Enhancing Fine-Grained Vulnerability Detection with Reinforcement Learning

Yuan Jiang, Zhichen Qu, Christoph Treude, Xiaohong Su and Tiantian Wang

Abstract—The rapid growth of vulnerabilities has significantly accelerated the development of automated vulnerability detection methods, especially those based on data-driven models. However, most of them primarily focus on extracting accurate code representations while overlooking the complex vulnerability patterns among vulnerable statements, thereby leaving room for improvement. To overcome this limitation, we present a novel reinforcement learning framework (*RLFD*) for detecting vulnerabilities at a fine-grained level. *RLFD* redefines the detection task as a sequential decision-making process and then employs reinforcement learning to automatically learn vulnerability-relevant structures from code snippets. Moreover, by designing reward functions aligned with fine-grained evaluation metrics, *RLFD* focuses on the co-existence relations among statements from a global perspective, enabling the model to capture complex interactions that lead to vulnerabilities. Additionally, the framework utilizes CodeBERT-HLS for code representation, ensuring consistency with the state-of-the-art method while highlighting the improvements brought by the proposed reinforcement learning-based approach. Comprehensive experiments show that our method achieves a locating precision (IoU) of 69.7% and a Top-5% Acc of 67.7% on the *big_vul* dataset, outperforming the state-of-the-art method by an overall 3.4% improvement in IoU. Notably, our method achieves up to a 19.7% increase in IoU for specific categories, e.g., CWE-416 (use-after-free).

Index Terms—Vulnerability detection, Fine-grained, Data-driven methods, Reinforcement learning

1 Introduction

Despite significant efforts to enhance software security, vulnerabilities remain a major challenge in modern software development. Advances in Deep Learning (DL), particularly in Large Language Models (LLMs), have prompted new directions for developing more intelligent vulnerability detection methods. Existing DL-based methods typically treat vulnerability detection as a classification task at the file, function, or code slice level [1], [2]. The main differences among these methods lie in the neural networks used for feature extraction, such as sequence-based models, Graph Neural Networks (GNNs), and Transformer-based models. Although these methods show promising performance improvements, several limitations persist. A significant limitation is their inability to pinpoint the precise locations of vulnerabilities, providing only coarse-grained insights that are insufficient for developers to quickly and accurately identify and address vulnerabilities in their code.

To overcome this limitation, recent works have proposed DL-based fine-grained methods for statement-level vulnerability detection. Among these methods, StagedVulBERT has achieved notable detection performance [3]. It first employs the pre-trained code language (PCL) model, CodeBERT-HLS, to learn the feature representation of each statement and then assesses the likelihood of each statement being vulnerable based on the learned features. The workflow of StagedVulBERT is illustrated in the blue part of Fig. 1, which operates similarly to many other fine-grained detection approaches [4].

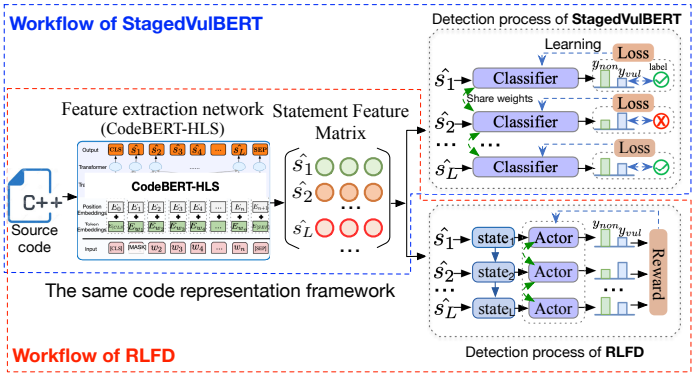

Fig. 1. Illustration of the StagedVulBERT and the proposed RLFD framework.

As shown in the workflow of StagedVulBERT, a great effort has been made toward acquiring accurate code representations. However, is it sufficient to determine if a statement is vulnerable based solely on its features? More specifically, a critical question arises:

> Q: Do neural network-extracted statement features provide sufficient information for high-performance fine-grained vulnerability detection?

Our investigation reveals that while accurately extracting code features is important for achieving high performance, relying solely on a simple classifier with a feed-forward neural network (FFNN) is insufficient to establish a clear mapping between each statement's feature and the overall vulnerability patterns. The primary challenge lies in

- *Y. Jiang (jiangyuan@hit.edu.cn, yuanjiang@smu.edu.sg), Z. Qu (24s003112@stu.hit.edu.cn), X. Su (sxh@hit.edu.cn), and T. Wang (wangtiantian@hit.edu.cn), are with the School of Computer Science and Technology, Harbin Institute of Technology, Harbin, Heilongjiang, 15001.*
- *C. Treude is with the School of Computing and Information Systems, Singapore Management University, Singapore. E-mail: ctreude@smu.edu.sg*

the binary classifier of these methods, where the FFNN takes each statement as input and learns its parameters based on the statement's label ("0" for non-vulnerable and "1" for vulnerable). In this setup, the learning process treats each statement independently, without considering the co-existence relations between multiple vulnerable statements that collectively contribute to a vulnerability. As a result, these methods struggle to effectively identify vulnerability patterns that depend on interactions among multiple statements from a global perspective, reducing their effectiveness in providing detailed, actionable insights for developers.

To address the limitations of existing methods, we propose *RLFD*, *a **R**einforcement **L**earning-based framework for effectively extracting vulnerability patterns to enhance **F**ine-grained **D**etection performance*. This framework is highlighted in the red part of Fig. 1, which introduces several innovations over traditional approaches.

Our method redefines fine-grained vulnerability detection as a sequential decision-making problem. By employing reinforcement learning (RL), RLFD captures the complex interactions between statements that jointly lead to a vulnerability. The core innovation lies in this reformulation and in designing tailored reward functions aligned with fine-grained evaluation metrics (e.g., Intersection over Union (IoU) and Top-%k accuracy). This reward guides the learning process to emphasize the co-existence relations among statements, enabling the model to identify vulnerability patterns from a global perspective. This strategy addresses the limitation of traditional methods, which treat each statement independently when making decisions.

For code representation, RLFD leverages CodeBERT-HLS, the same framework used in the state-of-the-art method StagedVulBERT [3]. By adopting CodeBERT-HLS, we ensure consistency in code representation, enabling a clear focus on the improvements brought by our RL-based pattern extraction. This design choice also aims to answer the earlier question by evaluating how our method enhances the modeling of vulnerability patterns without changing the baseline code representation.

We assess the effectiveness of RLFD on the largest publicly available vulnerability dataset. The experimental results reveal that our model outperforms existing approaches in detecting vulnerabilities at the statement level, achieving an overall 3.4% improvement in IoU. Specifically, RLFD achieves up to a 19.7% increase in IoU for certain vulnerability categories, such as CWE-416 (use-after-free).

The main **contributions** of our paper are:

- **A novel RL-based framework**. We propose using RL techniques to precisely identify lines of code that contain vulnerabilities, simulating manual security code review. To our knowledge, this is the first work to apply RL to fine-grained vulnerability detection.
- **Effective learning of vulnerability patterns**. We design reward functions aligned with fine-grained evaluation metrics to guide the RL process, which focuses the model on the co-existence relations among statements, enhancing its ability to identify complex vulnerability patterns and resulting in a 170.2% performance improvement.
- **Extensive experiments**. We conduct extensive experiments on the large *big_vul* dataset and a real-world project (Linux kernel) to validate our method. Results show a 3.4% IoU improvement over the state-of-the-art on *big_vul*. In the Linux kernel case study, RLFD detected 31 vulnerabilities, whereas Staged-VulBERT detected only 41.9% of them.

2 Background and Motivation

2.1 Motivating Example

Fig. 2 illustrates a vulnerable code snippet [1] from the *ssdp-responder* project, representing an instance of CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer). This vulnerability is documented under CVE-2019-14323 and involves a buffer overflow due to improper handling of received data in a network application. As a preview, in our experiments, the proposed RL-based method successfully detects the vulnerable lines, whereas StagedVulBERT, a leading statement-level classification-based approach, fails to identify the vulnerability at the fine-grained level.

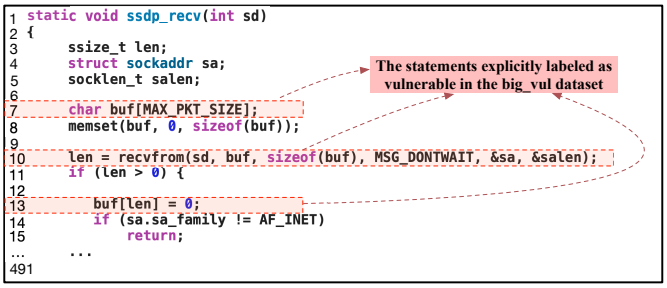

Fig. 2. A code snippet from the *ssdp-responder* project illustrating a buffer overflow vulnerability.

Observations. From this example, we can observe the following:

1) The code defines a buffer *buf* with a fixed size *MAX_PKT_SIZE* and initializes it using memset.

2) It then calls *recvfrom* to receive data into *buf*, specifying the maximum number of bytes to read as *sizeof(buf)*. The number of bytes received is stored in *len*.

3) After receiving the data, the code attempts to null-terminate the buffer by writing a zero byte at *buf[len]*.

The vulnerability arises when *len* equals *MAX_PKT_SIZE*. In this case, *buf[len] = 0* attempts to write beyond the allocated buffer, as valid indices range from 0 to *MAX_PKT_SIZE - 1*. This off-by-one error can lead to memory corruption or a program crash.

These observations reveal two basic facts. The first is that a file or function may contain many lines of code (e.g., 491 lines in this example function), but only a few lines contribute to the vulnerability. Identifying these critical lines can be tedious and time-consuming, emphasizing the need for automated, fine-grained vulnerability detection tools.

The second, and more important fact, is that accurately detecting vulnerabilities cannot be achieved by analyzing

1. https://github.com/troglobit/ssdp-responder/commit/ce04b1f29a137198182f60bbb628d5ceb8171765

each statement in isolation. The vulnerability in Fig. 2 emerges from the interplay of multiple statements. It involves the allocation of *buf* with a fixed size (Line 7), receiving data into *buf* without proper bounds checking (Line 10), and writing to *buf[len]* (Line 13) without verifying that *len* is within the valid range. Thus, any effective vulnerability detection method must account for these co-existence relations between statements to uncover the underlying patterns that collectively lead to vulnerabilities. Given this motivation, our research aims to develop a method, RLFD, for predicting vulnerable lines of code by considering the co-existence relations between multiple statements using RL.

2.2 Our Motivation for Using Reinforcement Learning

2.2.1 Reinforcement Learning

Reinforcement Learning (RL) is a field within machine learning that focuses on solving sequential decision-making tasks by enabling an agent to learn control policies through interactions with its environment E [5] [6]. In the standard reinforcement learning paradigm, considering an environment consisting of a set of internal states S =$\{s_0, ..., s_I\}$, a learning agent decides which action from a predefined set of actions A =$\{a_0, ..., a_K\}$ to perform at state s_t by following certain policies or rules. Hence, the agent's decision-making procedure at each time step t can be characterized by a policy, $\pi(s_t, a_t, \theta) = p_\pi(a_t|s_t; \theta)$, where $\forall s_t \in S, a_t \in A$ and θ denotes parameters [7]. We assume that π is differentiable almost everywhere with respect to its parameter, i.e., $\frac{\partial \pi}{\partial \theta}$ exists, where π is an abbreviation for $\pi(s_t, a_t, \theta)$. Next, performing the action a_t will produce an effect on the environment, which in turn results in a new state s_{t+1} and returns a reward r_t to the agent. Formally, the dynamics of the environment are characterized by the state transition probabilities $p^{a_t}_{s_t s_{t+1}} = p_r(s_{t+1}|s_t, a_t)$, and the expected rewards $R^{a_t}_{s_t} = E(r_t|s_t, a_t)$, for all $\forall s_t, s_{t+1} \in S, a_t \in A$ [7]. Each such action forms a transition tuple (s_t, a_t, r_t, s_{t+1}) of a Markov Decision Process (MDP) [8] [5]. The learner's objective is to find a policy that assigns actions to states in a manner that maximizes the cumulative reward accumulated over time steps.

2.2.2 Why do we use reinforcement learning to solve fine-grained vulnerability detection?

Fine-grained vulnerability detection aims to precisely predict the potentially vulnerable lines of code for each candidate vulnerability (e.g., function or file) which has been identified by the coarse-grained detection model. We refer to this process as the vulnerability-relevant structure generation, which can be viewed conceptually and modelled as a sequential decision-making problem: the current decision of the vulnerability statement affects the following decisions. Hence, policy gradient RL (i.e., the policy-based approaches) can be applied to solve this problem. To see this, we formalize the above intuition as follows:

- Given a code snippet $f = \{c_1, c_2, ..., c_L\}$ where c_i corresponds to a line of code[2], our goal is to find a policy that generates a sequence of actions $a = \{a_1, a_2, ..., a_L\}$ from the discrete action space A with the objective of maximizing the expected reward, where a_i indicates whether c_i is relevant ($a_i = 1$) or not relevant ($a_i = 0$) to any vulnerability. All statements of the corresponding non-zero actions form the vulnerability-relevant structure of the code snippet predicted by the policy.

Intuitively, the proposed method can be regarded as a simulation of manual security code review, which is also well known as the code inspection proposed by Fagan [9]. For example, suppose a developer or code reviewer checks for possible vulnerabilities of a program, he or she first reads the source code line-by-line to understand the semantic meaning of the program, and then conducts dependency analysis to find (either partial or all) semantically related statements. If these statements satisfy the patterns of vulnerable code in terms of semantics and structures, they will be marked as "vulnerable" together.

The automatic code reviewer is what we intend to learn by using the RL techniques, which can capture statistical vulnerability patterns (i.e., co-existence relations between vulnerable statements in a vulnerability) by analyzing training data and predict which statements in unknown programs may cause potential vulnerabilities. It is also important to note that, unlike real-world developers, the automatic code reviewer is specifically designed for making sequential decisions, but without the ability to perform code understanding tasks. This is because having a deep semantic understanding of the programs is another challenging problem, which is often accomplished through automatic feature engineering based on DL [10].

Modeling the fine-grained vulnerability detection task via RL is inspired by the recent work on common AI tasks through deep RL. To clearly see the differences when applying RL to the fine-grained detection task and other AI tasks, let us consider the example of using RL to play the space invaders game. The reason for choosing this example is that it is a typical application of RL. As illustrated in Fig. 3 (a), the gameplay is divided into discrete time steps, and at each time step, the agent (AI player) selects an action a_t from a predefined set of possible actions a= $\{a_0, ...a_K\}$, according to the current state. The emulator executes the action on the current state, and generates a new state for the next action prediction. The game score (i.e., cumulative reward) is updated when a reward r_t is returned to the agent [11]. Similar to the above process, when using RL for fine-grained vulnerability detection, the code review process is divided into discrete time steps, and at each time step, the agent (automatic code reviewer) only handles one statement, i.e., selects an action a_t from the possible action set (a= $\{a_0$, $...a_K\}$) for each state to determine whether the current line of code is related to any vulnerability. After applying the action to the current state, we can obtain a new state, as shown in Fig. 3 (b). The cumulative reward can be computed when all the statements are covered. More technique details for vulnerability detection using RL will be discussed in Section 4.

Overall, in this work, we provide a novel solution for the fine-grained vulnerability detection task, which models the

2. A line of code is equivalent to a statement in this paper. If a statement spans multiple lines, it will be normalized into the standard form of a line of code during preprocessing.

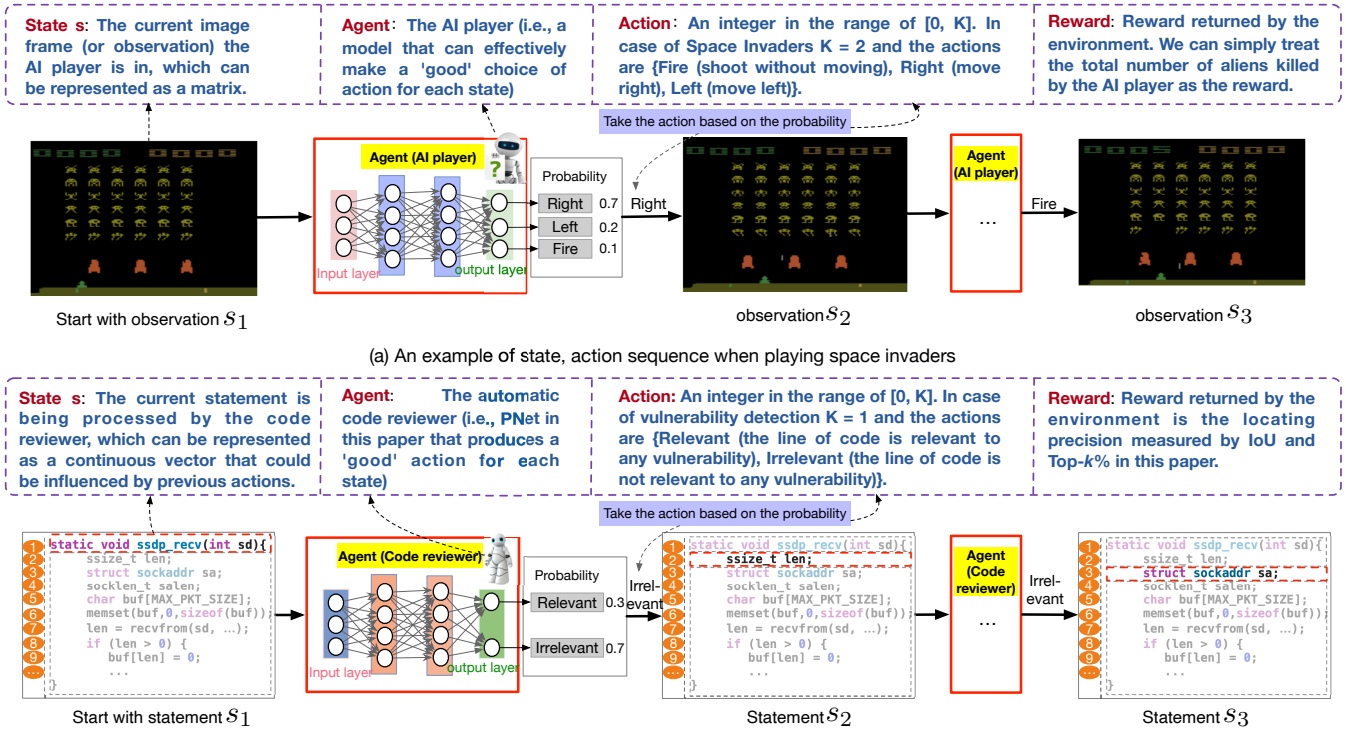

Fig. 3. Comparison when applying reinforcement learning to detect vulnerabilities at the fine-grained level and to play the space invaders game.

detection process as a sequential decision problem in order to mimic the manual security code review.

3 The Overall Framework of Our Method

Fig. 4 presents our proposed vulnerability detection framework, which consists of two phases: accurately generating code features and detecting vulnerabilities at both the coarse- and fine-grained levels. The goal of the first phase is to generate code representation for source code at the statement and program level, which better fits as input for multi-level detection. The second phase is to determine whether a program is vulnerable, and if so, pin down the vulnerable lines of code. Unlike StagedVulBERT, our method employs RL in the fine-grained detection phase to consider the co-existence relations between multiple statements, leading to more accurate identification of vulnerable lines (as illustrated in the framework comparison shown in Fig. 1). Since the fine-grained detection method via RL is what this paper focuses on, we give a brief explanation of our framework in this section and detail the main RL components of our proposed detection method in Section 4.

3.1 Code Feature Representation via CodeBERT-HLS

To accommodate semantic information pertinent to vulnerabilities, this paper employs CodeBERT-HLS, which was proposed in our previous study [3] and has demonstrated superior performance in multi-granularity vulnerability detection. The architecture of the CodeBERT-HLS network comprises three main components: TETransformer, Token2Statement Embedding, and SETransformer. Firstly, the

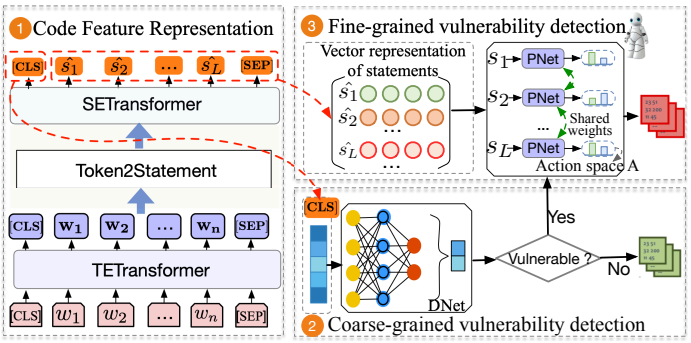

Fig. 4. Illustration of the entire proposed framework, which first generates the program and statement representations via CodeBERT-HLS and performs vulnerability detection following the coarse-to-fine-grained strategies.

TETransformer views code snippets as a sequence of code tokens, employing an architecture identical to CodeBERT to learn features for each token. Then, based on these token vector representations, Token2Statement Embedding leverages the correspondence between tokens and statements to obtain initial vector representations for each statement. Finally, the SETransformer takes these initial representations as input and learns accurate statement representations by capturing the dependencies among statements. Following these steps, we obtain effective vector representations of the input program and its statements, which apply to subsequent coarse-grained and fine-grained vulnerability detection tasks.

3.2 Coarse-to-Fine-Grained Vulnerability Detection

In the previous stage, we obtained representations for each statement and the entire code snippet, which can be regarded as feature vectors for vulnerability detection. Next, we employ an FFNN (DNet depicted in the bottom right region of Fig. 4) that takes the feature vector of a code snippet as input and predicts the corresponding vulnerability label at the coarse-grained level. Once a code snippet is predicted to be vulnerable, we proceed with fine-grained vulnerability detection by applying the proposed RL-based approach to identify vulnerability-relevant structures based on the feature representation of statements within the code snippet. As mentioned in Section 2.2, the vulnerability-relevant structure generation process can be viewed as generating a sequence of actions $a = \{a_1, a_2, ..., a_L\}$ from a discrete action space A for a given code snippet $c = \{c_1, c_2, ..., c_L\}$.

The RL-based fine-grained detection network architecture, as illustrated in Fig. 5, follows a policy-based framework [7], which has been successfully applied to many decision-making tasks, such as playing Atari [12]. We split the architecture into three submodules: (a) CodeBERT-HLS. This module provides feature representations for each line of code, and these representations are used to compute the states s for the policy network. (b) Policy Network (PNet). An action for each state is produced by the stochastic policy in this module, which is used to determine the existence of the current line of code in the vulnerability-relevant structures. (c) Environment (E). This module offers reward computation to guide the learning of PNet. Since the reward can be computed given a trajectory of actions based on the fine-grained evaluation metric, the process can be naturally addressed by the policy gradient method [7], [13].

4 Our Approach

This section provides a comprehensive explanation of the steps involved in our proposed method for fine-grained vulnerability detection using RL.

4.1 RL for Fine-grained Vulnerability Detection

To further enhance the performance of fine-grained vulnerability detection, we employ RL to learn a stochastic policy π that can discover vulnerability-relevant structures. In this paper, we define this RL process as an MDP, which can be represented as a tuple (s, a, p, r). The explanations of the state s, action a, transition probability p, and reward r in our problem are shown below.

State: In our method, the state at time step t, denoted as $\boldsymbol{s_t}$, consists of the representation of the current line of code and a context vector that summarizes information from previously predicted vulnerable statements. To construct the state $\boldsymbol{s_t}$ for the policy network, we proceed as follows.

First, we extract the vector representation $\hat{\boldsymbol{s_t}} \in \mathbb{R}^k$ for each line of code using CodeBERT-HLS. This provides a fixed-dimensional embedding that captures the semantic information of the current line of code. Next, we construct a context vector $\hat{\boldsymbol{c_t}} \in \mathbb{R}^k$ that aggregates the representations of all prior lines up to time $t-1$ that have been predicted as vulnerable. Specifically, at each time step t, the context vector is defined as:

$$\hat{\boldsymbol{c_t}} = \frac{1}{M} \sum_{i=1}^{t-1} a_i \hat{\boldsymbol{s_i}} \tag{1}$$

where $a_i \in \{0, 1\}$ is the action at time step i predicted by the PNet (with $a_i = 1$ indicating that i-th line of code is predicted as vulnerable), $\hat{\boldsymbol{s_i}}$ is the vector representation of the i-th line of code, and $M = \sum_{i=1}^{t-1} a_i$ is the total number of previous lines predicted as vulnerable. If $M = 0$ (i.e., no prior lines were predicted as vulnerable), we define $\hat{\boldsymbol{c_t}}$ as a zero vector of dimension k.

Finally, we define the state representation $\boldsymbol{s_t} \in \mathbb{R}^{2k}$ for the policy network as:

$$\boldsymbol{s_t} = \hat{\boldsymbol{c_t}} \oplus \hat{\boldsymbol{s_t}} \tag{2}$$

where $\oplus$ denotes the concatenation operation. By incorporating both the current line's representation and the context vector into the state $\boldsymbol{s_t}$, the policy network can consider the influence of prior vulnerable lines when predicting the vulnerability of the current line.

Action and Probability: The action space A in our task is assumed to be discrete and to consist of two actions: *Relevant* and *Irrelevant*. *Relevant* (denoted by "1") indicates that the line of code is relevant to any vulnerability, whereas *Irrelevant* (denoted by "0") indicates it is not. Let p_π denote a policy π determined by the policy network (i.e., PNet). When a state s_t is given, p_π can take action $a_t \in A$ by computing the transition probability: $p_\pi(a_t|s_t;\theta) = \sigma(\boldsymbol{W_P s_t} + \boldsymbol{b_P})$, where $\theta = \{\boldsymbol{W_P} \in \mathbb{R}^{2\times 2k}, \boldsymbol{b_P} \in \mathbb{R}^2\}$ denotes the parameters of PNet and $\boldsymbol{s_t}$ is the representation of state s_t.

Reward: Once the action sequence $a = \{a_1, ..., a_L\}$ is determined by the policy for a given set of states $s = \{s_1, ..., s_L\}$, we can get a trajectory $\tau = \{s_1, a_1, s_2, ..., s_L\}$ which means that, for all $t < L$, the policy takes action a_t under a certain state s_t, and will result in a new state s_{t+1}. Based on the trajectory, the environment (as shown in the right panel of Fig. 5) produces a reward $R(\tau)$ by comparing the prediction and ground truth to guide the learning of the policy. In this study, we define the reward for the trajectory τ as follows:

$$R(\tau) = \frac{|V(\tau) \cap U|}{|V(\tau) \cup U|} + w \sum_{l \in U} \frac{1}{\text{index}(l) + 1} \tag{3}$$

where $V(\tau)$ is the set of indices for all non-zero actions in a, which indicates the predicted locations (i.e., line numbers of code) of any vulnerability in the input f, and U is the ground truth of vulnerable locations. For example, if a vulnerable code snippet consists of ten lines of code, and the ground truth locations where the vulnerability occurs in code are *Line* 5 and *Line* 7 (i.e., the fine-grained label of the code gadget is [5, 7]), then the term $\frac{|V(\tau)\cap U|}{|V(\tau)\cup U|}$ can be calculated based on the predicted actions $a = \{0, 0, 0, 0, 1, 0, 1, 0, 0, 1\}$, i.e., $\frac{|V(\tau)\cap U|}{|V(\tau)\cup U|} = \frac{[5,7,10]\cap[5,7]}{[5,7,10]\cup[5,7]} = \frac{2}{3}$. According to the aforementioned definition, $\frac{|V(\tau)\cap U|}{|V(\tau)\cup U|}$ measures the deviation between the forecast and the true data, so it can also act as the metric (called IoU in the previous study [14], [15]) to assess the performance of the fine-grained detection models.

In addition, we assign penalties for each true vulnerable location based on its rank l in the ordered list of predicted

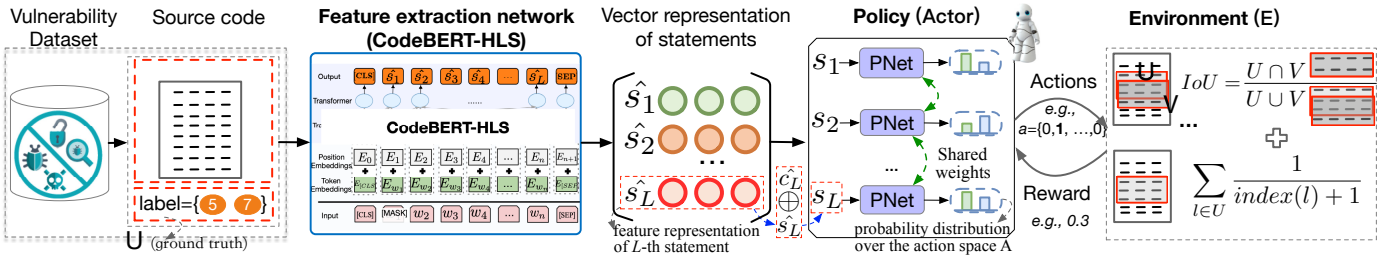

Fig. 5. Illustration of the proposed RLFD framework, which consists of CodeBERT-HLS, PNet, and Environment. CodeBERT-HLS is leveraged to learn vector representations for each line of code, which are then fed into PNet to estimate the probability of each line being vulnerable. The Environment calculates rewards based on the predicted results for all lines of code, guiding the learning of PNet to improve its prediction accuracy.

probabilities generated by $p_\pi(a_t|s_t;\theta)$. This ordered list is created by sorting the predicted probabilities for all L statements in descending order, where statements ranked higher have a greater likelihood of being vulnerable. The function index(l) returns the position of the true vulnerable location l in this ordered list. To penalize later detections, we calculate the penalty as the reciprocal of index(l), with an adjustment of 1 added to avoid division by zero. This mechanism encourages the model to assign higher probabilities to vulnerable statements over non-vulnerable ones.

Therefore, formula 3 combines a measure of accuracy in predicting vulnerable locations with a penalty for late identification of true vulnerabilities, encouraging not only accurate but also early detection within the sequence of predictions.

After obtaining the reward, we utilize the REINFORCE policy gradient algorithm [7], [16] to develop the optimal policy. We can define the standard reinforcement learning objective in terms of the above quantities as:

$$\begin{aligned}\bar{R}_\theta &= \mathbb{E}_{\tau\sim P_\theta(\tau)} R(\tau) = \sum_\tau R(\tau)P(\tau|\theta) \\ &\sim \sum_\tau R(\tau)p(s_1)\prod_{t=1}^{L} p_\pi(a_t|s_t,\theta)p(r_t,s_{t+1}|s_t,a_t) \\ &\sim \sum_\tau R(\tau)\prod_{t=1}^{L} p_\pi(a_t|s_t,\theta) \\ &\approx \sum_{n=1}^{N} R(\tau^n)\prod_{t=1}^{L} p_\pi(a_t^n|s_t^n,\theta)\end{aligned} \tag{4}$$

Where τ^n is the n-th real sample trajectory over vulnerable programs, N is the total number of trajectories, and $p(s_1)$ is the probability of being in state s_1. Since $p(s_1)$ and $p(r_t, s_{t+1}|s_t, a_t)$ are not dependent on the policy parameters θ, they can be disregarded assuming that the state transition is unknown to the agent in model free RL [17], [18]. As shown in [7], [16], a sample approximation to the gradient of the PNet parameters is:

$$\nabla_\theta \bar{R}_\theta = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{L} R(\tau^n)\nabla log p_\pi(a_t^n|s_t^n,\theta) \tag{5}$$

The PNet's parameters are then updated using this gradient and a learning rate α, which controls how quickly the parameters are adjusted in the gradient's direction within the parameter space.

$$\theta \leftarrow \theta + \alpha\nabla_\theta \bar{R}_\theta \tag{6}$$

An intuitive interpretation of the formulas 5 and 6 is that the gradient updates will be stronger for the probability of taking this action in the future. If this action is beneficial, the reward value will most probably be high, otherwise it will be low. By multiplying the rewards into the gradients, our model achieves scaling the update. This means the higher the reward is, the more the probability of taking this action increases [19]. Additionally, since the reward indicates whether the PNet correctly identifies a set of potentially vulnerable statements, through accumulated reward signals, the policy will consistently enhance its knowledge to accurately identify co-occurring vulnerable lines of code in vulnerability patterns.

4.2 Exploration Policy

In contrast to supervised learning (SL), an important characteristic of RL is that it explores the environment to build sample trajectories for learning an effective policy. Through reward-guided exploration, the PNet model theoretically will know which combinations of statements are potentially harmful and are most likely to trigger real vulnerabilities. However, from the current studies, it seems difficult to achieve this goal with SL because of the enormous number of potential combinations of statements. To balance exploration and exploitation, we employ the ϵ-greedy policy, one of the most widely used algorithms for exploration [11]. Algorithm 1 provides a detailed description of this process. Each iteration of the outer *for* loop (i.e., each epoch) of Algorithm 1 proceeds as follows, where here, ϵ represents the probability that the policy will select the optimal action; its value decreases linearly from 1.0 to 0.1 throughout the training process: First of all, for each code snippet f_i, the agent will collect a set of state-action samples $\{(s_t, a_t)\}_{t=0}^{T}$ (i.e., a trajectory) under policy π. Specifically, for each state s_t, the agent occasionally engages in random exploration with probability ϵ, and most of the time selects the optimal action with probability $1-\epsilon$ (Lines 6-15). After obtaining the trajectory, the reward can be calculated easily according to formula 3 presented in Section 4.1 (Line 16). Based on the reward and the REINFORCE algorithm (see Section 4.1), the PNet can be updated (Line 17) aiming to improve accuracy in fine-grained vulnerability detection. The above process for f_i repeats several times (e.g., $SN = 5$ as default) to discover code vulnerability patterns. Note that the update often comes in batches rather than in a single trajectory.

Algorithm 1: Epsilon greedy exploration

for *each training epoch* **do**
  for *each sample* f_i **do**
    $j = 0$;
    while $j < SN$ **do**
      Define a trajectory $\tau \leftarrow \emptyset$ {Initialized list of state-action sequence for sample f_i } ;
      $t = 0$;
      while $t < T$ **do**
        // Choose action a_t for s_t ;
        p = random() ;
        if $p < \epsilon$ **then**
          $a_t = random\ action$;
        else
          $a_t = argmaxp_\pi(s_t)$;
        $\tau \leftarrow \{(s_t, a_t)\}$;
        $t = t + 1$;
      Compute the reward r for the trajectory τ through the proposed reward function;
      Update the parameters of the policy network $PNet$ (i.e., p_π) using reward r with REINFORCE algorithm;
      $j = j + 1$;

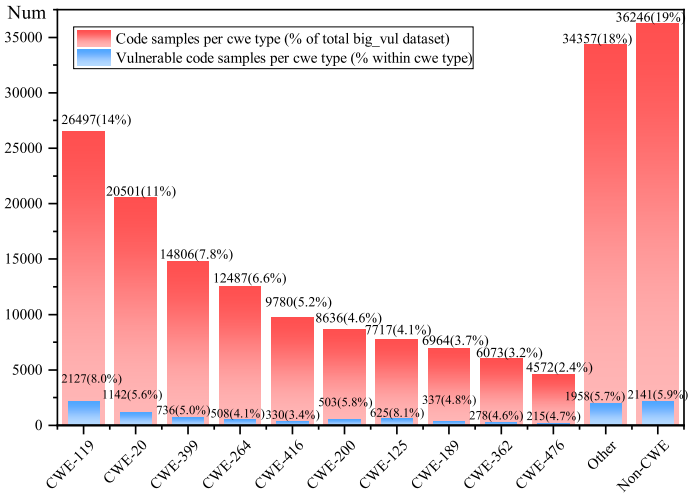

Fig. 6. Distribution of vulnerability types in the programs of the dataset. For each CWE type, the red bar represents the number of code samples and their proportion relative to the entire *big_vul* dataset. The blue bar shows the number of vulnerable code samples within each CWE type and their proportion relative to all code samples of that specific CWE type. The category *Non-CWE* includes code samples that are not associated with any specific CWE type.

4.3 Advantages Over Other Methods

Unlike the state-of-the-art supervised methods that treat each statement independently when making decisions, we argue that there are at least three advantages to the proposed approach. First, our method is capable of capturing high-order correlations within the fine-grained label (i.e., co-existence relations between vulnerable statements in a vulnerability) from training data through the RL mechanism, which is intuitively difficult to achieve with state-of-the-art DL-based methods. Second, the designed reward not only comprehensively evaluates the accuracy of the predicted vulnerable statements, but also enables the proposed method to fully utilize fine-grained labels to guide model training. Third, our method is capable of quantifying how likely a subset of statements is to cause vulnerabilities by observing their reward calculated by formula 3, and thus provides a clear "signal" for learning effective models.

5 Evaluation Setting

5.1 Research Questions

In this section, we pose the following research questions (RQs) to evaluate the effectiveness of our method[3].

RQ1: To what extent can the proposed RLFD effectively improve the performance of fine-grained vulnerability detection?

RQ2: To what extent does the proposed RLFD outperform state-of-the-art LLM-based methods in fine-grained vulnerability detection?

RQ3: To what extent does each component of the designed reward in RLFD contribute to its overall effectiveness?

RQ4: How robust is the proposed RL-based fine-grained detection method when applied to various code representation models?

RQ5: To what extent do key parameters (exploration number and data scale) impact model performance?

RQ6: How can the proposed method be generalized for detecting vulnerabilities in real-world projects?

5.2 Dataset

We employ the *big_vul* dataset provided by Fan *et al.* [20], which is widely used in recent studies due to its large size and real-world project origins [1]. The dataset is collected from 348 large-scale open-source C/C++ projects covering the period from 2002 to 2019. It contains a total of 188,636 code functions, of which 10,900 are labeled as vulnerable and 177,736 as non-vulnerable. This dataset includes both function-level and line-level ground truths, making it suitable for our study focused on fine-grained vulnerability detection. Other well-known datasets, such as Devign [21] and Reveal [22], only offer function-level labels, limiting their applicability for methods requiring detailed line-level evaluation. Therefore, the *big_vul* dataset is the most appropriate choice for evaluating the effectiveness of our method. Table 1 presents the statistics of the *big_vul* dataset, including the number of samples per Common Weakness Enumeration (CWE) type and various metrics related to lines of code (LOC), such as mean, minimum, maximum, median, and standard deviation. Fig. 6 illustrates the distribution of vulnerability types within the dataset. As the dataset is not pre-divided into training and testing sets, we follow the practice of prior studies [1], [4] by randomly dividing it into training, evaluation, and testing sets with ratios of 80%, 10%, and 10%, respectively. Table 2 provides a summary of the number of vulnerable and non-vulnerable samples in each set.

3. All resources are available at https://github.com/YuanJiangGit/RLFD.git.

TABLE 1
Statistics of the dataset used in this paper, where # iNum represents the number of programs, and # MeaL, # MinL, # MaxL, # MedL and # StdL denote the mean, minimum, median, maximum and standard deviation of the number of lines of code, respectively.

CWE ID	# iNum	# MeaL	# MinL	# MaxL	# MedL	# StdL
CWE-119	26,497	36.8	2	3822	16	78.9
CWE-20	20,501	29.8	2	6676	13	82.4
CWE-399	14,806	25.6	2	2761	11	71.7
CWE-264	12,487	25.2	2	788	15	34.6
CWE-416	9,780	26.3	2	4679	13	70.2
CWE-200	8,636	35.3	2	1854	18	63.6
CWE-125	7,717	48.6	2	2448	21	100.8
CWE-189	6,964	33,8	2	3087	17	76.3
CWE-362	6,073	30.0	2	1288	16	48.1
CWE-476	4,572	40.4	2	3574	22	93.3
ALL	188,636	30.9	2	6820	14	76.2

TABLE 2
The number of vulnerable and non-vulnerable programs in training, evaluation, and testing datasets

Dataset	# Total	# Vul	# Non-Vul	# Vul/Total (%)
Training Set	150,908	8,736	142,172	5.79
Evaluation Set	18,864	1,109	17,755	5.88
Testing Set	18,864	1,055	17,809	5.59
Total	188,636	10,900	177,736	5.78

5.3 Labelling Code Gadgets

In the *big_vul* dataset, each code function is labeled with both function-level and line-level ground truth [20]. A code function is labeled as vulnerable if it is modified in a vulnerability-fixing commit associated with a Common Vulnerabilities and Exposures (CVE) entry. The modified lines within such commits are considered line-level labels for fine-grained vulnerability detection [1]. Functions not modified in such commits are labeled as non-vulnerable. The authors of the *big_vul* dataset have made extensive manual efforts in the data collection process to ensure the high quality of this dataset [4].

5.4 Baseline Methods

In this paper, we consider the following three types of fine-grained vulnerability detection methods as our baselines:

(1) **GNN Interpretation Methods**, including GNN-LRP, DeepLIFT [23], GradCAM [24], GNNExplainer [25], PGExplainer [26], and SubGraphX [27], utilize techniques such as decomposition, gradient analysis, and perturbation to interpret GNN outputs by identifying critical nodes, edges, or subgraphs that influence vulnerability detection [15].

(2) **Transformer-based methods**, including LineVul [1] and StagedVulBERT [3], utilize Transformer architectures to detect vulnerabilities by learning code representation and leveraging attention mechanisms or statement classification to identify and rank statements that are most likely to be vulnerable.

(3) **LLM-based methods**, such as DeepSeek-Coder-v2 [28], CodeLlama [29], StarCoder2-instruct [30], WizardCoder [31], Mistral [32], and Phi-2 [33], leverage the strong code understanding capabilities of LLMs by fine-tuning them for binary classification at the statement level, enabling the models to classify each code statement as either "vulnerable" or "not vulnerable".

5.5 Parameter configurations

We adopt the CodeBERT-HLS framework for code representation and DNet for coarse-grained detection, following previous work [3] to ensure consistency and to clearly demonstrate the improvements introduced by our RL-based method in fine-grained vulnerability detection. For more details on the CodeBERT-HLS framework and its pre-training process, please refer to the original work [3]. Our fine-grained detection model, PNet, employs a single-layer fully connected network as the classifier to calculate transition probabilities, with a hidden size of 256. PNet is trained using RL, as described in Equations (3)–(6), allowing the model to dynamically learn from the rewards provided by the environment. Note that during the fine-grained training stage, we freeze the parameters of the code representation model learned via the coarse-grained training phase. By doing so, we ensure that optimizing the fine-grained detector does not interfere with the performance of the coarse-grained detection model. Specifically, we optimize the PNet model using the Adam optimizer [34], with a learning rate of 2e-5 and no weight decay. The model is trained for 10 epochs using randomly shuffled mini-batches of size 16. In our implementation, the default parameter SN in the exploration algorithm is set to 5, determined through parameter tuning experiments shown in Section 6.5. All experiments are conducted on a machine equipped with an Intel Xeon(R) 6348 CPU, 1024 GB DDR4 RAM, and an NVIDIA A800 GPU with 80 GB of memory.

5.6 Evaluation Metrics

To evaluate the performance of our proposed method, we use the following metrics, which have been widely accepted by previous work [14], [15].

Intersection over Union (IoU) measures the precision in locating vulnerabilities [14]; $IoU = \frac{|V \cap U|}{|V \cup U|}$, where V is the predicted set of vulnerable lines of code, and U is the ground truth of vulnerable locations.

Top-*k*% Accuracy (Top-*k*% Acc) refers to the proportion of programs where at least one vulnerable line of code is successfully detected within the top-*k* recommended lines ($N_{detected}$) among all vulnerable programs (N_{total}) in the testing dataset.

$$\text{Top-}k\%\text{ Accuracy} = \frac{N_{\text{detected}}}{N_{\text{total}}} \tag{7}$$

Following [1], [15], we employ two criteria for a prediction hit: identifying one of the vulnerable statements within the top 5% of the predicted list of lines (the Top-5% accuracy) and identifying one of the vulnerable statements within the top 10% of the predicted list of lines (the Top-10% accuracy).

In our evaluation, fine-grained detection is performed only on true positive (TP) samples, referring to those correctly identified as vulnerable by the coarse-grained detector. We exclude false positives (FP), as they contain no vulnerable statements and thus yield zero fine-grained scores (e.g., IoU, Top-5% Acc, Top-10% Acc). Including them would

unfairly penalize fine-grained performance, particularly for models with low precision at the coarse-grained stage. This evaluation setting focuses on assessing the fine-grained detection capability of each model under the assumption that all input samples have been correctly identified by the coarse-grained detector.

In addition, to enable a more comprehensive analysis, we assess state-of-the-art fine-grained vulnerability detection methods under different evaluation scenarios and investigate how coarse-grained results influence fine-grained performance, as discussed in Section 7.1.

6 Evaluation Results

6.1 RQ1: To what extent can the proposed RLFD effectively improve the performance of fine-grained vulnerability detection?

To answer this question, we compare our approach with six GNN-based methods (i.e., GNN-LRP, DeepLIFT, GradCAM, GNNExplainer, PGExplainer, and SubGraphX) and two strong transformer-based detection baselines (i.e., StagedVulBERT and LineVul). Since these GNN-based baselines and LineVul treat fine-grained vulnerability detection as a ranking problem, we compute their IoU by selecting the top-ranked statements matching the number of ground-truth vulnerable statements and using them as predicted vulnerable lines. Table 3 shows the experimental results on the *big_vul* testing dataset.

TABLE 3
Comparison of our method with other GNN-interpretation-based and Transformer-based fine-grained vulnerability detection methods

Type	Method	IoU (%)	Top-5% Acc	Top-10% Acc
GNN interpretation-based methods	GNN-LRP	25.5	27.6	42.3
	DeepLIFT	25.6	32.3	42.7
	GradCAM	29.8	26.1	46.2
	GNNExplainer	25.1	31.8	45.9
	PGExplainer	24.2	30.1	45.6
	SubGraphX	23.1	35.4	43.9
Transformer-based methods	StagedVulBERT	67.4	65.9	67.6
	LineVul	26.7	37.5	45.2
RL-based method	**Our method**	**69.7**	**67.7**	**68.6**

Taking a closer look at the experimental results in Table 3, we can draw several interesting conclusions.

First, our method outperforms the existing detection methods, such as PGExplainer, LineVul and StagedVulBERT. Specifically, compared to the state-of-the-art transformer-based model StagedVulBERT, our method achieves a 3.4% improvement in IoU, and improvements of 2.7% and 1.5% in Top-5% Acc and Top-10% Acc, respectively. These results demonstrate the effectiveness of our method in fine-grained vulnerability detection. The enhanced performance can be attributed to our method's ability to leverage fine-grained labels to effectively capture patterns of vulnerabilities (e.g., co-existence relations between vulnerable statements) through the designed reward, which we discuss further in Section 6.3.

Second, among Transformer-based methods, we observe that StagedVulBERT improves locating precision by 128.5% in IoU compared to LineVul. This improvement is largely due to StagedVulBERT's innovative PCL model, CodeBERT-HLS, which generates more accurate statement feature representations, enhancing its suitability for fine-grained vulnerability detection. Unlike StagedVulBERT, LineVul utilizes token attention scores within the CodeBERT architecture to identify potentially vulnerable statements. However, empirical experiments in [3] have shown that attention scores may not reliably reflect the semantic importance of code tokens, leading to lower performance.

Third, we notice that GNN interpretation-based methods perform much worse than state-of-the-art Transformer-based methods in identifying vulnerable locations. This is because Transformer-based methods are better equipped to comprehend the semantic and syntactic information within code snippets, while GNN interpretation-based methods struggle to recognize these vulnerability semantics due to their simpler networks lacking sufficient parameters.

Additionally, we present the performance of our method for each of the 14 most frequent and dangerous CWE types in the *big_vul* dataset, as presented in Fig. 7. These CWEs represent the most critical vulnerabilities [1], [3]. As shown in Fig. 7, RLFD achieves an overall IoU of 69.7%, demonstrating strong performance across various CWE types. It performs particularly well on CWE-476 (Null Pointer Dereference) with an IoU of 78.2%, CWE-416 (Use After Free) with an IoU of 75.9%, and CWE-362 (Race Condition) with an IoU of 76.1%, indicating its effectiveness in capturing patterns for these vulnerabilities.

Furthermore, we compare the IoU performance of our method, RLFD, with the state-of-the-art approach, StagedVulBERT, on testing samples across 14 critical CWE types, as shown in Fig. 8. Since RLFD is designed to capture co-occurrence relationships among multiple vulnerable statements through RL, we divide the test samples for each CWE into two categories: single-line vulnerabilities and multi-line vulnerabilities. Fig. 8 (a) shows the proportion of single-line versus multi-line vulnerabilities within each CWE type, revealing the distribution of structural complexity across different categories. Fig. 8 (b), (c), and (d) present the IoU comparisons between RLFD and StagedVulBERT for each CWE category, evaluated separately on single-line vulnerabilities, multi-line vulnerabilities, and the combined set of all vulnerabilities (including both types), respectively.

As illustrated in Fig. 8 (d), RLFD outperforms StagedVulBERT on 10 out of 14 CWE categories, while StagedVulBERT performs better in 2 cases. A standard Wilcoxon Signed-Rank Test, conducted at a 0.05 significance level, confirms that the observed performance improvements of RLFD are statistically significant, as the T-statistic for Wilcoxon's test (T_{Wilcox}) falls below the critical threshold [35]. For instance, RLFD achieves a relative improvement of 19.7% over StagedVulBERT in detecting CWE-416 (use-after-free) vulnerabilities. In the multi-line vulnerability evaluation (Fig. 8 (c)), RLFD achieves the best performance in 13 out of 14 CWE categories. The only exception is CWE-77, where StagedVulBERT slightly outperforms RLFD; however, this particular result is not statistically meaningful as CWE-77 contains only two multi-line vulnerable samples. These findings further demonstrate the effectiveness of RLFD in modeling and detecting complex vulnerability patterns that involve multiple co-occurring vulnerable statements,

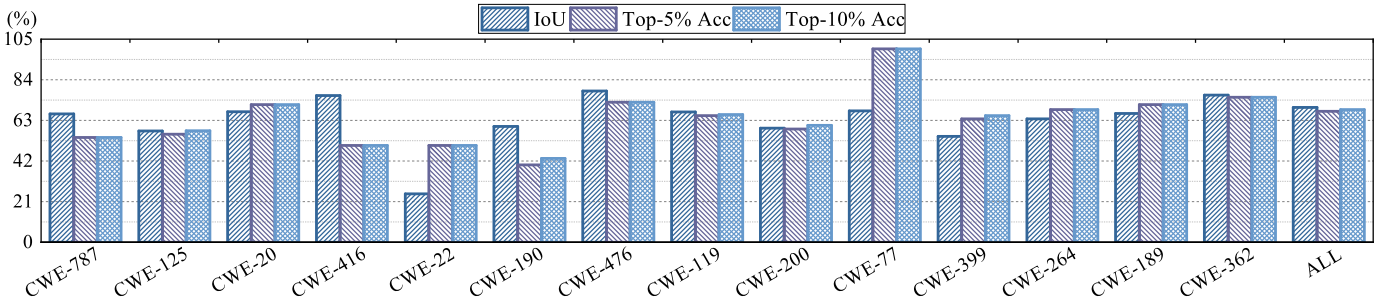

Fig. 7. Performance of our proposed RLFD method in detecting vulnerabilities across 14 of the most frequent and dangerous CWE types in the *big_vul* dataset.

thereby outperforming the state-of-the-art baseline Staged-VulBERT.

Conclusion: Through extensive experiments on the real-world dataset, we demonstrate that the proposed RL-based approach achieves an overall IoU improvement of 3.4% and up to 19.7% for specific CWEs such as CWE-416.

6.2 RQ2: To what extent does the proposed RLFD outperform state-of-the-art LLM-based methods in fine-grained vulnerability detection?

LLMs have recently demonstrated remarkable capabilities in various code-related tasks, such as code generation and test case generation, effectively addressing limitations inherent in traditional DL- and transformer-based methods. To explore whether the proposed RLFD method outperforms LLMs in fine-grained vulnerability detection, we conduct experiments to evaluate performance differences between RLFD and six state-of-the-art open-source LLMs: CodeLlama 7B, WizardCoder 7B, Mistral 7B, Phi-2 2.7B, StarCoder2-instruct 3B and DeepSeek-Coder-v2 16B on the *big_vul* dataset.

Building on previous work [36], we formalize fine-grained vulnerability detection via LLM as a binary classification problem at the statement level, where each code statement is classified as either vulnerable or not vulnerable. For example, if a function consists of five lines and an LLM model predicts line 2 as vulnerable, we represent this prediction as [0, 1, 0, 0, 0], where '1' indicates a predicted vulnerability. Based on this formalization, we use standard binary classification metrics, including precision, recall, F1-score, accuracy, and FPR, to evaluate the performance of both the baseline LLMs and our method.

In addition, since LLMs are typically designed for generative tasks, we adapt them for fine-grained vulnerability detection by using the `AutoModelForSequenceClassification` class from the Transformers library[4], which adds a classification layer to each pre-trained LLM, converting it into a sequence classifier. These models are fine-tuned on the training dataset of the *big_vul* to enable accurate classification of code statements as either "vulnerable" or "not vulnerable". This experimental setting is consistent with recent empirical research [36]. The experimental results are shown in Table 4.

4. https://pypi.org/project/transformers/, a Python library

TABLE 4
Comparison of our method with LLM-based fine-grained vulnerability detection methods

Method	IoU (%)	Top-5% Acc	Top-10% Acc
CodeLlama 7B	39.7	54.3	59.1
WizardCoder 7B	40.9	54.9	59.9
Mistral 7B	20.9	46.4	54.9
Phi-2 2.7B	38.4	54.4	58.8
StarCoder2-instruct 3B	36.5	49.3	55.1
DeepSeek-Coder-v2 16B	38.7	54.2	59.4
Our method	**69.7**	**67.7**	**68.6**

As shown in Table 4, our method RLFD outperforms the state-of-the-art LLM-based methods in fine-grained vulnerability detection across all evaluated metrics. Specifically, RLFD achieves higher IoU, Top-5% Acc, and Top-10% Acc compared to the LLMs tested. For instance, RLFD achieves an IoU of 69.7%, which is a substantial improvement over the best-performing LLM, WizardCoder 7B, which achieved an IoU of 40.9%.

These results indicate that while LLMs perform well in general code understanding, the proposed method RLFD, which leverages RL to capture complex patterns associated with code vulnerabilities, is more effective for the fine-grained detection task. These findings underscore the necessity for continued research into specialized methods tailored for vulnerability detection, as LLMs, despite their advantages in many tasks, may not be sufficient to address the complexities of fine-grained detection.

Conclusion: Although LLMs excel in many code understanding tasks, they perform worse than our proposed specialized method RLFD, which effectively captures complex vulnerability patterns through RL.

6.3 RQ3: To what extent do the designed reward and its components contribute to the effectiveness of our method?

To answer this question, we first investigate the impact of the reward on the performance of our proposed RL-based model. Then, we examine the effect of varying strengths of two components in the reward function during training on model performance.

To demonstrate whether or not an optimization objective based on our designed reward can bring advantages for constructing fine-grained detection models, we develop

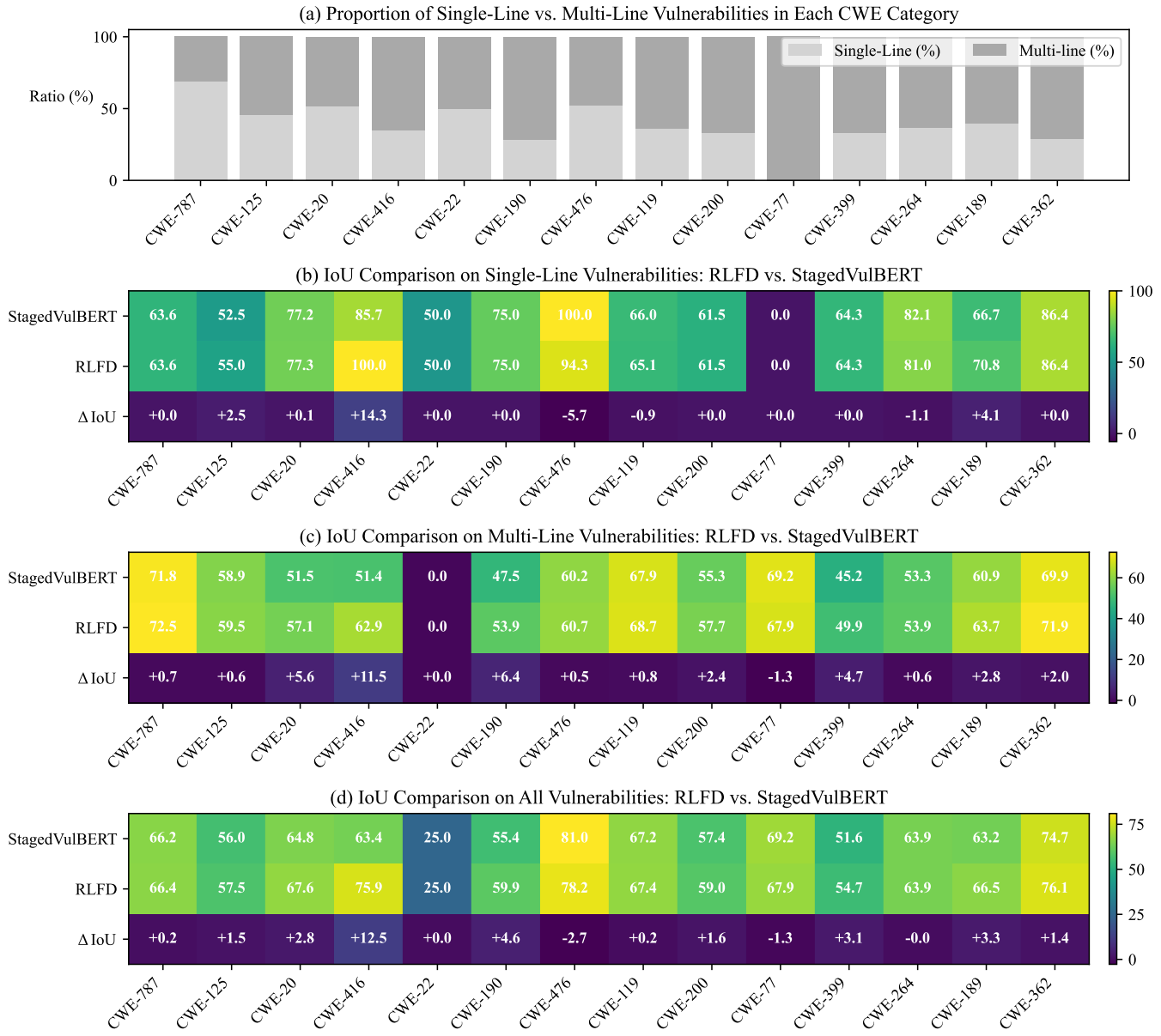

Fig. 8. Comparison of vulnerability structure (single vs. multi-line) and IoU performance between our method and the state-of-the-art baseline on the Top-14 common and high-risk CWE types

a variant (named RLFD_var) of our proposed method which only changes $R(\tau^n)\nabla log p_\pi(a_t^n|s_t^n,\theta)$ in formula 5 to $\nabla log p_\pi(\hat{a_t^n}|s_t^n,\theta)$, where $\hat{a_t^n}$ is the true label that indicates whether the state (s_t^n) is related to any vulnerabilities. The experimental results are shown in Table 5.

TABLE 5
Comparison of RLFD and its variant RLFD_var at a fine granularity on the *big_vul* testing set

Method	IoU (%)	Top-5% Acc	Top-10% Acc
RLFD_var	25.8	47.4	53.0
RLFD	**69.7**	**67.7**	**68.6**

As seen from the results, RLFD achieves a performance improvement of 170.2% compared to the variant RLFD_var, which does not include our designed reward component. The main reason for the performance difference lies in the update mechanisms. The standard REINFORCE algorithm updates the policy parameters θ in the direction $R(\tau^n)\nabla log p_\pi(a_t^n|s_t^n,\theta)$, which provides an unbiased estimate of $\nabla \mathbb{E}_{\tau\sim P_\theta(\tau)} R(\tau)$. In contrast, the variant essentially follows a supervised learning (SL) approach, as derived in [37], where the classifier parameters are updated in the direction of $\nabla p_\pi(\hat{a_t^n}|s_t^n,\theta)$, the gradient computed from the cross-entropy loss function. Obviously, the proposed RL-based approach can directly optimize the locating performance measure via the designed reward to learn the automatic "code reviewer". However, the variant introduces a bias between the learning process and the actual objective of accurately locating vulnerabilities. A more detailed comparison between methods conceptually similar to RLFD and

its variant is thoroughly discussed in [37].

In summary, since the main difference between these two models is that RLFD_var employs cross-entropy loss to learn the policy, while RLFD is based on the designed reward together with the log transition probability derived from the REINFORCE algorithm, the results demonstrate that the designed reward in formula 5 plays an important role in improving the performance of fine-grained detection.

Our overall reward function combines two components, $\frac{|V(\tau)\cap U|}{|V(\tau)\cup U|}$ and $\sum_{l\in U}\frac{1}{\text{index}(l)+1}$, through a weighted factor w. To explore how different strengths of the two terms influence model training, we adjust the value of w within the range [0, 5] on the *big_vul* testing set to analyze variations in model performance. The results are shown in Table 6.

As shown in the Table 6, when the value of w is 0, the model performs at its lowest, with a Top-5% accuracy of 66.6%. This suggests that introducing a penalty for late identification of true vulnerable statements in the prediction list effectively enhances model performance. However, as the value of w increases, the improvements in model performance become more gradual, indicating the model's robustness. Optimal performance occurs when w is set to 2, at which point the IoU and Top-5% Acc values are 69.7% and 67.7%, respectively.

TABLE 6
Comparison of RLFD with varying w values in the reward function on the *big_vul* testing set

w value in Formula 3	IoU (%)	Top-5% Acc	Top-10% Acc
0	69.1	66.6	67.5
1	69.3	67.6	68.5
2	**69.7**	**67.7**	**68.6**
3	69.4	67.8	68.5
4	69.1	67.6	68.4
5	69.6	67.4	68.4

Conclusion: The reward function is the key component of our method, ensuring strong performance in fine-grained vulnerability detection. Additionally, each reward component plays an important role in guiding the model to effectively capture complex vulnerability patterns.

6.4 RQ4: How robust is the proposed RL-based fine-grained detection method when applied to various code representation models?

Experiments in RQ1 and RQ2 demonstrated the effectiveness of the proposed RLFD framework, which leverages CodeBERT-HLS to generate fine-grained statement representations that serve as states for the policy network (see Section 4.1). In this section, we investigate whether the proposed method remains effective when built upon alternative code representation models.

To this end, we select two widely used pre-trained code language models, CodeT5 [38] and UniXcoder [39], as alternatives. Unlike CodeBERT-HLS, these models produce representations only at the function or token level and do not directly yield statement-level embeddings. To bridge this gap, we employ an "Average Token-to-Statement" approach, where the embedding of each statement is computed as the mean of its token embeddings. Once these statement-level representations are obtained, each is assigned a binary label indicating whether the statement is vulnerability-related. Consistent with the approach used in StagedVulBERT, we then optimize a statement-level classifier using binary cross-entropy loss, which serves as a baseline method under the SL setting in our experiments.

To demonstrate the effectiveness of our RL-based fine-grained detection method under the same code encoding setting, we apply the RL framework described in Section 4.1 to these statement representations and train a policy model. By comparing the results with those of the SL baselines, we assess whether our RL strategy consistently enhances fine-grained vulnerability detection performance across different code representation models.

TABLE 7
Comparison of fine-grained vulnerability detection performance across different code representation models, where (w/SL) denotes the SL-based approach and (w/RL) denotes our RL-based fine-grained detection method.

Model	IoU (%)	Top-5% Acc	Top-10% Acc
CodeT5 (w/SL)	2.1	12.9	17.2
CodeT5 (w/RL)	18.0	50.8	50.8
UniXcoder (w/SL)	2.7	32.5	33.9
UniXcoder (w/RL)	14.6	57.6	57.6
CodeBERT-HLS (w/SL)	67.4	65.9	67.6
CodeBERT-HLS (w/RL)	**69.7**	**67.7**	**68.6**

From Table 7, it is evident that across different code representation models, the proposed RL-based approach consistently outperforms baselines trained with SL. For example, with the CodeT5 model, the SL-based method achieves an IoU of 2.1, while our RL-based method reaches 18.0, representing an 8.6x improvement. Similar gains are observed in the Top-5% and Top-10% accuracy. These results confirm that our RL strategy is robust and can generalize effectively across various code representation models.

Furthermore, the results show that the CodeBERT-HLS, a model specifically designed for fine-grained vulnerability detection, achieves the best overall performance. This can be attributed to its ability to capture semantics both within and across code statements, as well as its optimization for handling longer token sequences. For more details on CodeBERT-HLS, please refer to [3].

Conclusion: The RLFD framework is flexible with respect to the choice of code representation model. Notably, the RL-based training strategy consistently improves fine-grained vulnerability detection performance and achieves the best results when combined with CodeBERT-HLS.

6.5 RQ5: To what extent do key parameters (exploration number and data scale) impact model performance?

Extensive experiments in RQ3 and RQ4 have demonstrated the effectiveness of the designed reward function and the generalizability of our method across different code representation models. In this section, we investigate two additional key factors that may impact the performance of fine-grained vulnerability detection. The first is the exploration number (i.e., *SN*) per sample, which determines how many

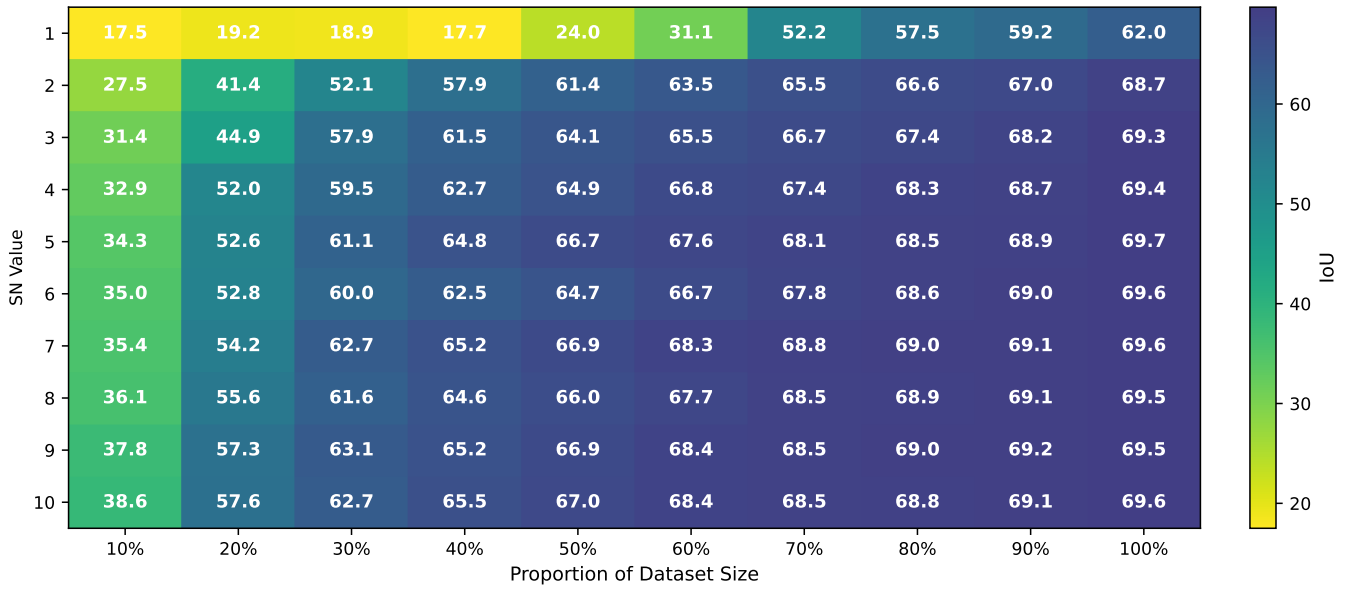

Fig. 9. The performance of the proposed RL-based fine-grained detection method under varying *SN* values and training data proportions

times PNet is allowed to explore potential vulnerability-relevant structures within a sample during its learning process, as described in Algorithm 1. The second is the scale of the training data, defined as the proportion of the *big_vul* training set used for model optimization. As widely recognized in practice and prior work, data scale plays a critical role in influencing the accuracy and generalization of DL models. We report the experimental results in terms of IoU scores for all combinations of exploration number (ranging from 1 to 10) and training data proportion (ranging from 10% to 100%). The results are summarized in Fig. 9.

As shown in Fig. 9, when the training data proportion is limited to 10%, the IoU performance is significantly lower. This is primarily due to the insufficient data for effectively training the policy network. However, under the same data constraint (10%), increasing the exploration number *SN* leads to a consistent improvement in IoU, indicating that generating more trajectories per sample enables the model to better explore and learn vulnerability-relevant structures. In contrast, when the training data proportion is large (e.g., 100%), we observe a certain performance gain as *SN* increases from 1 to 5. However, further increasing *SN* beyond this point does not lead to additional improvements, indicating that the model has already achieved sufficient exploration and performance tends to plateau under abundant data conditions.

These results demonstrate that both the exploration mechanism and training data scale have a significant impact on the performance of our RL-based method. When a large-scale dataset is available, a relatively small value of *SN* is sufficient to achieve satisfactory performance while maintaining training efficiency. Conversely, when the dataset is limited in size, increasing *SN* can help partially compensate for the lack of data by generating more trajectories per sample, thereby enhancing model effectiveness.

Conclusion: The performance of our method RLFD is influenced by the exploration number (*SN*) and the scale of training data. When using the full *big_vul* training dataset, the model achieves the best performance with *SN* set to 5.

6.6 RQ6: How can the proposed method be generalized for detecting vulnerabilities in real-world projects?

To evaluate the generalization capability of the proposed RLFD method in detecting vulnerabilities within real-world software projects, we conduct a case study on the Linux Kernel version 5.12[5], released in April 2021. This version is chosen because it was released after the collection of the *big_vul* dataset, ensuring no overlap exists between the training data and the target project. This selection allows us to rigorously assess the generalization of RLFD without the risk of data leakage.

The Linux Kernel 5.12 contains 578,953 functions, with a total of over 15 million lines of code. We apply RLFD to analyze all functions, with inference performed using a batch size of one (i.e., one function per batch). The entire detection process took approximately 560 minutes on a machine equipped with an Intel Xeon 6348 CPU and an A800 GPU with 80 GB of memory. This translates to an average of about 0.0584 seconds per function, demonstrating the model's efficiency in handling large-scale software projects.

After training RLFD using the experimental dataset described in earlier sections, we apply the trained model to the extracted functions from the Linux Kernel. The model generates predictions of potentially vulnerable statements. To verify these predictions, we conduct a manual analysis by cross-referencing each predicted vulnerable statement and its containing function against known vulnerabilities listed in the National Vulnerability Database (NVD). Only if both the predicted vulnerable statement and its corresponding function match or are closely related to a vulnerability documented in the NVD do we classify it as a confirmed real-world vulnerability.

5. https://github.com/torvalds/linux/releases/tag/v5.12

TABLE 8
Detection of vulnerable lines of code in Linux Kernel 5.12 using RFLD, where columns include CVE IDs, CWEs, Release dates, Vulnerable file, Vulnerable function, and Predicted Locations of truly Vulnerable Statements (PL of VS). The PL of VS specifies the line numbers within the corresponding function where RLFD successfully detected the actual vulnerable statements located. For example, (12) indicates that RLFD successfully detected the truly vulnerable statement at line number 12.

CVE ID	CWE	Release date	Vulnerable file	Vulnerable Function	PL of VS
CVE-2023-23000	CWE-476	03/01/2023	drivers/phy/tegra/xusb.c	tegra_xusb_find_port_node	(12)
CVE-2022-47519	CWE-787	12/18/2022	drivers/net/wireless/microchip/wilc1000/cfg80211.c	wilc_wfi_cfg_parse_ch_attr	(14, 18)
CVE-2023-2008	CWE-129	04/14/2023	drivers/dma-buf/udmabuf.c	udmabuf_vm_fault	(4)
CVE-2024-0641	CWE-667, CWE-833	01/17/2024	net/tipc/crypto.c	tipc_crypto_key_revoke	(4, 10)
CVE-2021-35039	CWE-347	07/07/2021	kernel/module.c	set_module_sig_enforced	(1)
CVE-2023-4385	CWE-476	08/16/2023	fs/jfs/jfs_dmap.c	dbFree	(55)
CVE-2023-2019	CWE-911	04/24/2023	drivers/net/netdevsim/fib.c	nsim_fib_event_schedule_work	(13)
CVE-2022-2959	CWE-362, CWE-667	08/25/2022	fs/pipe.c	pipe_resize_ring	(10, 11, 12, 13, 14, 15)
CVE-2021-32078	CWE-125	06/07/2021	arch/arm/mach-footbridge/personal-pci.c	personal_pci_init	(1)
CVE-2021-41864	CWE-190	10/02/2021	kernel/bpf/stackmap.c	prealloc_elems_and_freelist	(2)
CVE-2022-1975	CWE-248	08/31/2022	net/nfc/netlink.c	nfc_genl_fw_download_done	(5, 17)
CVE-2023-3863	CWE-416	07/24/2023	net/nfc/netlink.c	nfc_genl_llc_get_params	(23)
CVE-2023-2513	CWE-416	05/08/2023	fs/ext4/xattr.c	ext4_xattr_ibody_set	(7)
CVE-2021-38201	CWE-119	08/08/2021	net/sunrpc/xdr.c	xdr_set_page_base	(9, 10, 11, 12)
CVE-2022-1353	CWE-200	04/29/2022	net/key/af_key.c	pfkey_register	(12)
CVE-2022-40307	CWE-416	09/09/2022	drivers/firmware/efi/capsule-loader.c	efi_capsule_flush	(1, 2, 3, 4, 5, 6, 7, 8, 9)
CVE-2022-2318	CWE-416	07/06/2022	net/rose/rose_timer.c	rose_stop_idletimer	(2)
CVE-2022-3586	CWE-416	10/19/2022	net/sched/sch_sfb.c	sfb_enqueue	(109)
CVE-2022-39842	CWE-190	09/05/2022	drivers/video/fbdev/pxa3xx-gcu.c	ssize_tpxa3xx_gcu_write	(8)
CVE-2022-1043	CWE-416	08/29/2022	fs/io_uring.c	io_register_personality	(8)
CVE-2023-2007	CWE-367, CWE-667	04/24/2023	drivers/scsi/dpt_i2o.c	adpt_init	(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22)
CVE-2022-42895	CWE-824	11/23/2022	net/bluetooth/l2cap_core.c	l2cap_parse_conf_req	(145)
CVE-2021-38204	CWE-416	08/08/2021	.drivers/usb/host/max3421-hcd.c	voidmax3421_set_address	(2, 5, 6, 8, 9, 10, 11, 12, 13, 15, 16, 17, 19, 20, 21, 27, 34)
CVE-2021-38206	CWE-476	08/08/2021	.net/mac80211/tx.c	ieee80211_parse_tx_radiotap	(40)
CVE-2023-32233	CWE-416	05/08/2023	.net/netfilter/nft_objref.c	nft_objref_map_activate	(4)
CVE-2022-2639	CWE-191, CWE-192, CWE-681, CWE-787	09/01/2022	net/openvswitch/flow_netlink.c	reserve_sfa_size	(12)
CVE-2021-38160	CWE-120	08/07/2021	drivers/char/virtio_console.c	get_inbuf	(8)
CVE-2022-1734	CWE-416	05/18/2022	drivers/nfc/nfcmrvl/main.c	nfcmrvl_nci_unregister_dev	(8)
CVE-2022-1974	CWE-367, CWE-416	08/31/2022	net/nfc/core.c	nfc_start_poll	(14)
CVE-2022-30594	CWE-863	05/12/2022	kernel/ptrace.c	ptrace_setoptions	(8)
CVE-2021-43267	CWE-20	11/02/2021	net/tipc/crypto.c	tipc_crypto_key_rcv	(11, 17, 25, 26, 27, 28, 29, 35)

Our findings indicate that RLFD successfully identifies 31 known vulnerabilities within the Linux Kernel 5.12 at the statement level. These vulnerabilities have been manually confirmed to match CVE entries in the NVD. In comparison, only 13 of these 31 vulnerabilities are detected at the statement level by the state-of-the-art method StagedVulBERT. Table 8 presents detailed information on the vulnerabilities detected by RLFD, including the CVE ID, CWE type, affected files, vulnerable functions, and the predicted locations of truly vulnerable statements (PL of VS).

These results demonstrate the effectiveness of RLFD in generalizing to unseen, real-world projects. The model is able to detect vulnerabilities across various CWE types, including but not limited to buffer overflows (CWE-119), null pointer dereferences (CWE-476), and use-after-free errors (CWE-416), indicating its ability to handle different categories of vulnerabilities. In addition, RLFD predicts another 184 functions as vulnerable; however, these do not have corresponding entries in the NVD, making it difficult to automatically confirm whether they actually contain vulnerabilities. To address this, we conduct a manual inspection of the 184 predicted cases, and no clear evidence of vulnerabilities is found. Therefore, reducing the false positives will be an important direction for our future work.

Conclusion: RLFD successfully discovered 31 real-world vulnerabilities at the statement level by scanning Linux Kernel 5.12, whereas StagedVulBERT detected only 41.9% of them. This demonstrates RLFD's effectiveness in fine-grained vulnerability detection in large real-world software, offering valuable insights for improving software security.

7 Discussion

In this section, we further analyze the performance advantages of our method from different evaluation perspectives by comparing it against several strong baselines, including WizardCoder, LineVul, and StagedVulBERT, as evidenced by the results in RQ1 and RQ2.

7.1 Investigating the Impact of Coarse-grained Performance on Fine-grained Vulnerability Detection

In our primary evaluation, fine-grained vulnerability detection is applied only to samples correctly identified as vulnerable during the coarse-grained stage (i.e., true positives, TP).

False positives (FP), which are misclassified non-vulnerable samples, are excluded from the fine-grained evaluation, as they contain no true vulnerable statements and always yield zero scores on fine-grained metrics such as IoU, Top-5% Acc, and Top-10% Acc. Including these FP samples would unfairly degrade the measured fine-grained performance, since a high false positive rate primarily reflects lower precision in the coarse-grained stage rather than limitations in fine-grained detection. By restricting the evaluation to TP samples, we ensure a level playing field across all methods.

Even so, to provide a more comprehensive analysis, we further consider two additional evaluation settings. First, we evaluate performance on all samples predicted as vulnerable by the coarse-grained detector (i.e., TP+FP), which represents a fine-grained evaluation setting influenced by coarse-grained detection performance. Second, we assume perfect coarse-grained detection and evaluate on all truly vulnerable samples (i.e., TP+ false negatives, FN), aiming to isolate the intrinsic fine-grained capability of each method. Table 9 presents the results under these three evaluation settings. To assess the potential influence of coarse-grained performance on fine-grained results, we also report the coarse-grained detection metrics for all models in Table 10.

TABLE 9
Fine-grained vulnerability detection results of our method and state-of-the-art baselines under three evaluation settings

Method	IoU (%)	Top-5% Acc	Top-10% Acc
Results on True Positive (TP) Examples			
WizardCoder	40.9	54.9	59.9
LineVul	26.7	37.5	45.2
StagedVulBERT	67.4	65.9	67.6
RLFD	**69.7**	**67.7**	**68.6**
Results on All Predicted Vulnerable (TP + FP) Examples			
WizardCoder	23.5	31.5	34.4
LineVul	25.2	35.3	42.6
StagedVulBERT	63.5	62.1	63.7
RLFD	**65.7**	**63.8**	**64.7**
Results on All Positive (TP + FN) Examples			
WizardCoder	32.8	53.2	51.5
LineVul	21.4	35.1	42.2
StagedVulBERT	63.2	63.9	65.5
RLFD	**64.8**	**65.9**	**66.7**

TABLE 10
Coarse-grained vulnerability detection results of our method and state-of-the-art baselines

Method	F1-score	Recall	Precision	Accuracy
WizardCoder	69.5	88.2	57.4	95.5
LineVul	85.7	78.6	94.2	98.5
StagedVulBERT/RLFD	**92.3**	**90.3**	**94.3**	**99.2**

As shown in Table 9, our RLFD method consistently achieves superior fine-grained vulnerability detection performance in all evaluation settings, demonstrating its robustness and effectiveness. Moreover, since both StagedVulBERT and our RLFD framework share the same coarse-grained detection architecture (as shown in Fig. 1), their coarse-grained performance is comparable, and differences in fine-grained detection performance can be attributed solely to their respective fine-grained mechanisms. This design ensures a fair assessment of our RL-based approach in enhancing fine-grained vulnerability detection. Finally, a comparison between Tables 9 and 10 reveals that when the coarse-grained detector performs well (e.g., RLFD), the variation in fine-grained metrics across different evaluation settings remains minimal due to low FP and FN. In contrast, a less effective coarse-grained stage leads to larger fluctuations in fine-grained performance (e.g., WizardCoder).

7.2 Investigating the Capability of RLFD in Detecting Complete Vulnerability Patterns

In our previous evaluations, we employ IoU, Top-5% Accuracy, and Top-10% Accuracy to assess the effectiveness of our approach. IoU primarily measures the overlap between predicted and ground-truth vulnerable statements, while Top-5% and Top-10% Accuracy treat a detection as successful if at least one vulnerable statement appears within the top-5% or top-10% ranked statements. However, in certain practical scenarios, it is often desirable to identify all vulnerable lines within a sample comprehensively.

To better reflect this requirement, we introduce stricter evaluation metrics: Strict Top-5% Accuracy (STop-5% Acc) and Strict Top-10% Accuracy (STop-10% Acc). These metrics consider a detection successful only if all ground-truth vulnerable lines are ranked within the top-5% or top-10% of statements, respectively. Table 11 presents a comparison between our method and state-of-the-art fine-grained vulnerability detection approaches under these stricter criteria.

TABLE 11
Comparison of RLFD and state-of-the-art fine-grained vulnerability detection baselines under strict evaluation metrics.

Method	**Strict Top-5 Acc** (%)	**Strict Top-10 Acc** (%)
WizardCoder	44.9	53.5
LineVul	28.2	42.4
StagedVulBERT	50.1	57.8
RLFD	**55.7**	**59.9**

As shown in Table 11, even under these more stringent evaluation metrics, our RLFD method continues to demonstrate promising performance and outperforms all baseline approaches. Specifically, RLFD reaches 55.7% in Strict Top-5% Accuracy, representing an improvement of 11.2% over StagedVulBERT (50.1%). Moreover, compared to the results reported in Table 3, RLFD exhibits a more pronounced relative improvement under the strict evaluation setting than under the standard Top-5% Accuracy metric (where the improvement is only 2.7%). This further indicates that RLFD effectively leverages reinforcement learning to guide the model toward learning the co-existence relations among vulnerable statements, thereby enhancing its capability to capture complex vulnerability patterns and enabling more comprehensive detection of all vulnerable lines.

7.3 Investigating the Generalizability of RLFD to Other Programming Languages

In RQ1–RQ5, we conduct experiments on the C/C++ *big_vul* dataset, demonstrating the effectiveness of our proposed method. This section further evaluates whether the proposed RLFD framework can be generalized effectively to other programming languages, particularly Java, for fine-grained vulnerability detection.

As existing Java vulnerability detection benchmarks generally lack large-scale code samples with fine-grained labels [40], we construct a new dataset[6] by following the *big_vul* construction procedure, leveraging the labeled vulnerable and non-vulnerable commits provided in [41], [42]. Table 12 presents the statistics of the dataset we successfully collected, in which each sample is assigned to the training, validation, or testing set following the same split strategy as in prior work [41].

TABLE 12
Statistics of the dataset constructed for fine-grained Java vulnerability detection

Dataset	# Total	# Vul	# Non-Vul	# Vul/Total (%)
Training	12,655	1,146	11,509	9.1
Validation	1,422	119	1,303	8.4
Testing	4,604	415	4,189	9.0

To ensure a fair comparison, we retrain RLFD and three competitive baselines, WizardCoder, LineVul, and StagedVulBERT, on the constructed Java training set and evaluate them using the same testing set. Table 13 reports the fine-grained detection performance of each method, measured in terms of IoU, Top-5% Acc, and Top-10% Acc.

TABLE 13
Comparison of fine-grained vulnerability detection performance between RLFD and baselines on the Java dataset

Method	**IoU (%)**	**Top-5% Acc**	**Top-10% Acc**
WizardCoder	17.2	34.7	39.0
LineVul	8.2	7.8	9.8
StagedVulBERT	30.1	50.9	61.0
RLFD	**31.8**	**64.0**	**69.8**

From Table 13, we observe that RLFD achieves the best overall performance with an IoU of 31.8%, a Top-5% Acc of 64.0%, and a Top-10% Acc of 69.8%. These results represent relative improvements of 5.6% in IoU, 25.7% in Top-5% Acc, and 14.4% in Top-10% Acc compared to StagedVulBERT, the strongest baseline. In contrast, WizardCoder exhibits substantially lower performance, further confirming that specialized fine-grained vulnerability detection frameworks such as RLFD and StagedVulBERT considerably outperform general-purpose code language models on this task. These findings underscore the generalizability and robustness of our RL-based fine-grained detection framework across different programming languages.

6. https://github.com/YuanJiangGit/Java_vul_dataset.git

7.4 Investigating the Generalizability of RLFD to Cross-Function Vulnerability Detection

The *big_vul* dataset, widely adopted in recent studies due to its large scale and origin from real-world software projects, is used in our experiments to evaluate the effectiveness of the proposed method. However, since each sample in *big_vul* is limited to a single function, it does not capture vulnerabilities that span across function boundaries.

To investigate whether our method can generalize to detect cross-function vulnerabilities, we conduct additional experiments on a dataset released by [14], which consists of 14,511 programs. In this dataset, code snippets are constructed by applying program slicing techniques to extract semantically related statements from multiple functions, thereby capturing vulnerability patterns that span across function boundaries. Each snippet is then annotated at the statement level to indicate which specific lines contribute to the vulnerability. The experimental setting follows the same configuration described in Section 5.5, and the results of our method and baseline approaches are presented in Table 14.

TABLE 14
Comparison of fine-grained vulnerability detection performance between RLFD and baselines on the cross-function vulnerability dataset.

Method	**IoU (%)**	**Top-5% Acc**	**Top-10% Acc**
WizardCoder	72.4	86.3	88.2
LineVul	18.9	21.0	25.0
StagedVulBERT	80.5	94.6	95.5
RLFD	**80.9**	**97.3**	**98.3**

As shown in Table 14, RLFD achieves the highest performance, with a Top-10% Accuracy of 98.3%, representing a 2.93% relative improvement over the state-of-the-art method StagedVulBERT (95.5%). This improvement highlights the advantage of our RL-based approach in capturing co-occurrence patterns of vulnerable statements across function boundaries. These findings further confirm the generalizability of RLFD to more complex vulnerability detection scenarios involving cross-function code contexts.

7.5 Investigating Cases Where the RLFD Model Succeeded and Failed to Detect Vulnerabilities

As shown in our RQ1 results, RLFD achieves high IoU scores in detecting vulnerabilities, demonstrating its effectiveness. For example, in the code snippet (Fig. 10) from the Neomutt project [7] associated with CVE-2018-14359, our method RLFD accurately identifies the vulnerable lines, achieving an IoU of 100%. This means our model can successfully detect vulnerabilities like buffer overflows caused by incorrect memory allocation calculations.

However, there are still instances where our method does not achieve perfect detection. For example, consider the code (Fig. 11) from the Leptonica project [8] associated with CVE-2018-7186. While the RLFD model correctly predicts the vulnerable line involving the *fscanf* function that can cause a buffer overflow, the IoU score is only 50%. This

7. https://github.com/neomutt/neomutt
8. https://github.com/DanBloomberg/leptonica

```
1  static char *rfc2047_decode_word(const char *s, size_t len, enum
   ContentEncoding enc)
2  {
3    const char *it = s;
4    const char *end = s + len;
5    if (enc == ENCQUOTEDPRINTABLE)
6    {
...    ...
27   }
28   else if (enc == ENCBASE64)
29   {
30     char *out = mutt_mem_malloc(3 * len / 4 + 1);
31     int dlen = mutt_b64_decode(out, it);
32     if (dlen == -1)
33     {
34       FREE(&out);
35       return NULL;
36     }
37     out[dlen] = '\0';
38     return out;
39   }
40   assert(0); /* The enc parameter has an invalid value */
41   return NULL;
42 }
```

The statements explicitly labeled as vulnerable in the big_vul dataset

Fig. 10. A real-world vulnerability (i.e., CVE-2018-14359) in Neomutt, causing buffer overflow due to incorrect memory allocation, was successfully detected by our proposed method

discrepancy arises because the ground truth labels in the *big_vul* dataset include an additional line, *char typestr[128];*, which was modified in the commit to *char typestr[128]; /* hardcoded below in fscanf */* but is not directly related to the vulnerability. Our model cannot detect such unrelated modifications made by developers during code revisions, leading to a lower IoU score.

```
1  PTA * ptaReadStream(FILE  *fp)
2  {
3      char        typestr[128];
4      l_int32     i, n, ix, iy, type, version;
5      l_float32   x, y;
6      PTA        *pta;
7      PROCNAME("ptaReadStream");
8      if (!fp)
9          return (PTA *)ERROR_PTR("stream not defined", procName, NULL);
10     if (fscanf(fp, "\n Pta Version %d\n", &version) != 1)
11         return (PTA *)ERROR_PTR("not a pta file", procName, NULL);
12     if (version != PTA_VERSION_NUMBER)
13         return (PTA *)ERROR_PTR("invalid pta version", procName, NULL);
14     if (fscanf(fp, " Number of pts = %d; format = %s\n", &n, typestr) != 2)
15         return (PTA *)ERROR_PTR("not a pta file", procName, NULL);
16     if (!strcmp(typestr, "float"))
17         type = 0;
18     else  /* typestr is "integer" */
19         type = 1;
...    ...
40}
```

The statements explicitly labeled as vulnerable in the big_vul dataset

Fig. 11. A real-world vulnerability (i.e., CVE-2018-7186) in Leptonica, leading to buffer overflow due to improper use of *fscanf*, was not fully detected by our proposed method

This limitation highlights an issue with the dataset labeling rather than with the model itself. Developers often make incidental changes to unrelated code when fixing vulnerabilities, and these lines are included in the ground truth labels. As a result, the model may receive lower evaluation scores even when it correctly identifies the actual vulnerable code. We further discuss the impact of dataset quality on model performance in Section 7.6.

7.6 Investigating the Impact of Big_vul Dataset Quality on Model Performance

Big_vul is a widely used large-scale vulnerability dataset with fine-grained labels, making it well-suited for benchmarking our method against baseline approaches. However, its ground-truth annotations are derived from bug-fix commits, which may introduce a certain degree of label noise, as also highlighted in our case study in Section 7.5. This challenge has been widely acknowledged in the field, and to date, no widely accepted alternative labeling methodology exists for fine-grained vulnerability detection [43].

To assess the reliability of existing labels, we conduct a manual verification study involving five computer science students and two faculty advisors. Each annotator is tasked with determining whether a given labeled statement in *big_vul* is related to a vulnerability and providing a brief justification for their decision. Based on their feedback, we categorize the existing labeled statements in *big_vul* into three confidence levels: (1) definitely vulnerability-related, (2) possibly vulnerability-related, and (3) clearly unrelated. Following this protocol, we systematically review and re-annotate all samples with existing fine-grained labels in the *big_vul* dataset[9]. As noted by [43], software vulnerability data is inherently scarce, and removing potentially noisy samples could reduce the dataset to a size insufficient for practical training and evaluation. Therefore, in this section, we consider statements manually labeled with the first two confidence levels as valid ground-truth vulnerability statements for downstream analysis. Table 15 shows the distribution of the ten most common CWE types among the manually validated samples. As observed, the number of confirmed vulnerable statements decreases by approximately 40–70% compared to the original dataset, which is consistent with, and numerically falls within, the 20–71% reduction range reported in [43].

TABLE 15
Vulnerability statistics before and after manual annotation on the *big_vul* dataset across ten common CWE types. **Orig #**, **Orig Avg**, **Manu #**, **Manu Avg**, and **Drop %** denote the original sample count, original average number of vulnerable statements per sample, post-annotation sample count, post-annotation average, and the relative reduction in average statement count, respectively.

CWE ID	Orig #	Orig Avg	Manu #	Manu Avg	Drop %
CWE-119	1619	8.26	1228	2.65	67.9
CWE-20	871	4.71	730	2.32	50.7
CWE-399	581	5.67	466	2.35	58.6
CWE-264	376	4.89	306	2.34	52.2
CWE-416	223	6.67	189	2.21	66.9
CWE-200	340	4.56	291	2.28	50.0
CWE-125	443	5.56	376	2.68	51.8
CWE-189	259	5.83	215	2.19	62.4
CWE-362	203	5.71	173	2.80	50.9
CWE-476	151	3.85	132	2.20	42.9

Furthermore, we conduct additional experiments on the manually validated *big_vul* dataset to evaluate the effectiveness of our method and the baselines. The results are summarized in Table 16. As shown, our method consistently outperforms all baseline approaches across all three evaluation metrics, which is in line with the results observed on the original dataset. Interestingly, however, both the performance of our method and the baselines decline to some extent on the manually annotated version compared with the original dataset (Tables 3 and 4). This observation suggests that real and complex vulnerability patterns are inherently more difficult to detect than those found in noisily labeled data. Therefore, to more accurately assess the true effectiveness of detection models, we recommend conducting experiments on the original *big_vul*, which remains

9. The *big_vul* dataset with manually verified fine-grained labels has been released at https://github.com/YuanJiangGit/Big_vul_clean.git

a widely used benchmark for comparing relative model performance, while complementing them with evaluations on high-quality datasets with verified ground-truth labels.

TABLE 16
Performance comparison of fine-grained vulnerability detection on the manually annotated *big_vul* dataset.

Method	IoU (%)	Top-5% Acc	Top-10% Acc
WizardCoder	27.5	37.7	40.5
LineVul	21.2	29.9	39.6
StagedVulBERT	43.2	50.4	54.2
RLFD	**44.2**	**52.8**	**55.8**

7.7 Investigating the Time Efficiency of RLFD During Training and Inference

Table 17 presents runtime comparisons among four fine-grained vulnerability detection models, including WizardCoder, LineVul, StagedVulBERT, and our proposed RLFD. As all these methods are based on the Transformer architecture, a direct comparison of their training and inference efficiencies is appropriate. We observe that RLFD incurs a longer training time compared to baseline methods, mainly due to the additional computational cost of exploration during the reinforcement learning process [44]. Even though we implement concurrent trajectory generation to improve efficiency, RLFD still requires more training time than StagedVulBERT. It is worth noting that model training is a one-time, fully offline process, and the increased training time is generally acceptable in most practical scenarios.

During inference, RLFD achieves higher efficiency than LineVul, though it is slightly less efficient than StagedVulBERT. Given that RLFD shares the same network architecture with StagedVulBERT, the additional inference overhead is likely due to its sequential decision-making protocol. LineVul, in contrast, incurs additional computational overhead by computing attention scores for each statement, resulting in higher inference complexity than the direct probability prediction strategy employed by RLFD and other baselines. Overall, inference-time differences are negligible, with all methods completing predictions within 0.1 seconds per sample.

Our extensive experiments demonstrate the suitability of RL for fine-grained vulnerability detection. Nevertheless, further reducing both training and testing times remains an important direction for future research.

TABLE 17
Running time comparison between our method and baselines on the fine-grained vulnerability detection task (Training Time in Minutes; Testing time and Average Testing Time per function in Seconds)

Methods	Training Time (Total)	Testing Time (Total)	Average Testing Time (Per function)
WizardCoder	364m	45s	0.0571s
LineVul	554m	68s	0.0863s
StagedVulBERT	506m	37s	0.0469s
RLFD	571m	48s	0.0609s

8 Limitations and Future work

While our proposed method demonstrates strong performance in fine-grained vulnerability detection, several limitations provide opportunities for future research.

First, RLFD relies on the PCL model CodeBERT-HLS, which has demonstrated strong capability in generating accurate statement-level embeddings [3]. While we explored alternative PCL models such as CodeT5 and UniXcoder in Section 6.4, CodeBERT-HLS consistently achieved superior performance. Future work will explore further optimization of specialized PCL models tailored for fine-grained vulnerability detection. Second, RLFD incurs higher training and inference time compared to baseline methods due to the property of RL. Reducing computational cost while maintaining model effectiveness is a promising direction for future improvement. Third, since each code sample in *big_vul* primarily contains a single vulnerability, our current evaluation confirms the effectiveness of RLFD in detecting individual vulnerabilities at the statement level. Future work will investigate its performance in more complex scenarios involving multiple vulnerabilities within a single function. Finally, RLFD has been primarily evaluated on datasets containing C/C++ code. Although we have demonstrated its applicability to other languages, such as Java, further work is needed to enhance its generalizability across a broader range of programming languages.

9 Related work

Vulnerability detection is a critical task in software engineering, aimed at identifying potentially dangerous code that could be exploited by adversaries, thereby ensuring the reliability and security of software systems. With the development of DL, various data-driven approaches have garnered significant attention. Notable methods such as Vulpecker [45], SySeVR [2], Devign [21], Reveal [22], IVDetect [46], AMPLE [47], and SVulD [48] have made substantial progress in detecting vulnerabilities at the function or slice level. However, these methods are often limited by their coarse granularity, which reduces their effectiveness in pinpointing vulnerabilities at the statement level, a crucial step for enabling effective vulnerability discovery.

Recent advances have focused on fine-grained vulnerability detection, which seeks to detect vulnerabilities at a more granular level, specifically at the code statement level. Approaches in this domain can be broadly classified into three categories: (1) interpretation-based methods, (2) attention-based methods, and (3) statement-level classification methods.

Interpretation-based methods employ explainable artificial intelligence techniques to identify the most critical parts of the input code that influence vulnerability detection outputs. For instance, Hu *et al.* [15] apply the methods introduced by Funke *et al.* [49] and Luo *et al.* [26] for vulnerability detection by modifying input code graphs to observe the impact of nodes and edges on detection results. Similarly, Ying *et al.* [25] propose GNNExplainer, a model that identifies the subgraph structures most responsible for a prediction, providing insights into model behavior. Other works, such as Schnake *et al.* [50] and Schwarzenberg *et al.* [51], apply layer-wise relevance propagation (LRP) and Shapley values

to decompose a model's predictions into contributions from individual input features. These methods improve model transparency, although they typically introduce significant computational overhead. Moreover, surrogate models have been explored to approximate the behavior of more complex models, as demonstrated by Huang *et al.* [52] and Vu *et al.* [53], enabling a clearer understanding of the underlying prediction mechanisms.

Attention-based methods leverage the attention mechanism to focus on the most relevant portions of the input code during vulnerability detection. These methods do not require external interpretability tools, as the attention values are directly available from the model itself. For example, LineVul [1] utilizes attention scores extracted from a fine-tuned CodeBERT model to identify vulnerable statements. By utilizing the model's internal attention weights, these methods provide a more efficient approach to statement-level vulnerability detection, while also reducing the reliance on post-hoc interpretability models.

Statement-level classification methods aim to directly model and predict vulnerabilities at the code statement level, providing the most fine-grained detection available. These methods utilize code representation models to learn feature vectors of individual code statements and classify them as vulnerable or non-vulnerable. For example, LineVD [4] first learns statement representations via GAT and then uses them as statement features to perform statement-level detection [4]. StagedVulBERT [3], a more recent method, focuses on learning accurate statement representations through the novel PCL model, CodeBERT-HLS, and then improves fine-grained vulnerability detection via supervised learning.

The application of LLMs in software engineering tasks, including vulnerability detection, has gained increasing attention. However, empirical evidence suggests that while LLMs, such as ChatGPT, excel in general-purpose code generation and understanding tasks, their performance in fine-grained vulnerability detection remains limited [54]. Studies have shown that LLMs often underperform compared to specialized transformer-based models designed for vulnerability detection [36]. Our experimental results in RQ2 also corroborate this observation, demonstrating that LLMs fail to match the IoU and Top-k% accuracy of task-specific models in detecting vulnerabilities at the statement level. This highlights the need for further research into optimizing LLMs for highly specialized tasks, such as fine-grained vulnerability detection.

In this work, we define fine-grained vulnerability detection as a sequential decision-making problem and apply RL to automatically learn vulnerability-relevant patterns from code. Although RL has been explored in prior work [55], that work focuses on coarse-grained, function-level classification, which is fundamentally different from our approach in both task definition and learning objective. To the best of our knowledge, this is the first RL-based framework designed for fine-grained vulnerability detection, and it achieves promising results in accurately identifying vulnerable statements.

10 Conclusion

In this paper, we introduce *RLFD*, a reinforcement learning-based method aimed at enhancing the performance of fine-grained vulnerability detection. We first leverage CodeBERT-HLS to generate accurate representations of programs and statements. Building upon these representations, we develop a fine-grained vulnerability detection model that employs reinforcement learning to capture complex vulnerability patterns. To achieve high performance, we carefully design a novel reward function that aligns with fine-grained evaluation metrics, guiding the model to focus on the co-existence relations among statements from a global perspective. We evaluate our method on the *big_vul* dataset provided in previous work [20]. The results for fine-grained vulnerability detection demonstrate that our approach is clearly superior to the state-of-the-art approaches.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Grant Nos.62302125 and 62272132), the Heilongjiang Postdoctoral Fund (Grant No.LBH-Z23019), and the Key technical projects of Shen-Zhen (Grant No.JSGG2021110892802003).

References

[1] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 608–620.

[2] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[3] Y. Jiang, Y. Zhang, X. Su, C. Treude, and T. Wang, "Stagedvulbert: Multi-granular vulnerability detection with a novel pre-trained code model," *IEEE Transactions on Software Engineering*, pp. 1–18, 2024.

[4] D. Hin, A. Kan, H. Chen, and M. A. Babar, "Linevd: Statement-level vulnerability detection using graph neural networks," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 596–607.

[5] H. Guo, "Generating text with deep reinforcement learning," *arXiv preprint arXiv:1510.09202*, 2015.

[6] L.-J. Lin and T. M. Mitchell, "Reinforcement learning with hidden states," *From animals to animats*, vol. 2, pp. 271–280, 1993.

[7] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, 2000, pp. 1057–1063.

[8] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[9] M. Fagan, "Design and code inspections to reduce errors in program development," in *Software pioneers*. Springer, 2002, pp. 575–607.

[10] Y. Jiang, X. Su, C. Treude, and T. Wang, "Hierarchical semantic-aware neural code representation," *Journal of Systems and Software*, vol. 191, p. 111355, 2022.

[11] N. Desai and A. Banerjee, "Deep reinforcement learning to play space invaders," 2017.

[12] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[13] T. Zhang, M. Huang, and L. Zhao, "Learning structured representation for text classification via reinforcement learning," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[14] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, "Vuldeelocator: a deep learning-based fine-grained vulnerability detector," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[15] Y. Hu, S. Wang, W. Li, J. Peng, Y. Wu, D. Zou, and H. Jin, "Interpreters for gnn-based vulnerability detection: Are we there yet?" in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1407–1419.

[16] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[17] R. S. Sutton, A. G. Barto *et al.*, *Introduction to reinforcement learning*. MIT press Cambridge, 1998, vol. 135.

[18] D. P. Bertsekas and J. N. Tsitsiklis, "Neuro-dynamic programming: an overview," in *Proceedings of 1995 34th IEEE conference on decision and control*, vol. 1. IEEE, 1995, pp. 560–564.

[19] F. Henkel, "A regularization study for policy gradient methods," Master's thesis, Institute of Computational Perception, Johannes Kepler University Linz, 2018.

[20] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "Ac/c++ code vulnerability dataset with code changes and cve summaries," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 508–512.

[21] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Advances in Neural Information Processing Systems*, 2019, pp. 10 197–10 207.

[22] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Transactions on Software Engineering*, 2021.

[23] A. Shrikumar, P. Greenside, and A. Kundaje, "Learning important features through propagating activation differences," in *International conference on machine learning*. PMLR, 2017, pp. 3145–3153.

[24] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-cam: Visual explanations from deep networks via gradient-based localization," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 618–626.

[25] Z. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec, "Gnnexplainer: Generating explanations for graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[26] D. Luo, W. Cheng, D. Xu, W. Yu, B. Zong, H. Chen, and X. Zhang, "Parameterized explainer for graph neural network," *Advances in neural information processing systems*, vol. 33, pp. 19 620–19 631, 2020.

[27] H. Yuan, H. Yu, J. Wang, K. Li, and S. Ji, "On explainability of graph neural networks via subgraph explorations," in *International conference on machine learning*. PMLR, 2021, pp. 12 241–12 252.

[28] Q. Zhu, D. Guo, Z. Shao, D. Yang, P. Wang, R. Xu, Y. Wu, Y. Li, H. Gao, S. Ma *et al.*, "Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence," *arXiv preprint arXiv:2406.11931*, 2024.

[29] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[30] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei *et al.*, "Starcoder 2 and the stack v2: The next generation," *arXiv preprint arXiv:2402.19173*, 2024.

[31] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, "Wizardcoder: Empowering code large language models with evol-instruct," *arXiv preprint arXiv:2306.08568*, 2023.

[32] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier *et al.*, "Mistral 7b," *arXiv preprint arXiv:2310.06825*, 2023.

[33] M. Javaheripi, S. Bubeck, M. Abdin, J. Aneja, S. Bubeck, C. C. T. Mendes, W. Chen, A. Del Giorno, R. Eldan, S. Gopi *et al.*, "Phi-2: The surprising power of small language models," *Microsoft Research Blog*, 2023.

[34] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[35] N. Japkowicz and M. Shah, *Evaluating learning algorithms: a classification perspective*. Cambridge University Press, 2011.

[36] X. Yin, C. Ni, and S. Wang, "Multitask-based evaluation of open-source llm on software vulnerability," *IEEE Transactions on Software Engineering*, 2024.

[37] E. Benhamou and D. Saltiel, "Similarities between policy gradient methods in reinforcement and supervised learning." in *ESANN*, 2020, pp. 721–726.

[38] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[39] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.

[40] A. Shestov, R. Levichev, R. Mussabayev, E. Maslov, A. Cheshkov, and P. Zadorozhny, "Finetuning large language models for vulnerability detection. arxiv 2024," *arXiv preprint arXiv:2401.17010*.

[41] T. G. Nguyen, T. Le-Cong, H. J. Kang, R. Widyasari, C. Yang, Z. Zhao, B. Xu, J. Zhou, X. Xia, A. E. Hassan *et al.*, "Multi-granularity detector for vulnerability fixes," *IEEE Transactions on Software Engineering*, vol. 49, no. 8, pp. 4035–4057, 2023.

[42] J. Zhou, M. Pacheco, Z. Wan, X. Xia, D. Lo, Y. Wang, and A. E. Hassan, "Finding a needle in a haystack: Automated mining of silent vulnerability fixes," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 705–716.

[43] R. Croft, M. A. Babar, and M. M. Kholoosi, "Data quality for software vulnerability datasets," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 121–133.

[44] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, "An introduction to deep reinforcement learning," *arXiv preprint arXiv:1811.12560*, 2018.

[45] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "Vulpecker: an automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016, pp. 201–213.

[46] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 292–303.

[47] X.-C. Wen, Y. Chen, C. Gao, H. Zhang, J. M. Zhang, and Q. Liao, "Vulnerability detection with graph simplification and enhanced graph representation learning," *arXiv preprint arXiv:2302.04675*, 2023.

[48] C. Ni, X. Yin, K. Yang, D. Zhao, Z. Xing, and X. Xia, "Distinguishing look-alike innocent and vulnerable code by subtle semantic representation learning and explanation," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1611–1622.

[49] T. Funke, M. Khosla, and A. Anand, "Hard masking for explaining graph neural networks," 2020.

[50] T. Schnake, O. Eberle, J. Lederer, S. Nakajima, K. T. Schütt, K.-R. Müller, and G. Montavon, "Higher-order explanations of graph neural networks via relevant walks," *IEEE transactions on pattern analysis and machine intelligence*, vol. 44, no. 11, pp. 7581–7596, 2021.

[51] R. Schwarzenberg, M. Hübner, D. Harbecke, C. Alt, and L. Hennig, "Layerwise relevance visualization in convolutional text graph classifiers," *arXiv preprint arXiv:1909.10911*, 2019.

[52] Q. Huang, M. Yamada, Y. Tian, D. Singh, and Y. Chang, "Graphlime: Local interpretable model explanations for graph neural networks," *IEEE Transactions on Knowledge and Data Engineering*, 2022.

[53] M. Vu and M. T. Thai, "Pgm-explainer: Probabilistic graphical model explanations for graph neural networks," *Advances in neural information processing systems*, vol. 33, pp. 12 225–12 235, 2020.

[54] M. Fu, C. Tantithamthavorn, V. Nguyen, and T. Le, "Chatgpt for vulnerability detection, classification, and repair: How far are we?" *arXiv preprint arXiv:2310.09810*, 2023.

[55] Z. Ren, X. Ju, X. Chen, and H. Shen, "Prorlearn: boosting prompt tuning-based vulnerability detection by reinforcement learning," *Automated Software Engineering*, vol. 31, no. 2, p. 38, 2024.

Yuan Jiang is an assistant professor at the School of Computer Science and Technology, Harbin Institute of Technology, and a postdoctoral researcher at Singapore Management University. His main research interests include pre-trained code language models, mining software repositories, software vulnerability detection, and large language model security.

Zhichen Qu is a graduate student at the School of Computer Science and Technology, Harbin Institute of Technology, under the supervision of Yuan Jiang. His main research interests include software vulnerability detection and pre-trained code language models.

Christoph Treude is an Associate Professor of Computer Science at Singapore Management University. His notable achievements include receiving an ARC Discovery Early Career Research Award (2018-2020) and securing funding from industry giants such as Google and Facebook. Treude has been honored with four best paper awards, including two ACM SIGSOFT Distinguished Paper Awards. Currently, he serves on the Editorial Boards of the IEEE Transactions on Software Engineering and the Springer journal on Empirical Software Engineering. Additionally, he is the Open Science Editor for the Elsevier Journal of Systems and Software and has chaired conferences such as ICSME 2020, ICPC 2023, and TechDebt 2023.

Xiaohong Su is a professor at the School of Computer Science and Technology, Harbin Institute of Technology. Her research interests include Intelligent software engineering, software vulnerability identification, code representation learning, bug triaging and localization, clone detection, and code search.

Tiantian Wang born in 1980. Received the Doctor's degree from Harbin Institute of Technology, Harbin, Heilongjiang, China, in 2009. Since 2013, she has been an Associate Professor in computer science department of Harbin Institute of Technology. Her current research interests are software engineering, program analysis and computer aided education.