for 200 hours, of which 129 have been verified and 115 have been fixed by developers. In [51], only 2500 test cases successfully triggered two denial-of-service vulnerabilities. In [56], the total number of paths covered for BIND 9 within the same test time was 2360, an increase of approximately 85.1% in the overall path coverage.

D. TEST CASE EFFECTIVENESS

Due to various filtering and protection mechanisms in the target program, it is necessary to measure the effectiveness of test cases. Test case effectiveness primarily refers to the rate at which test cases can effectively input into the target program, representing their bypass capability. Different literature sets different specific indicators for this purpose. For example, in GANFuzz [39], the test rejection rate represents the percentage of test cases that are rejected. A lower rejection rate indicates better test case quality, and experimental results show that GANFuzz achieves rejection rates as low as 43%. PGNFuzz [40] introduces the test case identification rate, which primarily measures the percentage of test cases recognized by the test target. Experimental results demonstrate that PGNFuzz achieves an average test case identification rate improvement of 8%-13%. Similarly, SeqFuzzer [41] achieves a pass rate of 90.86%-99.99%. In [51], the acceptance rate has been improved by approximately 45%.

V. PROBLEMS AND PROSPECTS

Research on machine learning-based fuzzing techniques is currently a hot topic. Despite the numerous research results available, the complex and diverse architecture, syntax and input of target programs have resulted in a broad range of vulnerabilities with various causes and types. As such, efficiently and comprehensively detecting vulnerabilities using fuzzing remains a challenge, requiring continued efforts to address obstacles and limitations in this area.

A. MACHINE LEARNING MODELS CAN SLOW DOWN FUZZ TESTING

Fuzzing combined with machine learning techniques has emerged as a cybersecurity research hotspot. However, current efforts focus primarily on improving fuzzing coverage and achieving more accurate vulnerability detection by leveraging the image recognition and feature extraction capabilities of machine learning models to guide the generation of high-quality fuzzing cases in various domains. Nonetheless, given the time-consuming and computationally expensive nature of machine learning model training, fuzzing execution speed may be slower than traditional fuzzing methods, resulting in reduced overall efficiency.

To address this issue, some scholars have begun exploring strategies to expedite the generation of highly structured test cases. SmartSeed proposes a generic and efficient approach for test case generation that employs a WGAN model to learn valuable document features, which are then used to generate additional high-value test cases [60]. RapidFuzz introduces an improved Wassertein-Generative Adversarial

Network implementation utilizing a gradient penalty-based method to stabilize the GAN model training process and optimize probability distribution learning on the original dataset, addressing unstable model training and unexpected behavior issues caused by weight cropping introduced in WGAN [55].

As testing tasks become increasingly complex, parallel computing is also being leveraged to enhance fuzzing efficiency and effectiveness. Honggfuzz automatically supports multi-process and multi-thread execution of fuzzing techniques [61], ClusterFuzz deploys parallel fuzzing across multiple machines and cores [62], and literature [63] presents a parallelized task allocation method based on execution paths to reduce duplicate tests among fuzzing instances, fully utilize distributed computing resources, and improve parallelized fuzzing efficiency.

Execution speed is a critical metric for fuzzing. While improving machine learning models can reduce fuzzing time and increase efficiency, it is crucial to balance overall fuzzing efficiency with vulnerability detection capabilities through continuous research in this area.

B. EXPANDED APPLICATION AREAS

As research on machine learning techniques gains momentum, an increasing number of scholars are focusing solely on applying machine learning models in fuzzing. However, it is important to note that machine learning models are susceptible to adversarial examples, which may contain vulnerabilities that can cause serious security issues. Yi Qin proposed a hard-labeled black-box attack method based on fuzzing for machine learning models and developed two fuzzers, AdvFuzzer and LocalFuzzer, capable of generating numerous successful adversarial examples [64].

In recent years, there has been a gradual increase in research proposing fuzzing methods for machine learning frameworks. For example, FAME is a DL framework fuzzy system with an API mutation generation model and proposes the optimization of layer and weight mutations capable of detecting NaN errors and crash errors in deep learning frameworks [65]. Park et al. proposed a mixed constraint mutation (MCM) strategy for fuzzing deep learning systems, generating diverse variant results while preserving the original input semantics by combining various image transformation algorithms [66]. Muffin proposed a new model fuzzing approach to explore target libraries by developing metrics for measuring inconsistencies between different deep learning libraries and testing various models for differences, allowing the generation of different deep learning models [67]. DeepController uses feedback obtained during test execution to dynamically select seed and mutation strategies, proposing adaptive seed selection strategy-AS2, which uses feedback information from test execution to select seeds with high fault detection potential, and an adaptive mutation strategy selection method-AMS2, which analyzes the performance of mutation strategies on different seeds and selects the most suitable mutation strategy for different seeds [68]. Experimental validation on eight deep learning models shows that the method can generate more adversarial inputs and explore more internal states of the deep learning model with less time overhead.

Although existing fuzzing tools have been applied in various domains, such as file formats, network and industrial protocols, binary programs and security vulnerabilities of IoT devices, the application of machine learning models brings advantages such as improved detection accuracy to fuzzing. However, the models themselves may have security issues such as resource leakage, crashes, computational errors and anomalous behavior. An increasing number of scholars have worked on extending fuzzing techniques to machine learning frameworks and conducted experiments to verify their feasibility [65], [66], [67]. Hence, it is crucial to explore the nature of fuzzing technology, extend it to more application objects, and develop fuzzing techniques oriented toward multi-domain vulnerability detection.

C. DATASET STANDARDIZATION

Currently, there is a lack of standardized datasets for benchmarking in the field of fuzzing because different target programs have different characteristics and requirements. Researchers typically collect data through web crawlers, generate test cases using fuzzing, or utilize some publicly available datasets [69]. For example, in fuzzing for file formats and protocols, the commonly used dataset is LAVA-M [70], which was created by NIST and contains various types of files and protocols such as JPEG, MP3, PDF, HTTP, etc. The developers selected four programs, uniq, who, md5sum and base64, to create a corpus and injected some validated errors into each program. For fuzzing for web application vulnerabilities, spider technology is mainly used to collect and organize test input data, and the test data set is constructed manually. For fuzzing for binary programs, publicly available datasets include AFLSmart [71] and Fuzzingbook [72], which contain over 1,000 binary programs and their corresponding seed files that can be directly obtained from the official website. The quality of the dataset directly affects the training effect of the machine learning model and the performance of the vulnerability detection model. Therefore, it is meaningful to establish a standardized dataset for programs and vulnerability types in various fields.

D. EXPAND THE TYPES OF VULNERABILITY DETECTION

Three problems exist with current fuzzing techniques for detecting vulnerability types. Firstly, a large number of current fuzzers typically rely on program crashes as an indication of detected anomalies, but not all vulnerabilities result in program crashes, such as memory corruption, Trojans and viruses. Secondly, vulnerabilities have increasingly been triggered by multiple input points at different levels, and testing a single input point may not effectively monitor program anomalies, making it less effective in detecting vulnerabilities of the multi-point trigger type [73]. Thirdly, for some newly emerged vulnerability types, it can be challenging to build machine learning models due to the lack of relevant reference materials. Therefore, future research on how to detect more types of vulnerabilities will become one of the key research directions.

VI. CONCLUSION

The present study investigates the application of machine learning in the field of fuzzing, based on an extensive review of the relevant literature. Machine learning methods are most commonly applied in the test case generation phase, where genetic algorithms effectively generate diverse test cases to improve the coverage and effectiveness of fuzzing. Deep learning methods leverage their powerful pattern recognition and feature extraction capabilities to generate more targeted and high-quality test cases, further uncovering potential vulnerabilities and abnormal behaviors in the system. Reinforcement learning employs reward mechanisms to guide the generation of test cases that are conducive to exploring and discovering abnormal behaviors, thereby enhancing the efficiency and quality of test case generation. Moreover, machine learning has made many experiments and improvements in the preprocessing, input selection and result analysis and evaluation stages of fuzzing, effectively improving the efficiency and vulnerability detection capabilities of fuzzing. As an important approach in the field of fuzzing, machine learning provides new ideas and technical means for improving fuzzing techniques. Future research should further explore how to integrate different machine learning methods, harnessing their strengths to address the challenges faced in fuzzing, thus promoting the development and application of fuzzing technology.

Fuzz testing will continue to play a crucial role in future project vulnerability assessments. With the advancement of research in this field, we hope to witness the continuous application of machine learning to address bottlenecks in the fuzzing process. This article provides a detailed introduction to the development of machine learning-based fuzzing techniques and related research, aiming to serve as a valuable reference for researchers in this field.

ACKNOWLEDGMENT

The authors thank the anonymous referees for their helpful comments and suggestions on the initial version of this articles. They declare that they have no conflict of interest.

REFERENCES

[1] W. P. Wen, "Automated vulnerability mining and attack detection," *J. Inf. Secur. Res.*, vol. 8, no. 7, pp. 630–631, Jul. 2022.

[2] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990.

[3] R. Kaksonen, M. Laakso, and A. Takanen, *Communications and Multimedia Security Issues of the New Century*, vol. 64. Boston, MA, USA: Springer, 2001, pp. 173–183. [Online]. Available: https://link.springer.com/book/10.1007/978-0-387-35413-2

[4] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox fuzzing for security testing: SAGE has had a remarkable impact at Microsoft," *Queue*, vol. 10, no. 1, pp. 20–27, Jan. 2012, doi: 10.1145/2090147.2094081.

[5] J. Li, S. Li, G. Sun, T. Chen, and H. Yu, "SNPSFuzzer: A fast greybox fuzzer for stateful network protocols using snapshots," *IEEE Trans. Inf. Forensics Security*, vol. 17, pp. 2673–2687, 2022, doi: 10.1109/TIFS.2022.3192991.

[6] L. Liu, X. He, L. Liu, L. Qing, Y. Fang, and J. Liu, "Capturing the symptoms of malicious code in electronic documents by file's entropy signal combined with machine learning," *Appl. Soft Comput.*, vol. 82, Sep. 2019, Art. no. 105598, doi: 10.1016/j.asoc.2019.105598.

[7] A. Javaid, Q. Niyaz, W. Q. Sun, and W. Alam, "A deep learning approach for network intrusion detection system," in *Proc. 9th EAI Int. Conf. Bio-Inspired Inf. Commun. Technol.*, vol. 2016, pp. 21–26.

[8] Z. H. Ren, H. Zheng, J. Y. Zhang, W. J. Wang, T. Feng, and Y. Q. Zhang, "A review of fuzzing techniques," *J. Comput. Res. Develop.*, vol. 58, no. 5, pp. 944–963, May 2021, doi: 10.7544/issn1000-1239.2021.20201018.

[9] M. Zalewski. (2017). *American Fuzzy Lop*. Accessed: Jul. 4, 2023. [Online]. Available: https://lcamtuf.coredump.cx/afl/

[10] X. Zhou and Y. Hu, "Fuzzing method based on path risk fitness," *Commun. Technol.*, vol. 55, no. 4, pp. 500–505, Apr. 2022, doi: 10.3969/j.issn.1002-0802.2022.04.014.

[11] M. Cho, S. Kim, and T. Kwon, "Intriguer: Field-level constraint solving for hybrid fuzzing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA, Nov. 2019, pp. 515–530.

[12] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *Proc. IEEE Symp. Secur. Privacy*, San Jose, CA, USA, May 2015, pp. 725–741.

[13] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2018, pp. 711–725.

[14] H. R. Fang, F. Guo, and H. Y. Li, "TaintPoint: Fuzzing taint flow efficiently with live trace," *J. Softw.*, vol. 33, no. 6, pp. 1978–1995, Jan. 2022, doi: 10.13328/j.cnki.jos.006564.

[15] T. T. Gu, S. B. Lu, X. Li, X. H. Kuang, and G. Zhao, "Overview of parallel fuzzing," *Comput. Eng. Sci.*, vol. 44, no. 6, pp. 1046–1055, Jun. 2022, doi: 10.3969/j.issn.1007-130X.2022.06.012.

[16] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang, "PANGOLIN: Incremental hybrid fuzzing with polyhedral path abstraction," in *Proc. IEEE Symp. Secur. Privacy (SP)*, San Francisco, CA, USA, May 2020, pp. 1613–1627.

[17] L. Y. Liu, F. Li, Y. Y. Zou, J. H. Zhou, A. H. Piao, F. Liu, and W. Huo, "SiCsFuzzer: A sparse-instrumentation-based fuzzing platform for closed source software," *J. Cyber Secur.*, vol. 7, no. 4, pp. 55–70, Jul. 2022, doi: 10.19363/J.cnki.cn10-1380/tn.2022.07.05.

[18] T. Xiao, Z. H. Jiang, P. Tang, Z. Huang, J. Guo, and D. W. Qiu, "High-performance directional fuzzing scheme based on deep reinforcement learning," *Chin. J. Netw. Inf. Secur.*, vol. 9, no. 2, pp. 132–142, Apr. 2023.

[19] D. Luo, T. Li, L. Chen, H. Zou, and M. Shi, "Grammar-based fuzz testing for microprocessor RTL design," *Integration*, vol. 86, pp. 64–73, Sep. 2022, doi: 10.1016/j.vlsi.2022.05.001.

[20] F. J. Gao, Y. Wang, L. Y. Situ, and L. Z. Wang, "Deep learning-based hybrid fuzz testing," *J. Softw.*, vol. 32, no. 4, pp. 988–1005, Apr. 2021.

[21] X. Wang, J. Sun, Z. Chen, P. Zhang, J. Wang, and Y. Lin, "Towards optimal concolic testing," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng. (ICSE)*, Gothenburg, Sweden, May 2018, pp. 291–302.

[22] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 745–761.

[23] P.-H. Lin, Z. Hong, Y.-H. Li, and L.-F. Wu, "A priority based path searching method for improving hybrid fuzzing," *Comput. Secur.*, vol. 105, Jun. 2021, Art. no. 102242, doi: 10.1016/j.cose.2021.102242.

[24] W. Xiao, A. M. Zhou, and P. Jia, "Optimizing seed selection in fuzzing based on deep learning," *Mod. Comput.*, vol. 28, no. 8, pp. 30–35, Apr. 2022.

[25] Y. Wang, Z. Wu, Q. Wei, and Q. Wang, "NeuFuzz: Efficient fuzzing with deep neural network," *IEEE Access*, vol. 7, pp. 36340–36352, 2019, doi: 10.1109/ACCESS.2019.2903291.

[26] Y. Li, S. Ji, C. Lyu, Y. Chen, J. Chen, Q. Gu, C. Wu, and R. Beyah, "V-Fuzz: Vulnerability prediction-assisted evolutionary fuzzing for binary programs," *IEEE Trans. Cybern.*, vol. 52, no. 5, pp. 3745–3756, May 2022, doi: 10.1109/TCYB.2020.3013675.

[27] G. Y. Zhang, W. L. Shang, B. W. Zhang, C. Y. Chen, and R. Zhang, "Fuzzy test method for industrial control protocol combining genetic algorithm," *Appl. Res. Comput.*, vol. 38, no. 3, pp. 680–684, 2021, doi: 10.19734/j.issn.1001-3695.2020.03.0048.

[28] X. S. Zhou, "Research and implementation of web vulnerability detecting based on fuzzing test," M.S. thesis, Dept. Cyberspace Secur., Beijing Univ. Posts Telecommun., Beijing, China, 2020.

[29] H. Zhang, W. Dong, and L. Jiang, "Zokfuzz: Detection of web vulnerabilities via fuzzing," in *Proc. 2nd Int. Conf. Consum. Electron. Comput. Eng. (ICCECE)*, Guangzhou, China, Jan. 2022, pp. 281–287.

[30] P. Jauernig, D. Jakobovic, S. Picek, E. Stapf, and A.-R. Sadeghi, "DARWIN: Survival of the fittest fuzzing mutators," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, 2023, pp. 24–27.

[31] X. Zhao, H. Qu, J. Xu, S. Li, and G.-G. Wang, "AMSFuzz: An adaptive mutation schedule for fuzzing," *Expert Syst. Appl.*, vol. 208, Dec. 2022, Art. no. 118162, doi: 10.1016/j.eswa.2022.118162.

[32] M. Lee, S. Cha, and H. Oh, "Learning seed-adaptive mutation strategies for greybox fuzzing," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, May 2023, pp. 384–396.

[33] K. Böttinger, P. Godefroid, and R. Singh, "Deep reinforcement fuzzing," in *Proc. IEEE Secur. Privacy Workshops (SPW)*, San Francisco, CA, USA, May 2018, pp. 116–122.

[34] C. Chen, "Grey-box fuzzing with deep reinforcement learning and process trace back," in *Proc. 4th Int. Conf. Adv. Electron. Mater., Comput. Softw. Eng. (AEMCSE)*, Changsha, China, Mar. 2021, pp. 1167–1171.

[35] Z. Zhang, "Research on fuzz testing technology based on DDPG reinforcement learning algorithm," M.S. thesis, Dept. Cyberspace Secur., Beijing Univ. Posts Telecommun., Beijing, China, 2021.

[36] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Urbana, IL, USA, Oct. 2017, pp. 50–59.

[37] W. Q. Liu and W. C. Yang, "Research on efficient fuzzing technology based on deep learning," *Highlights Sciencepaper*, vol. 14, no. 2, pp. 160–167. Jun. 2021.

[38] M. Wang, D. G. Feng, L. Cheng, and Y. Zhang, "Optimization of fuzzing seed input based on machine learning," *Comput. Syst. Appl.*, vol. 30, no. 6, pp. 1–8, Jun. 2021.

[39] Z. Hu, J. Shi, Y. Huang, J. Xiong, and X. Bu, "GANFuzz: A GAN-based industrial network protocol fuzzing framework," in *Proc. 15th ACM Int. Conf. Comput. Frontiers*, May 2018, pp. 138–145.

[40] T. Y. Wang, S. H. Wu, Z. J. Li, H. G. Xin, X. Li, and Y. L. Chen, "PGNFuzz: Pointer generation network based fuzzing framework for industry control protocols," *Comput. Sci.*, vol. 49, no. 10, pp. 310–318, Jun. 2022.

[41] H. Zhao, Z. Li, H. Wei, J. Shi, and Y. Huang, "SeqFuzzer: An industrial protocol fuzzing framework from a deep learning perspective," in *Proc. 12th IEEE Conf. Softw. Test., Validation Verification (ICST)*, Xi'an, China, Apr. 2019, pp. 59–67, doi: 10.1109/ICST.2019.00016.

[42] Y. Y. Liu, "Research on fuzzy testing technology based on deep learning," M.S. thesis, Dept. Cyberspace Secur., Beijing Univ. Posts Telecommun., Beijing, China, 2021.

[43] X. Liu, X. T. Li, R. Prajapati, and D. H. Wu, "DeepFuzz: Automatic generation of syntax valid C programs for fuzz testing," in *Proc. 33rd AAAI Conf. Artif. Intell. 31st Innov. Appl. Artif. Intell. Conf. 9th AAAI Symp. Educ. Adv. Artif. Intell.*, 2019, pp. 1044–1051.

[44] H. R. Xu, Y. J. Wang, Z. J. Huang, P. D. Xie, and S. H. Fan, "Compiler fuzzing test case generation with feed-forward neural network," *J. Softw.*, vol. 33, no. 6, pp. 1996–2011, Jun. 2022.

[45] S. Lee, H. Han, S. K. Cha, and S. Son, "Montage: A neural network language model-guided JavaScript engine fuzzer," in *Proc. 29th USENIX Secur. Symp.*, Aug. 2020, pp. 2613–2630.

[46] G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, X. Sun, L. Bian, H. Wang, and Z. Wang, "Automated conformance testing for JavaScript engines via deep compiler fuzzing," in *Proc. 42nd ACM SIGPLAN Int. Conf. Program. Lang. Design Implement.*, New York, NY, USA, Jun. 2021, pp. 435–450.

[47] G. J. Saavedra, K. N. Rodhouse, D. M. Dunlavy, and P. W. Kegelmeyer, "A review of machine learning applications in fuzzing," 2019, *arXiv:1906.11133*.

[48] M. Rajpal, W. Blum, and R. Singh, "Not all bytes are equal: Neural byte sieve for fuzzing," 2017, *arXiv:1711.04596*.

[49] J.-M. Zhang, Z.-Q. Cui, X. Chen, H.-H. Wu, L.-W. Zheng, and J.-B. Liu, "DeltaFuzz: Historical version information guided fuzz testing," *J. Comput. Sci. Technol.*, vol. 37, no. 1, pp. 29–49, Feb. 2022.

[50] Z. J. Li, T. Y. Wang, Z. Q. Zhou, Y. Wang, and Y. L. Chen, "Directed greybox fuzzing technology based on LSTM and dynamic strategy," *Comput. Eng. Appl.*, vol. 58, no. 18, pp. 147–153, 2022.

[51] L. Xiang and R. F. Ma, "Research on fuzzy testing technology of Modbus TCP protocol based on genetic algorithm," *Ship Electron. Eng.*, vol. 40, no. 10, pp. 149–153, Oct. 2020.

[52] S. Qu, Z. Zhang, B. Ma, and Y. Shao, "Optimization method of web fuzzy test cases based on genetic algorithm," *J. Phys., Conf. Ser.*, vol. 2078, no. 1, Nov. 2021, Art. no. 012015.

[53] Z. H. Wang, H. F. Wang, and M. M. Cheng, "Fuzzing testing sample set optimization scheme based on heuristic genetic algorithm," *J. Beijing Univ. Aeronaut. Astronaut.*, vol. 48, no. 2, pp. 217–224, 2022, doi: 10.13700/j.bh.1001-5965.2020.0422.

[54] W. Wang, Z. Chen, Z. Zheng, and H. Wang, "An adaptive fuzzing method based on transformer and protocol similarity mutation," *Comput. Secur.*, vol. 129, Jun. 2023, Art. no. 103197.

[55] A. Ye, L. Wang, L. Zhao, J. Ke, W. Wang, and Q. Liu, "RapidFuzz: Accelerating fuzzing via generative adversarial networks," *Neurocomputing*, vol. 460, pp. 195–204, Oct. 2021.

[56] Z. H. Hu and Z. L. Pan, "Testcase filtering method based on QRNN for network protocol," *Comput. Sci.*, vol. 49, no. 5, pp. 318–324, 2022.

[57] S. Karamcheti, G. Mann, and D. Rosenberg, "Improving grey-box fuzzing by modeling program behavior," 2018, *arXiv:1811.08973*.

[58] P. Y. Zong, T. Lv, D. W. Wang, Z. Z. Deng, R. G. Liang, and K. Chen, "FuzzGuard: Filtering out unreachable inputs in directed greybox fuzzingthrough deep learning," in *Proc. 29th USENIX Secur. Symp.*, Aug. 2020, pp. 2255–2269.

[59] H. Lal and G. Pahwa, "Root cause analysis of software bugs using machine learning techniques," in *Proc. 7th Int. Conf. Cloud Comput., Data Sci. Eng.-Confluence*, Noida, India, Jan. 2017, pp. 105–111.

[60] C. Y. Lv, Y. W. Li, and S. L. Ji, "SmartSeed: Smart seed generation strategy for fuzzing testing," *J. Eng. Heilongjiang Univ.*, vol. 12, no. 3, pp. 90–108, Sep. 2021.

[61] SwieckiR. (2016). *Honggfuzz*. Accessed: Jul. 4, 2023. [Online]. Available: http://code.google.com/p/honggfuzz

[62] (2020). *ClusterFuzz*. Accessed: Jul. 4, 2023. [Online]. Available: https://google.github.io/clusterfuzz/

[63] R. Tang, "Research on efficient fuzzing technology based on deep learning and parallelization," M.S. thesis, Dept. Comput. Syst. Org., China Electron. Technol. Group Corp. Electron. Sci. Res. Inst., Beijing, China, 2022.

[64] Y. Qin and C. Yue, "Fuzzing-based hard-label black-box attacks against machine learning models," *Comput. Secur.*, vol. 117, Jun. 2022, Art. no. 102694.

[65] X. Shen, J. Zhang, X. Wang, H. Yu, and G. Sun, "Deep learning framework fuzzing based on model mutation," in *Proc. IEEE 6th Int. Conf. Data Sci. Cyberspace (DSC)*, Oct. 2021, pp. 375–380.

[66] L. H. Park, J. Kim, J. Park, and T. Kwon, "Mixed and constrained input mutation for effective fuzzing of deep learning systems," *Inf. Sci.*, vol. 614, pp. 497–517, Oct. 2022.

[67] J. Gu, X. Luo, Y. Zhou, and X. Wang, "Muffin: Testing deep learning libraries via neural architecture fuzzing," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*, May 2022, pp. 1418–1430.

[68] H. Dai, C.-A. Sun, and H. Liu, "DeepController: Feedback-directed fuzzing for deep learning systems," in *Proc. 34th Int. Conf. Softw. Eng. Knowl. Eng.*, Jul. 2022, pp. 531–536.

[69] X. Zhou and B. Wu, "Web application vulnerability fuzzing based on improved genetic algorithm," in *Proc. IEEE 4th Inf. Technol., Netw., Electron. Autom. Control Conf. (ITNEC)*, vol. 1, Jun. 2020, pp. 977–981.

[70] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "LAVA: Large-scale automated vulnerability addition," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 110–121.

[71] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart GreyBox fuzzing," *IEEE Trans. Softw. Eng.*, vol. 47, no. 9, pp. 1980–1997, Sep. 2021.

[72] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. Accessed: Oct. 25, 2023. [Online]. Available: https://www.fuzzingbook.org/

[73] S. Miao, J. Wang, C. Zhang, Z. Lin, J. Gong, X. Zhang, and J. Li, "Deep learning in fuzzing: A literature survey," in *Proc. IEEE 2nd Int. Conf. Electron. Technol., Commun. Inf. (ICETCI)*, Changchun, China, May 2022, pp. 220–223.

AO ZHANG received the bachelor's degree in software engineering from Hebei University. She is currently pursuing the master's degree with the Tianjin University of Science and Technology. Her research interests include fuzzing, vulnerability mining, and network security.

YIYING ZHANG received the B.E. degree from Northeast Normal University, in 1996, the M.Ec. degree from Northeastern University, China, in 2003, and the Ph.D. degree from Korea University, in 2010. He was a Postdoctoral Fellow with State Grid Information and Telecommunication Branch, from 2011 to 2013. He is currently a Professor with the Tianjin University of Science and Technology. His research interests include network security, wireless sensor networks, the Internet of Things, and smart grids.

YAO XU received the bachelor's degree in software engineering from Jilin University and the master's degree from the School of Artificial Intelligence, Tianjin University of Science and Technology. His research interests include network security and cloud environment security.

CONG WANG was born in Hunan, China, in 1988. She received the M.S. and Ph.D. degrees in computer science from Tianjin University, China, in 2012 and 2017, respectively. She is currently a Teacher with the Tianjin University of Science and Technology. Her research interests include network security and authentication protocol design and the Internet of Things.

SIWEI LI received the B.E. degree in Jilin Engineering Normal University in 2011. He is currently pursuing the Ph.D. degree with Tianjin University. He is working in State Grid Information and Telecommunication Co., Ltd. He has long been engaged in the research of smart grid load management and grid artificial intelligence related content.