International Journal for Housing Science and Its Applications
Publish August 6, 2025. Volume 46, Issue 3 Pages 8497-8511

https,//doi.org/10.70517/ijhsa463727

Research on Reinforcement Learning-Driven Software Supply Chain Vulnerability Detection and Repair Path Optimization Methods

Chenyu Liu[1], Jianheng Shi[1] and Shubin Yuan[1,*]
[1] CNOOC Information Technology Co., Ltd., Beijing Branch, Beijing, 102209, China
Corresponding authors: (e-mail: 18810525396@163.com).

Abstract Given the severe harm caused by vulnerabilities, vulnerability mining targeting the software supply chain has become a key focus for security researchers. As an effective technique for automated vulnerability mining in the software supply chain, this paper applies reinforcement learning algorithms to fuzz testing technology. It models the fuzz testing process using reinforcement learning and then employs the DDPG reinforcement learning algorithm to select strategies and solve the modeled problem. Additionally, this paper proposes an automated software vulnerability repair method based on large models, enhancing the model's vulnerability repair performance across three stages: input, model itself, and output. Experimental results show that the target site coverage speed of this paper's vulnerability detection method is 3.43 times and 1.45 times faster than the baseline method, and the discovery speed of real target vulnerabilities is 3.67 times and 1.84 times faster, demonstrating superior software supply chain vulnerability detection capabilities. Compared to other methods, the vulnerability repair method proposed in this paper achieves optimal repair effects for different vulnerability types and vulnerability program lengths, with recall rates improved by 39.38% to 142.49% in comparative experiments. Therefore, the vulnerability repair method proposed in this paper demonstrates superior vulnerability repair performance.

Index Terms reinforcement learning, software supply chain, vulnerability detection, fuzz testing, vulnerability repair

I. Introduction

Software vulnerabilities, also known as software weaknesses, refer to errors, defects, or flaws in computer software that may cause unexpected behavior or incorrect functionality during use. These issues can be exploited by malicious users or attackers to perform unauthorized operations, access sensitive information, disrupt system functionality, or cause other security issues [1]. Software vulnerabilities can occur in any type of software, including operating systems, applications, websites, and mobile applications. Common vulnerabilities may involve logical errors in code, insufficient input validation, memory management issues, and more [2]-[4]. According to a report published by the CVE Details website, the number of software vulnerabilities disclosed under the Common Vulnerabilities and Exposures (CVE) system was 20,153, 25,083, and 29,065 in 2021, 2022, and 2023, respectively. Over a five-year period, the number of disclosed software vulnerabilities nearly doubled [5]. Additionally, the situation regarding software vulnerabilities is becoming increasingly complex, not only in terms of the number of vulnerabilities increasing year by year, but also in terms of the complexity and diversity of their forms. These complex circumstances pose a threat to the security of computer systems and software, and there are significant potential risks and threats to their stable operation [6]-[8].

The presence of software vulnerabilities in safety-critical software is extremely dangerous [9]. Attackers may exploit these vulnerabilities to compromise system security or cause unforeseen operational outcomes during runtime, potentially leading to memory leaks or issues with security tools, resulting in significant economic losses, and even threatening human safety [10]-[12]. Therefore, software vulnerability detection technology plays a crucial role in developing high-reliability safety-critical software.

Software vulnerability detection techniques primarily encompass two categories: static analysis and dynamic analysis. Static analysis involves conducting a detailed analysis of software source code or binary code to identify potential vulnerabilities [13]. This method can rapidly scan large volumes of code to detect common vulnerability types, such as buffer overflows and format string vulnerabilities [14], [15]. However, static analysis is limited by code complexity and the accuracy of analysis tools, and may fail to detect some deeply hidden vulnerabilities [16]. Dynamic analysis uses various techniques and tools to perform dynamic testing on software, simulating different usage scenarios and attack scenarios. By running the software in actual operation, it observes its behavior and

state to identify potential vulnerabilities [17]. This includes techniques such as black-box testing, white-box testing, and fuzz testing [18]. Dynamic analysis can capture runtime errors and security vulnerabilities such as memory leaks and access permission restrictions. However, dynamic analysis requires significant time and resources and may be influenced by multiple factors such as the software runtime environment [19]-[21]. Traditional static and dynamic analysis methods have certain limitations in large-scale software project practices today, leading to poor practical application results.

Due to the limitations of traditional static and dynamic analysis, in recent years, with the development and advancement of artificial intelligence technology, AI has been increasingly applied to various complex and large-scale machine learning tasks [22]. Meanwhile, machine learning-based software vulnerability detection methods have begun to emerge and have become a mainstream research direction. An increasing number of researchers are exploring how to utilize AI technology to improve the efficiency and accuracy of vulnerability detection, thereby enhancing the reliability of security-critical software [23]-[26]. For example, [27] proposes a machine learning-based vulnerability detection method for predicting software vulnerabilities in Android applications, which performs well in terms of accuracy and recall compared to traditional static and dynamic detection models. Reference [28] reviews existing research on using deep learning for software vulnerability detection, focusing on how neural network technology can be used to understand code semantics and identifying challenges in this field. Reference [29] proposes a pattern-based software vulnerability detection method that combines traditional static analysis, machine learning, and graph mining to assist code analysts in identifying vulnerabilities in complex software systems, thereby overcoming the limitations of traditional automated analysis methods.

In code scenarios, defective code and non-defective code sometimes exhibit high similarity [30]. For example, the same single protection value or boundary conditions in other operations may differ only by subtle nuances [31]. Machine learning models often struggle to capture these subtle differences. To address this issue, [32] proposed the VulSniper model, which uses an attention mechanism to capture key code segments, focusing the defect detection task at the functional level. It achieved an F1 score of 80.6% on two defect types in the Sard dataset: buffer errors (CWE-119) and resource management (CWE-399). To address the high false positive rate in static analysis tools, [33] uses convolutional neural networks and clustering techniques to learn repair patterns from recurring violations, then validates their applicability to actual errors based on their acceptance rate. Reference [34] leverages transferable knowledge from multiple existing data sources to enhance machine learning-based vulnerability detection capabilities. By utilizing cross-domain data sources and deep learning techniques to address the cold start problem, its performance surpasses that of traditional models. Reference [35] proposes a model that explicitly encodes different levels of control flow, data dependencies, and natural code sequences into a joint graph of heterogeneous edges. This integrated representation helps capture as wide a range of vulnerability types and patterns as possible and learns better node representations through graph neural networks. The SySeVR framework proposed in [36] employs a deep learning technique to systematically identify software vulnerabilities in C/C++ programs, demonstrating its effectiveness by detecting 15 unreported vulnerabilities across four software products. While these studies exhibit strong performance in feature representation, they incur high training costs and involve complex processing algorithms, resulting in slower vulnerability detection models based on traditional machine learning approaches.

Reinforcement learning (RL) is a machine learning method based on the Markov decision process, enabling an agent to learn optimal strategies through trial and error interactions with the environment [37]. Reference [38] developed a deep reinforcement learning algorithm for automated vulnerability mining, reducing operational costs and time, and achieving high accuracy in establishing reverse shells in two application scenarios. Reference [39] proposed a coverage-guided reinforcement learning-based fuzz testing model that enhances the effectiveness of dynamic analysis (gray-box fuzz testing), aiming to significantly improve testing effectiveness for actual programs by maximizing code coverage. Reference [40] proposes a reinforcement learning-based prototype verification method that generates secure and diverse software configurations to address configuration errors in static analysis. By dynamically adjusting settings, this method maximizes the reduction of software vulnerability risks.

The study adopts fuzz testing technology as a starting point to investigate methods for identifying vulnerabilities in supply chain software. Addressing the issue that traditional fuzz testing techniques exhibit significant randomness when mutating samples, severely impacting their efficiency, the study abstracts three key elements—states, actions, and rewards—from the fuzz testing process and models the fuzz testing problem using reinforcement learning. Subsequently, the DDPG reinforcement learning algorithm is employed to guide the selection of mutation strategies during fuzz testing, aiming to solve the fuzz testing modeling problem. This study proposes a method to improve traditional fuzz testing techniques using the DDPG reinforcement learning algorithm. Additionally, we propose a large-model-based automatic software vulnerability repair method, designing prompt

engineering and model fine-tuning techniques to help the model better understand vulnerability repair tasks and generate higher-quality repair programs. We also construct a reordering algorithm focused on the security of generated code to distinguish repair programs ranked in the top k for security factors. Experiments are conducted on the LAVA-M dataset to compare the coverage speed of the proposed method and other methods on target sites, followed by CVE vulnerability testing to obtain the vulnerability reproduction speed of the proposed method, and to explore the software supply chain vulnerability detection and repair capabilities of the proposed method. Similarly, comparative experiments are conducted on the Big-Vul and CVEFixes datasets to evaluate the proposed vulnerability repair method, and the repair effectiveness of each method for different defect types and vulnerability program lengths is discussed.

II. Reinforcement learning-based software supply chain vulnerability mining technology

II. A.Fuzzy Testing Technology

Fuzz testing is a common testing technique used to detect security vulnerabilities in computer software or systems. The most widely used fuzz testing framework is AFL, which employs evolutionary algorithms, Fork Server, and coverage-guided techniques to mitigate the drawbacks of traditional fuzz testing, such as high randomness and lack of directionality. However, it still has many areas that require improvement. Fuzz testing is typically classified into three categories based on different dimensions. Depending on the degree of reliance on internal characteristics or runtime information of the target program, it can be divided into black-box fuzz testing, white-box fuzz testing, and gray-box fuzz testing. Based on sample generation algorithms, it can be categorized into mutation-based fuzz testing and generation-based fuzz testing. Depending on path exploration methods, it can be classified into directed fuzz testing and undirected fuzz testing. The complete fuzz testing process typically includes six stages: selecting the target program, pre-testing preparation, generating test cases, executing the target program, checking for anomalies, and classifying vulnerabilities. However, traditional fuzz testing techniques have certain limitations. Therefore, this paper introduces reinforcement learning technology into the fuzz testing process to study software supply chain vulnerability detection techniques based on reinforcement learning.

II. B.Problem Modeling

Reinforcement learning problems primarily consist of three elements: states, actions, and rewards. The interaction and influence of these three elements form the foundation of reinforcement learning problems. To use reinforcement learning algorithms to assist in optimizing traditional fuzz testing processes, it is first necessary to abstractly model the traditional fuzz testing process as a problem solvable by reinforcement learning algorithms. This involves extracting the three elements—states, actions, and rewards—from the traditional fuzz testing process and ensuring they satisfy the Markov property. This section primarily introduces the abstraction and selection process of the three elements—states, actions, and rewards—in the modeling of fuzz testing problems based on reinforcement learning.

II. B. 1) Environmental conditions

In fuzz testing, the key factor affecting testing efficiency is whether mutations can generate high-quality mutation samples. Based on this, the input samples are treated as environmental states in this problem modeling.

This paper uses a byte array method to represent the corresponding state s of the input sample data D, where each element e_i of s has a value range of [0, 255], and the maximum length of the array is $L_{\max}$, depending on the specific problem environment. If the length L_D of the sample data D is less than $L_{\max}$, it is padded with zeros to reach the maximum length. Then:

$$e_i = \begin{cases} d_i & i \in [0, L_D] \\ 0 & i \in (L_D, L_{max}] \end{cases} \tag{1}$$

According to FuzzerGym, in order to maximize the probability of finding new paths in the code, the initial seed sample set should be empty, i.e., $s_0 = \varnothing$. At the same time, in order to better utilize the existing data mutation history experience, the system chooses to maintain an effective sample queue Q_s and a set of all path information executed by existing samples P. If the sample state at time step t is s_t, the mutated sample state is s_t', and a new execution path p_t is generated during execution, it is appended to the queue Q_s and the set P is updated, and s_t' as the state input for the next time step. Otherwise, a sample data is randomly selected from the valid sample queue Q_s as the state input for the next time step, as shown in Equation (2):

$$s_{t+1} = \begin{cases} Random\_choose(Q_s) & p_t \in P \\ s_t^{'} & p_t \notin P \end{cases} \tag{2}$$

II. B. 2) Variation of movements

The core of the fuzz testing process lies in mutating the sample data to obtain new samples that can trigger abnormal states in the target program.

The reinforcement learning model selects the mutation action a_t from the mutation action space according to the strategy π based on the current state s, as shown in Equation (3):

$$a_t = \pi(s_t) \tag{3}$$

The system then performs mutation processing on the current input data state s_t based on the selected action, as shown in Equation (4), to fully explore the environment state space and mutation action space, and obtain the corresponding state $s_t^{'}$ of the mutated sample with higher path coverage:

$$s_t^{'} = Mutate(s_t, a_t) \tag{4}$$

II. B. 3) Feedback Rewards

This paper selects edge coverage as the feedback reward calculation method in problem modeling. According to the design of AFL, after the target program inputs the mutated sample state $s_t^{'}$ and executes, it records the execution path information of this execution to the shared memory, denoted as M_t, as shown in Equation (5):

$$M_t = Execute(s_t^{'}) \tag{5}$$

Each element m in the record represents the number of times the jump edge from a basic block b_i to another basic block b_j has been executed.

If $m > 0$, it means that the jump edge has been executed at least once. The subset of records that satisfy this condition is denoted as $M_t^{'}$, as shown in Equation (6):

$$M_t^{'} = \{m_i \mid m_i > 0, i \geq 0, m_i \in M_t\} \tag{6}$$

The feedback reward R_t is calculated as shown in Equation (7), which is the ratio of the jump edges traversed by the current sample during execution to all jump edges in the target program:

$$R_t = \frac{size(M_t^{'})}{size(M_t)} \tag{7}$$

In summary, the fuzz testing process is abstracted into a problem that can be solved by reinforcement learning. The reinforcement learning model intelligently selects actions based on the state to maximize cumulative rewards. In the context of reinforcement learning-based fuzzy testing modeling, this means the model can intelligently select appropriate mutation strategies based on the current input samples, ensuring that the mutated new input samples achieve the highest edge coverage when executed in the target program. This reduces the randomness and blindness of mutations in traditional fuzzy testing processes, improves the quality of mutation-generated samples, and thereby enhances fuzzy testing efficiency.

II. C.Strategy selection based on DDPG

In the above reinforcement learning-based fuzz testing problem modeling, the modeling problem has a huge environment state space and a variable action space. In order to efficiently explore this space and improve the problem-solving efficiency, this paper will use the DDPG algorithm to solve the problem.

II. C. 1) DDPG Algorithm

The DDPG algorithm, also known as Deep Deterministic Policy Gradient (DDPG), is an offshoot of the Actor-Critic (AC) algorithm and is a policy-based reinforcement learning algorithm, meaning that the policy network used to generate actions and the value network used to evaluate actions employ different strategies. The Actor-Critic algorithm primarily consists of two components: the Actor and the Critic. The Actor uses a policy function to generate actions and interact with the environment, while the Critic uses a value function to calculate the value of actions and evaluate the Actor's performance, thereby guiding the Actor's action generation in the next phase. In

summary, the AC algorithm typically includes a policy network and a value network, denoted as $Q(S_t, A_t; \omega)$ and $\pi(S_t; \theta)$, respectively, where ω and θ represent the parameters of the two neural networks.

The value network Q is updated using the temporal difference method. Since the goal is to minimize the TD error, the loss function is defined as:

$$L(\omega) = \frac{1}{2}[R_{t+1} + \gamma Q(S_{t+1}, \pi(S_{t+1})) - Q(S_t, A_t; \omega)]^2 \tag{8}$$

The gradient of this loss function is calculated as follows:

$$\nabla_\omega L(\omega) = [R_{t+1} + \gamma Q(S_{t+1}, \pi(S_{t+1})) - Q(S_t, A_t; \omega)] \cdot \nabla_\omega Q(S_t, A_t; \omega) \tag{9}$$

Then, update the parameters of the value network through gradient descent:

$$\omega \leftarrow \omega - \alpha \nabla_\omega L(\omega) \tag{10}$$

For learning the policy function π, the policy gradient method is used to update the parameters θ. Since the formula for calculating the policy gradient is difficult to solve directly, we use Monte Carlo approximation. Each time a state s is observed from the environment, it is treated as an observation value of the random variable S. A random action is sampled from the current policy network, i.e., $a \sim \pi(\cdot \mid s; \theta)$, to calculate the gradient:

$$g(s, a; \theta) = Q_\pi(s, a) \cdot \nabla_\theta \log \pi(a \mid s; \theta) \tag{11}$$

As an approximation of the policy gradient $\nabla_{\theta J}(\theta)$, it is clear that $g(s, a; \theta)$ is an unbiased estimate of $\nabla_{\theta J}(\theta)$. Substituting the value network $Q(s, a; \omega)$ into the above equation yields the formula for the approximate policy gradient:

$$\hat{g}(s, a; \theta) = Q(s, a; \omega) \cdot \nabla_\theta \log \pi(a \mid s; \theta) \tag{12}$$

Finally, since the approximate policy gradient is known, the parameters of the policy function can be updated directly through gradient ascent, i.e.:

$$\theta \leftarrow \theta + \beta \cdot \hat{g}(s, a; \theta) \tag{13}$$

The DDPG algorithm introduces several key technologies based on the AC algorithm:

(1) Target network: In addition to the policy network and value network, a policy target network and value target network are also introduced. The target network operates as an independent network with the same structure as the main network, but its parameters are not entirely identical. The update of its parameters follows specific replication rules: every C steps (where C is a constant), the target network is synchronized with the main network either through direct copying (hard update) of the main network's parameters or through exponential decay averaging (soft update).

(2) Experience replay: Also known as experience caching, this involves storing the interaction records between the agent and the environment in a finite array of size M, which is also called the experience replay pool. These interaction records can be repeatedly used to train the agent, thereby saving on the number of training samples. Specifically, the trajectory quadruple (s_t, a_t, r_t, s_{t+1}) obtained from the agent's interaction with the environment is stored in the experience replay pool, and only the latest M data points are retained. Once the pool is full, the oldest data points are immediately deleted. Experience replay breaks the sequence correlation, reusing collected experiences to achieve the same performance with fewer samples.

Traditional experience replay uses uniform random sampling; however, the importance of the obtained sample data is not the same, and important samples need to be sampled multiple times. Therefore, scholars have proposed priority experience replay, which assigns a weight to each quadruple and then performs non-uniform random sampling. Typically, non-uniform sampling uses the absolute value of the TD error as the weight, i.e.:

$$|\delta_j| = Q(s_j, a_j; \omega) - [r_t + \gamma \cdot \max_{a \in A} Q(s_{j+1}, a; \omega)] \tag{14}$$

If the absolute value of the TD error $|\delta_j|$ is large, it indicates that the TD target is significantly different from the Q function, and the reinforcement learning algorithm's assessment of the true value of (s_j, a_j) is inaccurate. Therefore, a higher weight should be assigned to (s_j, a_j, r_j, s_{j+1})

Since prioritized experience replay uses non-uniform sampling, different samples have different sampling probabilities. Assuming that the probability of sampling the quadruple (s_j, a_j, r_j, s_{j+1}) is p_j, there are generally two ways to define it, one of which is:

$$p_j \propto |\delta_j| + \grave{o} \tag{15}$$

Here, $\grave{o}$ is a very small number used to prevent the sampling probability from approaching 0, ensuring that all samples are drawn with a non-zero probability. Another way to define it is to first sort $|\delta_j|$ in descending order, then calculate the sampling probability:

$$p_j \propto \frac{1}{rank(j)} \tag{16}$$

The $rank(j)$ is the ordinal number of $|\delta_j|$. The basic principle behind these two definitions is the same, namely that samples with larger $|\delta_j|$ have a higher probability of being selected.

II. C. 2) Variation Strategy Selection

The DDPG algorithm can analyze and process the huge state space and action space obtained from fuzzy testing modeling, thereby achieving efficient solution of the modeling problem.

In the DDPG algorithm for solving problems obtained from fuzzy testing modeling based on reinforcement learning, the algorithm model selects a specific data mutation strategy a based on the current input sample data s using the strategy π learned through training. Then, based on the data mutation strategy a, the system modifies and mutates the input sample data s and inputs it into the preprocessed target to be tested. waits for its execution to complete and obtains the coverage reward data r and new environment state data s' returned by the environment, thereby completing a complete interaction process. At the same time, the system updates the model parameters and selects the strategy π. Then, the system continues the above execution steps by assigning $s = s'$ until the predefined training conditions are met, completing the model training process. In this process, the DDPG algorithm does not simply select the currently known optimal mutation strategy action when choosing a specific mutation strategy. Instead, it introduces OU noise to explore the action space A, thereby balancing exploration and exploitation across the entire action space and avoiding getting stuck in a local optimum while ignoring other potentially high-value mutation strategies π^*.

In summary, this paper uses the DDPG algorithm to intelligently select data mutation strategies, thereby generating more high-quality samples and improving the efficiency of fuzz testing.

III. Software supply chain vulnerability remediation methods based on LLM

The emergence of large language models (LLMs) has opened up new avenues for automated vulnerability repair. However, since the code corpora used in the pre-training process of LLMs do not have security labels, the repair programs are generated using a Top-K sorting algorithm based on probability, without considering code security factors. To address these issues, this paper proposes a large model-based automated repair method for software supply chain vulnerabilities.

III. A. Data Format Definition

The dataset constructed in this paper is derived from two large-scale vulnerability repair datasets: Big-Vul and CVEFixes. In the data format, “[INST]” and “[/INST]” denote the start and end of the model input. In the input prompt, “instruction” refers to the instruction, which specifies the model's role and the task to be completed. “cwe_id” denotes the CWE ID of the vulnerability code, corresponding to the vulnerability type, while “cwe_description” provides specific details about the vulnerability type, aiding the model in understanding the underlying principles of the vulnerability. “example” represents a repair case, illustrating the process of fixing a vulnerability from identification to logical analysis and ultimately to repair. Different vulnerability types are accompanied by distinct repair cases, and it also demonstrates the format of the output data. “source_code” refers to the vulnerability code with marked vulnerability locations. For the vulnerability code format input to the model, the vulnerability program with marked vulnerability locations is used as input. To distinguish between repair locations, two special tokens, <BUGS> and <BUGE>, are used to mark each segment of the vulnerability code that needs to be modified in the vulnerability program. For the data representation of the fixed program, two special

tokens, <FIXS> and <FIXE>, are used to mark the modified code segments. Each <BUGS> and <BUGE> marked vulnerability code segment has a corresponding <FIXS> and <FIXE> marked fixed code segment.

III. B. Model fine-tuning

The model architecture during the fine-tuning phase follows the Llama model. To endow the model with rich pre-trained knowledge, the weight parameters of the Deepseek model are used to initialize the Llama model as the base model for fine-tuning. Fine-tuning is performed using the LoRA method.

The Llama model only adopts the Decoder module structure of the Transformer architecture, i.e., the decoder. The decoder is composed of multiple layers of Llama Decoder Layers, each of which includes data normalization, multi-head self-attention, and a feedforward neural network.

Data normalization: For input vectors x_i and $weight_i$ being learnable parameters, the RMSNorm normalization function is calculated as follows:

$$\frac{x_i}{\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^{\,2} + eps}} * weight_i \tag{17}$$

Multi-head self-attention: The core component in each layer is the LlamaAttention layer. The attention calculation formula is:

$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{18}$$

Feedforward neural network: A feedforward neural network is a multi-layer perceptron (MLP) that primarily uses nonlinear activation functions and linear projections. The MLP module consists of multiple fully connected layers. In LLaMA, the fully connected layers use the FFN formula with the SwiGLU activation function:

$$FFN_{SwiGLU}(x,W,V,W_2) = (Swish_1(xW) \otimes xV)W_2 \tag{19}$$

Among them, W, V, and W2 are the parameters of the linear layer, and the Swish activation function is expressed as:

$$Swish_\beta(x) = x\sigma(\beta x) \tag{20}$$

Among them, $\sigma(x)$ is the Sigmoid function, and β is a learnable parameter. When $\beta = 1$, the Swish function here can be replaced by the SiLU function, resulting in the formula:

$$SiLU(x) = x\sigma(x) \tag{21}$$

Thus, Swishl(xW) was replaced by SiLU(xW), and the final FFN formula was obtained as follows:

$$FFN_{SwiGLU}(x,W,V,W_2) = (SiLU(xW) \otimes xV)W_2 \tag{22}$$

III. C. Reordering Algorithm

III. C. 1) Generating a repair program

First, the LLM uses Beam Search to generate candidate fixes. To avoid missing high-probability words that are hidden, Beam Search uses a best-first search strategy at each time step to select the top n sequences with the highest probability, and continues to generate using these n sequences at the next time step. n is typically referred to as the beam width or beam size. When generating repair programs, the beam size is set to N, thereby generating the top N candidate repair programs with the highest probabilities.

III. C. 2) Consistency matching

Compare the N candidate fixes with the standard answer for consistency. If any of the candidate fixes match the standard answer exactly, then that candidate fix is considered a valid fix.

For those candidate fixes that do not have samples matching the standard answer, the original vulnerability code is repaired and replaced at the corresponding location to obtain the candidate complete fix, which is then subjected to security sorting. This process consists of two steps: functional correctness screening and code security sorting.

III. C. 3) Functional Correctness Screening

For each candidate complete repair, calculate its CodeBLEU evaluation function score and perform functional correctness screening to obtain the m candidate complete repairs that pass the screening. If no candidate complete repairs pass the functional correctness screening, the algorithm ends. The CodeBLEU calculation formula is as follows:

$$Score_{CodeBLEU} = \alpha \cdot B + \beta \cdot B_{weight} + \gamma \cdot Match_{ast} + \delta \cdot Match_{df} \tag{23}$$

Among them, B represents N-GramMatch(BLEU), B_{weight} represents Weighted N-Gram Match(BLEU-weighted), $Match_{ast}$ represents abstract syntax tree matching score, and $Match_{df}$ represents data flow matching score.

III. C. 4) Safety Ranking

For the m candidate complete fixes that pass the functional correctness screening, calculate their LineVul evaluation function scores as their security coefficients. Perform a quick sort based on the security coefficients to obtain the top k candidate complete fixes with the highest security coefficients as the final fix program, ensuring the security of the fix program. In this paper, LineVul is used as the evaluation metric for code security. The final score formula is:

$$Score_{LineVul} = p \cdot Pro_P + q \cdot Pro_N \tag{24}$$

Where Prop denotes the probability that the candidate repair program is output as a positive sample by LineVul, and Pro_N denotes the probability that the candidate repair program is output as a negative sample by LineVul.

Quick sorting process for m candidate repair programs:

(1) If m is less than or equal to k, the sorting ends.

(2) Select the benchmark element: Select the score value of the LineVul evaluation function of the first candidate repair program among the m candidate repair programs as the benchmark element.

(3) Partitioning operation: Use two pointers, one scanning from left to right to find numbers less than the benchmark element, and the other scanning from right to left to find numbers greater than the benchmark element. Swap the found numbers with the benchmark element until the two pointers meet.

(4) Recursive sorting: Perform quick sort on the subarray to the left of the benchmark element, then perform quick sort on the subarray to the right, until the entire array is sorted.

IV. Experimental verification and analysis

IV. A. Analysis of Vulnerability Mining Results

IV. A. 1) Test Set

LAVA is a technique for inserting vulnerabilities and faults into programs. LAVA technology is widely used to construct effect evaluation test sets for various types of fuzz testing. The LAVA-M dataset is a dataset obtained by inserting faults into four programs (uniq, who, md5sum, and base64) using LAVA technology. It is a widely used effect evaluation test set in the field of fuzz testing.

In addition, two real programs were selected for crash reproduction in the testing: GNU Binutils is a collection of binary analysis tools for the Linux platform. Lib PNG is a relatively low-level image library for reading and writing PNG files, with nearly 500,000 lines of code.

IV. A. 2) LAVA-M Test

This test will compare the performance of AFL, AFLGO, and the proposed method in LAVA-M, evaluating the efficiency of each tool in detecting vulnerabilities given a target node. The test will be repeated 15 times, with each run lasting 20 hours. The proposed method uses coverage count, average discovery time, and performance gain as primary metrics. Coverage count measures the number of unique crashes covering the target node, where a unique crash refers to a path that triggers the crash and introduces new nodes. The average discovery time represents the average time taken to cover the target node. Performance gain is calculated as the ratio of the average discovery time of AFL or AFLGO to the average discovery time of the proposed method.

Table 1 shows the test results of various fuzz testing tools on the LAVA-M dataset. The proposed method achieves the fastest coverage of the target site. Compared to the non-directed fuzz tester AFL, the average speed of reaching the target site (i.e., the average performance gain) improved by 3.43 times, and compared to the directed fuzz tester AFLGO, the average speed of reaching the target site improved by 1.45 times. In terms of coverage count, both the directed AFLGO and the proposed method outperform AFL, demonstrating the advantage of directed fuzz testing in vulnerability reproduction. The small difference in test results is because the

number of inputs that can cause the target node to crash is limited, and the program size is not very large, so within 20 hours, both AFL and AFLGO are highly likely to obtain the majority of valid crash inputs.

Table 1: Experiment results on the LAVA-M dataset

Test object	Tools	Coverage times	Average discovery time/s	Performance gain
who	AFL	4	69126	3.74
	AFLGO	5	29466	1.59
	Our method	7	18502	—
base64	AFL	5	4035	4.76
	AFLGO	6	1604	1.89
	Our method	6	848	—
md5sum	AFL	5	68158	1.77
	AFLGO	5	43090	1.12
	Our method	6	38522	—
uniq	AFL	6	52551	3.46
	AFLGO	8	18097	1.19
	Our method	8	15192	—

IV. A. 3) CVE Vulnerability Testing

To reflect the actual situation, this test selected eight reproducible CVE vulnerabilities from the vulnerability reports of Lib PNG and Binutils, and conducted 10 repeated experiments to obtain the average results. These vulnerabilities can be identified by their CVE numbers.

Table 2 presents the experimental results of various fuzz testing tools in terms of reproducing real vulnerabilities. The primary metrics for evaluating the performance of fuzz testing tools include average discovery time, performance gain, and P-value (the probability of observing results more extreme than the obtained sample observations when the null hypothesis is true). The P-value measures the degree of mismatch between the sample data and a given statistical model. In multiple experiments, a P-value less than 0.05 is generally considered to indicate a low probability of unexpected results due to chance, and the difference between the two is significant and stable. Since AFLGO and the fuzzy testing scheme in this paper are both directional fuzzy testing, the P-values between the scheme in this paper and AFLGO are provided in the results to demonstrate the superior performance of the scheme in this paper.

From the comparison test results of LibPNG and Binutils, it can be seen that in the reproduction of the aforementioned vulnerabilities, the proposed scheme performs better, with an average speed improvement of 3.67 times compared to AFL and 1.84 times compared to AFLGO. This indicates that the proposed scheme has stronger adaptability and specificity, enabling it to complete fuzz testing with higher efficiency under user-specified target nodes and achieve vulnerability mining in the software supply chain. Compared with the test results on the LAVA-M dataset, the proposed method achieves higher performance gains than AFL and AFLGO, demonstrating its superior performance in complex real-world environments.

For the vulnerabilities CVE-2016-4491 and CVE-2016-6131, the proposed method exhibits a more significant advantage over AFL and AFLGO. From the average discovery time, it can be seen that these two vulnerabilities are relatively complex and difficult to detect, requiring more time to explore. This reflects that the proposed method, through deep reinforcement learning, can more effectively select test samples, making the fuzz testing process more targeted and efficient. Additionally, the p-values obtained by comparing with AFLGO are all below 0.05, indicating that the proposed method has strong stability.

IV. B. Analysis of Vulnerability Fix Effectiveness

IV. B. 1) Experimental Data

This paper obtains defects from two publicly available vulnerability datasets, Big-Vul and CVEFixes, and constructs an experimental dataset. The Big-Vul dataset crawls the CVE database to obtain information such as CVE-ID, CVE severity scores, and CVE summaries. CVEFixes directly crawls vulnerabilities from the US National Database and changes the data source and retrieval time range.

IV. B. 2) Evaluation Indicators

Since this paper deals with vulnerability repair location identification and vulnerability repair, the evaluation indicators in this paper include repair location identification effectiveness evaluation indicators and vulnerability repair effectiveness evaluation indicators.

Table 2: Experiment results on the crash reproduction

CVE	AFL		AFLGO		Our method	P value
	Average discovery time/s	Performance gain	Average discovery time/s	Performance gain	Average discovery time/s	
CVE-2016-4487	844	2.45	523	1.52	345	0.034
CVE-2016-4488	1666	3.40	917	1.87	490	0.006
CVE-2016-4489	1335	3.35	688	1.72	399	0.024
CVE-2016-4491	31694	2.95	28105	2.61	10760	0.009
CVE-2016-4492	953	2.45	672	1.73	389	0.042
CVE-2016-6131	33932	3.45	21293	2.17	9831	0.011
CVE-2011-2501	2207	4.54	686	1.41	486	0.008
CVE-2011-3328	12873	6.74	3256	1.70	1910	0.014

(1) Repair location identification effectiveness evaluation indicators

When evaluating the effectiveness of repair location positioning, this paper uses recall rate @Top-n and MFR (mean first rank) as metrics. Recall rate @Top-n is a commonly used evaluation metric in defect localization. This paper references its definition to assess the effectiveness of repair location localization, measuring the extent to which the standard answer appears among the top n items in the prediction list. The smaller the value of n, the more accurate the defect localization results. In this paper's experiments, the values of n selected are 1, 5, 10, and all. A higher Recall Rate @ Top-n value indicates better defect localization performance. Recall Rate @ Top-all is denoted as Recall Rate. MFR is used to calculate the average rank value of the first defect code element in the predicted list of suspicious code elements. A lower MFR value indicates higher defect localization accuracy.

(2) Repair Effect Evaluation Metrics

This paper adopts the recall rate, a common metric in the field of defect repair, as the evaluation metric. Additionally, to more fully measure the effectiveness of patch generation, the recall rate @Top-n, a commonly used evaluation metric in the field of automatic program repair, is also adopted. This metric measures the prevalence of the standard answer among the top n items in the prediction list.

IV. B. 3) Comparison of experimental results

This paper compares the performance of the LLM-based software supply chain vulnerability repair method with three baseline methods on a dataset. The comparison results between the proposed method and the baseline methods are shown in Figure 1. Compared with the baseline methods, the proposed LLM-based software supply chain vulnerability repair method demonstrates significant performance improvements. Specifically, in terms of recall, the recall rate of the proposed method reached 33.27%, representing improvements of 142.49%, 92.87%, and 39.38% over the baseline methods VRepair, VulRepair, and CotRepair, respectively. This means that the proposed method can generate 142.49%, 92.87%, and 39.38% more correct repair programs compared to the baseline methods. In terms of recall rate @Top-1, the proposed method achieved improvement rates of 250.50%, 86.28%, and 36.15% compared to the baseline methods VRepair, VulRepair, and CotRepair, respectively. Similarly, the proposed method also outperformed the baseline methods in terms of recall rate @Top-5 and recall rate @Top-10. This paper also uses the Wilcoxon signed-rank test to examine whether there are significant differences in the rankings of effective repair programs in the prediction results between this method and each baseline method, in order to verify whether there are significant differences in repair performance between this method and each baseline method. The evaluation results show that the p-values between this method and each baseline method are all less than 0.01, indicating that the performance differences between this method and each model are highly significant.

In summary, for software supply chain vulnerability programs, the LLM-based software supply chain vulnerability repair method proposed in this paper can effectively repair defects and significantly outperforms the baseline methods.

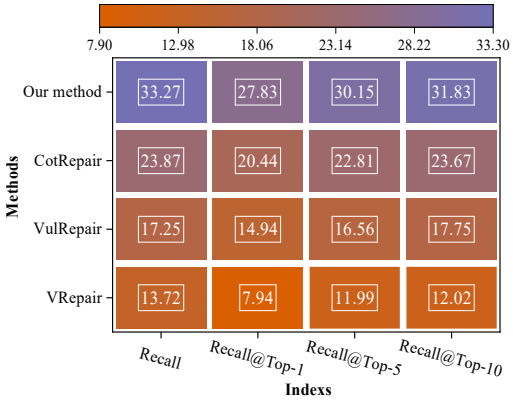


Figure 1: The repair performance comparison results of the method and the baseline method

IV. B. 4) Repair location prediction

Accurate vulnerability repair requires precise identification of the repair location. The prediction accuracy of various methods for repair locations is shown in Figure 2. Compared to the baseline method, the method proposed in this paper can more accurately capture the location of defects in the code. Specifically, the recall rate for the repair location using the method proposed in this paper is 52.39%, meaning that, without considering the correctness of the repair program, 52.39% of the repair programs generated by the method proposed in this paper were modified at the correct repair location. The recall rates for VRepair, VulRepair, and CotRepair are 19.25%, 41.54%, and 34.81%, respectively. Meanwhile, the proposed method also shows a significant improvement in the MFR metric compared to the baseline method. Compared to VulRepair, which currently performs well, the proposed method reduces the MFR metric by 19.53%, indicating that it can more accurately predict the locations of defects in vulnerable code.

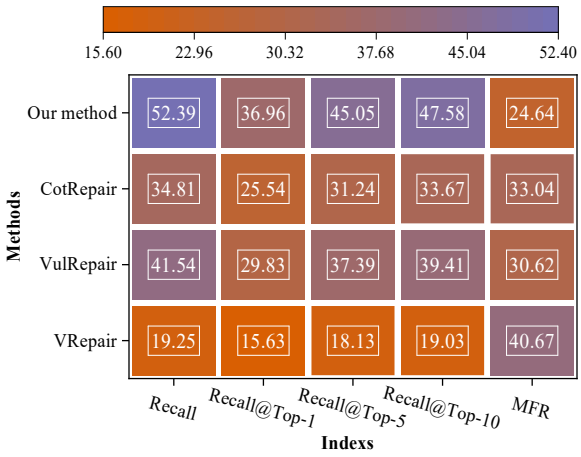


Figure 2: The prediction effect of all parties on repair location

IV. B. 5) Impact of defect types

Figure 3 shows the performance of the method described in this paper on the 10 most frequently occurring vulnerability categories. The method performed best on CWE-264 vulnerabilities, achieving a recall rate of 45.75%, and fixed 44.94%, 35.74%, and 35.09% of vulnerabilities in CWE-190, CWE-399, and CWE-416, respectively. The worst performance was observed for CWE-476 and CWE-20, with only 20.83% and 24.52% of vulnerabilities fixed, respectively. Overall, the recall rate was 33.23%, while the recall rate of the method described in this paper on the full dataset was 33.27%, with a small difference in recall rate compared to the top 10 most frequently occurring vulnerability categories. This means that the method described in this paper performs similarly on less frequently occurring vulnerability categories as it does on the most frequently occurring ones, indicating that it can learn vulnerability repair methods without requiring a large amount of vulnerability data.

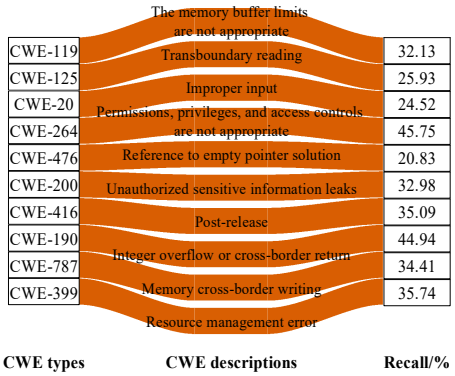
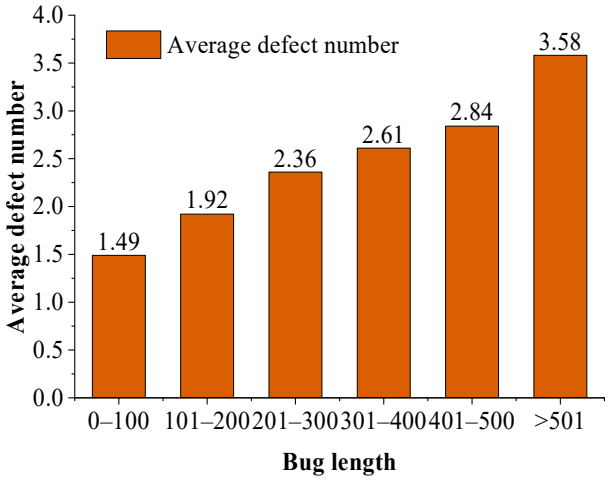


Figure 3: Performance of the 10 highest frequency vulnerabilities

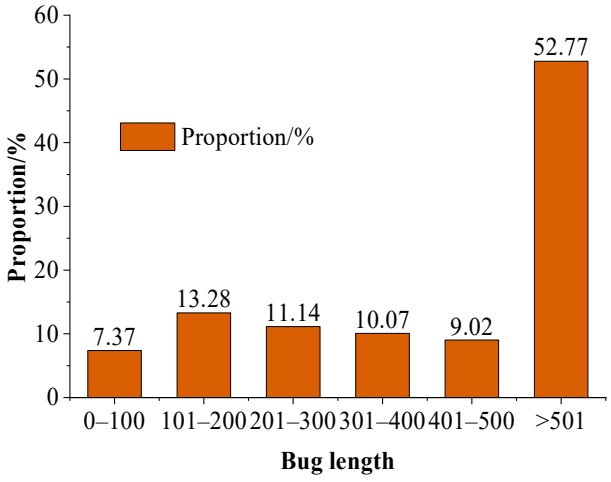
IV. B. 6) Impact of Vulnerability Length

Figure 4 shows the performance of each method on vulnerability programs of different lengths. Overall, the performance of VRepair, VulRepair, CotRepair, and the method described in this paper decreases as the length of the vulnerability program and the number of defective lines increase. Among vulnerability programs shorter than 100 lines, VulRepair, CotRepair, and the method proposed in this paper repaired 44.67%, 46.05%, and 59.57% of the vulnerability programs, respectively. For vulnerability programs with lengths between 300 and 400 lines, they repaired 16.75%, 18.08%, and 38.09% of the vulnerability programs, respectively. Compared to VRepair, VulRepair, and CotRepair, the method proposed in this paper shows a more significant performance improvement on longer vulnerability programs. Specifically, compared to VRepair, VulRepair, and CotRepair, the proposed method achieved improvements of 77.03%, 33.36%, and 29.36%, respectively, for vulnerability programs with lengths of 0–100, and improvements of 168.98%, 74.91%, and 52.18%, respectively, for vulnerability programs with lengths of 401–500.

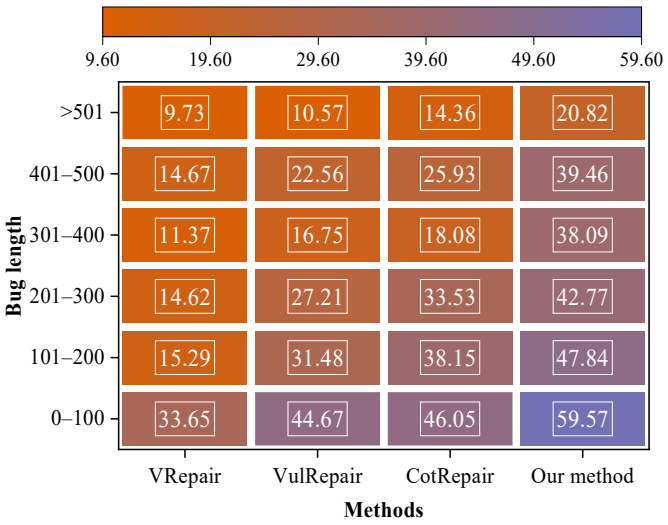
Additionally, this paper investigates the complexity of vulnerability repair by analyzing the average number of defective lines in vulnerability programs of different lengths. The study found that in vulnerability programs with lengths ranging from 0 to 100, the average number of lines involved in vulnerabilities was only 1.49, while in vulnerability programs with lengths ranging from 300 to 400, the average number of lines involved in vulnerabilities was 2.61. This indicates that as the average length of vulnerabilities increases, the average number of defect lines also significantly increases, leading to a corresponding increase in the difficulty of vulnerability repair. Therefore, the performance of various software supply chain vulnerability repair methods decreases as the average number of defect lines increases. To investigate the effectiveness of the proposed method in addressing multi-line defects, its performance was evaluated on vulnerabilities with more than one defect line. The evaluation results showed that the proposed method achieved a recall rate of 19.45% in this scenario, indicating that it can still accurately identify and address vulnerability locations even in cases of multi-line defects.



(a) Average defect number



(b) Proportion/%



(c) Recall/%

Figure 4: The performance of each algorithm on different length vulnerability program

V. Conclusion

The development of network information technology has led to an increase in the scale of software, while the likelihood of code containing vulnerabilities has also risen. This study investigates software supply chain vulnerability detection methods based on fuzz testing technology, constructs a fuzz testing method based on reinforcement learning, and combines it with large language models to propose a software supply chain vulnerability repair method. Through experiments, the effectiveness of the proposed vulnerability detection and repair methods is verified, with the main results as follows:

(1) In tests on the LAVA-M dataset, the proposed vulnerability detection method achieves faster target site coverage speeds, outperforming the AFL and AFLGO methods by 3.43 times and 1.45 times, respectively. In terms of reproducing real vulnerabilities, the proposed method identifies target vulnerabilities on average 3.67 times and 1.84 times faster, indicating significant improvements in guidance and targeting capabilities, enabling rapid identification of potential vulnerability paths.

(2) The results of multiple metrics for the paper's LLM-based vulnerability repair method outperform other methods, with recall rates 39.38% to 142.49% higher, and significant differences of over 1% compared to other methods. In the repair programs generated by this method, the recall rate for predicting repair locations reached 52.39%, while the MFR metric decreased by 19.53% compared to VulRepair, demonstrating excellent performance in predicting the repair locations of vulnerability code. Additionally, under different vulnerability types and program lengths, it exhibited better repair effects than other methods.

Fuzz testing, as a vulnerability detection method, can efficiently identify vulnerabilities in software compared to manual code audit-based vulnerability detection methods. This paper optimizes fuzz testing technology using the DDPG reinforcement learning algorithm and constructs an automated vulnerability repair path for the software supply chain, providing security assurance for deploying vulnerability detection and security protection tools across all stages of the software supply chain process.

Funding

This research was supported by the Science and Technology Major Projects of CNOOC Energy Technology & Services Limited: “Research and Application of Software Supply Chain Security Testing Techniques” (HYFZ-ZX-XK-2022-02).

References

[1] Zhao, S., Zhu, J., & Peng, J. (2024). Software Vulnerability Mining and Analysis Based on Deep Learning. Computers, Materials & Continua, 80(2).

[2] Kalouptsoglou, I., Siavvas, M., Ampatzoglou, A., Kehagias, D., & Chatzigeorgiou, A. (2023). Software vulnerability prediction: A systematic mapping study. Information and Software Technology, 164, 107303.

[3] Shamal, P. K., Rahamathulla, K., & Akbar, A. (2017, March). A study on software vulnerability prediction model. In 2017 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET) (pp. 703-706). IEEE.
[4] Kalouptsoglou, I., Siavvas, M., Ampatzoglou, A., Kehagias, D., & Chatzigeorgiou, A. (2024, July). Vulnerability classification on source code using text mining and deep learning techniques. In 2024 IEEE 24th International Conference on Software Quality, Reliability, and Security Companion (QRS-C) (pp. 47-56). IEEE.
[5] Adithya, A., Vyas, V., Mohan, M., Aaswin, V. A., & Lanka, S. (2024). Vulnerability Scanning by CPE-CVE Matching. Grenze International Journal of Engineering & Technology (GIJET), 10.
[6] Li, X., Chen, J., Lin, Z., Zhang, L., Wang, Z., Zhou, M., & Xie, W. (2017, August). A mining approach to obtain the software vulnerability characteristics. In 2017 fifth international conference on advanced cloud and big data (CBD) (pp. 296-301). IEEE.
[7] Murtaza, S. S., Khreich, W., Hamou-Lhadj, A., & Bener, A. B. (2016). Mining trends and patterns of software vulnerabilities. Journal of Systems and Software, 117, 218-228.
[8] Guo, W., Fang, Y., Huang, C., Ou, H., Lin, C., & Guo, Y. (2022). HyVulDect: a hybrid semantic vulnerability mining system based on graph neural network. Computers & Security, 121, 102823.
[9] Malhotra, R., & Vidushi. (2024). Text mining based an automatic model for software vulnerability severity prediction. International Journal of System Assurance Engineering and Management, 15(8), 3706-3724.
[10] Aslan, Ö., Aktuğ, S. S., Ozkan-Okay, M., Yilmaz, A. A., & Akin, E. (2023). A comprehensive review of cyber security vulnerabilities, threats, attacks, and solutions. Electronics, 12(6), 1333.
[11] Humayun, M., Niazi, M., Jhanjhi, N. Z., Alshayeb, M., & Mahmood, S. (2020). Cyber security threats and vulnerabilities: a systematic mapping study. Arabian Journal for Science and Engineering, 45, 3171-3189.
[12] Ruohonen, J., Rauti, S., Hyrynsalmi, S., & Leppänen, V. (2018). A case study on software vulnerability coordination. Information and Software Technology, 103, 239-257.
[13] Medeiros, I., Neves, N., & Correia, M. (2015). Detecting and removing web application vulnerabilities with static analysis and data mining. IEEE Transactions on Reliability, 65(1), 54-69.
[14] Pistoia, Marco, et al. "A survey of static analysis methods for identifying security vulnerabilities in software systems." IBM systems journal 46.2 (2007): 265-288.
[15] Filus, Katarzyna, et al. "Efficient feature selection for static analysis vulnerability prediction." Sensors 21.4 (2021): 1133.
[16] Xu, Y., Zhang, M., Wang, X., Chen, J., Liang, R., Zhen, Y., & Zhen, C. (2023, May). A Review of Code Vulnerability Detection Techniques Based on Static Analysis. In International Conference on Computational & Experimental Engineering and Sciences (pp. 251-272). Cham: Springer Nature Switzerland.
[17] Li, Y., Ma, L., Shen, L., Lv, J., & Zhang, P. (2019). Open source software security vulnerability detection based on dynamic behavior features. Plos one, 14(8), e0221530.
[18] Afianian, A., Niksefat, S., Sadeghiyan, B., & Baptiste, D. (2019). Malware dynamic analysis evasion techniques: A survey. ACM Computing Surveys (CSUR), 52(6), 1-28.
[19] Li, J., Chen, J., Huang, M., Zhou, M., Xie, W., Zeng, Z., ... & Zhang, Z. (2018). An integration testing framework and evaluation metric for vulnerability mining methods. China Communications, 15(2), 190-208.
[20] Padmanabhuni, B. M., & Tan, H. B. K. (2016). Auditing buffer overflow vulnerabilities using hybrid static–dynamic analysis. IET Software, 10(2), 54-61.
[21] Yitagesu, S., Xing, Z., Zhang, X., Feng, Z., Bi, T., Han, L., & Li, X. (2025). Systematic Literature Review on Software Security Vulnerability Information Extraction. ACM Transactions on Software Engineering and Methodology.
[22] Kim, S., Kim, R. Y. C., & Park, Y. B. (2016). Software vulnerability detection methodology combined with static and dynamic analysis. Wireless Personal Communications, 89, 777-793.
[23] Ghaffarian, S. M., & Shahriari, H. R. (2017). Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. ACM computing surveys (CSUR), 50(4), 1-36.
[24] Shah, I. A., Rajper, S., & ZamanJhanjhi, N. (2021). Using ML and Data-Mining Techniques in Automatic Vulnerability Software Discovery. International Journal of Advanced Trends in Computer Science and Engineering, 10(3).
[25] Jie, G., Xiao-Hui, K., & Qiang, L. (2016, June). Survey on software vulnerability analysis method based on machine learning. In 2016 IEEE first international conference on data science in cyberspace (DSC) (pp. 642-647). IEEE.
[26] Chernis, B., & Verma, R. (2018, March). Machine learning methods for software vulnerability detection. In Proceedings of the fourth ACM international workshop on security and privacy analytics (pp. 31-39).
[27] Scandariato, R., Walden, J., Hovsepyan, A., & Joosen, W. (2014). Predicting vulnerable software components via text mining. IEEE Transactions on Software Engineering, 40(10), 993-1006.
[28] Lin, G., Wen, S., Han, Q. L., Zhang, J., & Xiang, Y. (2020). Software vulnerability detection using deep neural networks: a survey. Proceedings of the IEEE, 108(10), 1825-1848.
[29] Yamaguchi, F. (2017). Pattern-based methods for vulnerability discovery. it-Information Technology, 59(2), 101-106.
[30] Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y., & Bener, A. (2010). Defect prediction from static code features: current results, limitations, new approaches. Automated Software Engineering, 17, 375-407.
[31] Hanif, H., Nasir, M. H. N. M., Ab Razak, M. F., Firdaus, A., & Anuar, N. B. (2021). The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches. Journal of Network and Computer Applications, 179, 103009.
[32] Duan, X., Wu, J., Ji, S., Rui, Z., Luo, T., Yang, M., & Wu, Y. (2019, August). VulSniper: Focus Your Attention to Shoot Fine-Grained Vulnerabilities. In IJCAI (pp. 4665-4671).
[33] Liu, K., Kim, D., Bissyandé, T. F., Yoo, S., & Le Traon, Y. (2018). Mining fix patterns for findbugs violations. IEEE Transactions on Software Engineering, 47(1), 165-188.
[34] Lin, G., Zhang, J., Luo, W., Pan, L., De Vel, O., Montague, P., & Xiang, Y. (2019). Software vulnerability discovery via learning multi-domain knowledge bases. IEEE Transactions on Dependable and Secure Computing, 18(5), 2469-2485.
[35] Ghaffarian, S. M., & Shahriari, H. R. (2021). Neural software vulnerability analysis using rich intermediate graph representations of programs. Information Sciences, 553, 189-207.
[36] Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., & Chen, Z. (2021). Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Transactions on Dependable and Secure Computing, 19(4), 2244-2258.

[37] Woergoetter, F., & Porr, B. (2008). Reinforcement learning. Scholarpedia, 3(3), 1448.
[38] AlMajali, A., Al-Abed, L., Ahmad Yousef, K. M., Mohd, B. J., Samamah, Z., & Abu Shhadeh, A. (2024). Automated Vulnerability Exploitation Using Deep Reinforcement Learning. Applied Sciences, 14(20), 9331.
[39] Pham, V. H., Chuong, N. P., Thai, P. T., & Duy, P. T. (2024). A Coverage-guided Fuzzing Method for Automatic Software Vulnerability Detection using Reinforcement Learning-enabled Multi-Level Input Mutation. IEEE Access.
[40] Dass, S., & Siami Namin, A. (2021). Reinforcement learning for generating secure configurations. Electronics, 10(19), 2392.