ProRLearn: Boosting Prompt Tuning-based Vulnerability Detection by Reinforcement Learning

Zilong Ren
Nantong University
Xiaolin Ju
Ju.xl@ntu.edu.cn

Nantong University
Xiang Chen
Nantong University
Hao Shen
Nantong University

Research Article

Keywords: Vulnerability Detection, Prompt Tuning, Pre-trained Language Model, Reinforcement Learning

Posted Date: January 15th, 2024

DOI: https://doi.org/10.21203/rs.3.rs-3856133/v1

Additional Declarations: No competing interests reported.

Version of Record: A version of this preprint was published at Automated Software Engineering on April 20th, 2024. See the published version at https://doi.org/10.1007/s10515-024-00438-9.

ProRLearn: Boosting Prompt Tuning-based Vulnerability Detection by Reinforcement Learning

Zilong Ren[1], Xiaolin Ju[1*†], Xiang Chen[1*†] and Hao Shen[1†]

[1*]School of Information Science and Technology, Nantong University, Nantong, 226019, Jiangsu, China.

*Corresponding author(s). E-mail(s): ju.xl@ntu.edu.cn; xchencs@ntu.edu.cn;
Contributing authors: zilongren23@gmail.com; shenhyc@gmail.com;
†These authors contributed equally to this work.

Abstract

Software vulnerability detection is a critical step in ensuring system security and data protection. Recent research has demonstrated the effectiveness of deep learning in automated vulnerability detection. However, it is difficult for deep learning models to understand the semantics and domain-specific knowledge of source code. In this study, we introduce a new vulnerability detection framework ProRLearn, which leverages two main techniques (i.e., prompt tuning and reinforcement learning). Since existing fine-tuning of pre-trained language models (PLMs) struggles to leverage domain knowledge fully, we introduce a new automatic prompt-tuning technique. Precisely, prompt tuning mimics the pre-training process of PLMs by rephrasing task input and adding prompts, using the PLM's output as the prediction output. The introduction of the reinforcement learning reward mechanism aims to guide the behavior of vulnerability detection through a reward and punishment model, enabling it to learn effective strategies for obtaining maximum long-term rewards in specific environments. The introduction of reinforcement learning aims to encourage the model to learn how to maximize rewards or minimize penalties, thus enhancing performance. Experiments on two datasets (FFMPeg+Qemu and Reveal) indicate that ProRLearn achieves an F1 score improvement of 3.58%-28.6% over state-of-the-art baselines. The combination of prompt tuning

and reinforcement learning can offer a potential opportunity to improve performance in vulnerability detection. This means that it can effectively improve the performance level of the system in responding to constantly changing network environments and new threats. This interdisciplinary approach contributes to a better understanding of the interplay between natural language processing and reinforcement learning, opening up new opportunities and challenges for future research and applications.

Keywords: Vulnerability Detection, Prompt Tuning, Pre-trained Language Model, Reinforcement Learning

1 Introduction

Software security issues [1] have become increasingly prominent with the rapid development of information technology. Malicious attackers persistently search for and exploit vulnerabilities in systems and applications to gain unauthorized access, steal sensitive information, or disrupt systems. CVE-2022-30190, disclosed in May 2022, is a remote code execution vulnerability in the Microsoft Windows Support Diagnostic Tool that allows remote attackers to execute arbitrary shell commands on the target system. Several existing cases involve exploiting this vulnerability, including multiple phishing attacks by government-linked threat actors (Sandworm, UAC-0098, and APT28) and Russian government agencies aimed at infecting victims with information-stealing malware. In today's globally connected world, safeguarding computer systems from these threats has become an utmost priority.

Currently, detecting source code vulnerabilities can be divided into two broad categories: traditional vulnerability detection methods [2–5] and machine learning (or deep learning)-based vulnerability detection methods [6–11]. Previous vulnerability detection methods [2–5] mainly use rules defined in advance by experts to analyze the code. However, such an analysis method [6, 7] cannot identify some deeply hidden vulnerabilities [12, 13]. Deep learning (DL) has gained widespread usage in recent years for detecting source code vulnerabilities via automatic feature extraction.

Recently, several vulnerability-identifying frameworks [6, 7] utilize DL to detect and learn source code vulnerabilities have been proposed. For example, Devign [8] and ReVeal [14] use Graph Neural Network (GNN) [14] [9] on attribute graphs that integrate control flow, data dependencies, and Abstract Syntax Trees (ASTs) [8]. VulDeePecker [6] employs static analysis to extract program slices and trains a Bidirectional Long Short-Term Memory (Bi-LSTM) model to detect function-level vulnerabilities. Li et al. [7] used the Bi-LSTM to detect vulnerabilities. However, these DL-based methods encountered limitations in capturing full code semantics due to challenges associated with syntax-semantics distinctions, context dependencies, and domain-specific

knowledge. Meanwhile, PLMs [15–18] acquire powerful semantic representation capabilities through self-learning on large-scale datasets, which enable them to capture potential vulnerability features and learn patterns from the source code. **However, considering that there are often certain gaps between upstream and downstream tasks, and the unique characteristics and complexity of source code, relying solely on pre-trained models may not fully utilize the vulnerability information embedded in the source code.**

Therefore, we address the above issues from two perspectives. Firstly, we use PLM-based prompt tuning to address the shortcomings of fine-tuning. Prompt tuning can make the model more focused on learning features and patterns related to vulnerability detection. **It can make downstream tasks accommodate PLMs, which is more in line with the pre-training process. On the other hand, fine-tuning PLMs accommodates various downstream tasks, which may result in input and output biases.** Secondly, **as the sample data increases [19], the performance gap between fine-tuning and prompt tuning gradually narrows.** Furthermore, we propose a reward mechanism using policy gradients to address these limitations. This idea comes from reinforcement learning, which can learn better strategies as sample data increases, further improving model performance.

In this study, we propose ProRLearn, a novel vulnerability detection framework that effectively learns representation information from code. ProRLearn has two main components: (1) Guiding the model to generate features and patterns relevant to vulnerability detection by providing specific prompt templates. (2) Further optimize the model's performance through interactive learning with the environment and reinforcement learning. The model can continuously adjust the detection results based on feedback from the environment to maximize performance metrics for vulnerability detection.

To evaluate the effectiveness of ProRLearn, we employed two widely used datasets for vulnerability detection: FFMPeg+Qemu [8] and Reveal [14]. We conducted a comparative experiment between ProRLearn and six existing software vulnerability detection methods, namely Sysevr, VulDeePecker, IVDetect, Devign [8], Reveal [14] and AMPLE [20]. The experimental results on two datasets indicate that ProRLearn can improve F1 scores by 3.57% and 4.07%, respectively, and improve accuracy by 1.13% and 2.02%, respectively.

The main contributions of this study are as follows:

1. We propose ProRLearn, a PLM-based method that applies improved prompts with pre-trained knowledge to specific tasks and employs a reward mechanism to guide the learning process to enhance vulnerability detection.
2. We compare ProRLearn with six state-of-the-art baselines and find that it outperforms by 3.58% and 4.07% in F1 score and by 1.13% and 2.02% in accuracy, respectively.
3. An ablation experiment is exploring the effectiveness of each component of ProRLearn.

4. We share ProRLearn to encourage future studies on vulnerability detection[1].

2 Background

2.1 Vulnerability detection

Deep learning (DL)-based vulnerability detection methods can be classified into two categories. One category treats the source code as a natural language sequence and employs NLP techniques to represent the input code [6, 10, 21, 22], often utilizing methods like word2vec to initialize token embeddings in the code. Although these methods have shown acceptable performance, they come with certain limitations. It cannot capture the code syntax well or effectively encode unknown identifiers among source code. Another category attempts to leverage the structural information in code, abstracting code into a graph representation [7, 9, 14, 23], such as Abstract Syntax Trees, Control Flow Graphs, Data Flow Graphs, Program Dependency Graphs, and using these graphs as inputs to the model. Our research aims to address the limitations and issues that arise when transforming source code into flat sequences to overcome some of the constraints of traditional methods.

2.2 Prompt tuning

Prompt tuning is a method for pre-trained language models [24] to improve their adaptation to specific tasks by designing and adjusting prompt information in the model input. In the context of vulnerability detection tasks, prompt tuning can guide PLM to comprehend better the semantics and context related to vulnerabilities, thereby enhancing the model's ability to discover potential vulnerabilities. By adding relevant prompts about vulnerable codes to the model input, prompt tuning encourages the model to make more accurate judgments about potential vulnerabilities.

According to the flexibility of prompt templates, prompt tuning can be divided into two types: hard prompt and soft prompt. Specifically, the hard prompt, a discrete prompt, defines the task by including specific information as part of the input and providing clear guidance. Taking vulnerability detection as an example, discrete prompts can include descriptions about the type of vulnerability, code structure, or examples of specific vulnerabilities that are included as part of the input. Such templates are usually created manually and require some domain knowledge. The soft prompt, known as the continuous prompt, differs from the traditional discrete prompt in that it guides the model's learning and inference process by using continuous values as input to the model. The benefit of using continuous prompts is the ability to provide more flexible task descriptions. Models can better understand task requirements, context, and reason by learning relationships and semantics in continuous space.

[1] https://github.com/ProRLearn/ProRLearn001

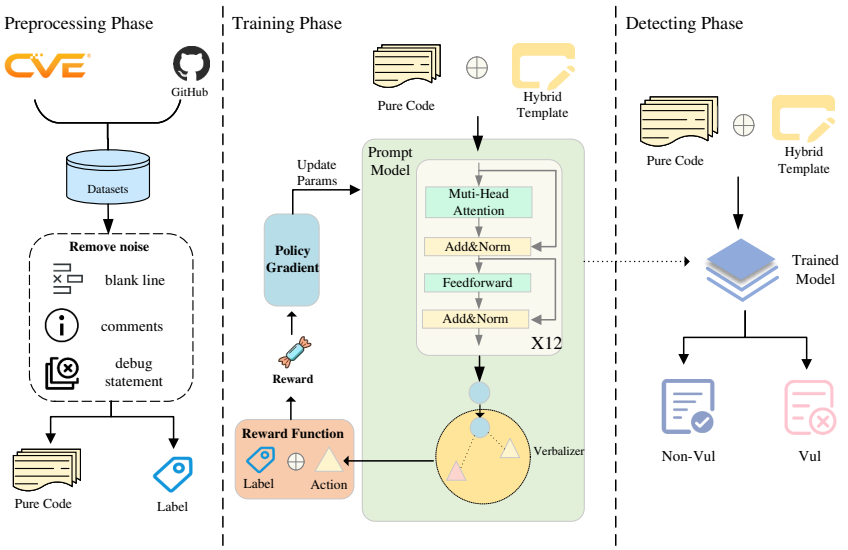

Fig. 1 Overview of ProRLearn

2.3 Reinforcement learning

Reinforcement learning (RL) possesses goal-directed advantages as it does not rely on exemplary supervision or comprehensive modeling of sample features [25]. Instead, RL [26–29] optimizes its strategy through multiple rounds of environment exploration and experience mining. RL performs superhuman without prior expert knowledge and exhibits the following characteristics [30–34]. First, the trial-and-error learning strategy ensures that the PLMs may have more feature choices, allowing them to explore different possibilities effectively. Second, the long-term reward mechanism is the feature selection of PLMs' long-term pursuit of reward maximization. These two characteristics can help the model better understand the task.

Classified according to maximizing long-term rewards, current reinforcement learning can be divided into value-based and policy-based methods. Value-Based method focuses on learning and optimizing the state value function, which is used to measure the quality of taking different actions in different states. Typical representatives of value function methods include Q-learning [35], Deep Q-Networks [36], etc. Policy-Based method: this method aims to directly learn the optimal policy without involving the value function. It represents the probability distribution of actions by parameterizing the policy and then uses various optimization techniques to maximize the expected reward. Policy gradient methods [37] and deep deterministic policy gradient methods [38] are examples of policy optimization methods.

3 Approach

The overall architecture of ProRLearn is shown in Fig. 1, which is divided into three main phases. For the preprocessing phase, we preprocess the collected datasets to remove noise that may affect vulnerability detection and ensure that the input to the model is pure code. For the training phase, the reinforcement learning environment is set up. The pure code and corresponding labels are used as the current environment, the pre-trained model is used as the agent, the strategy algorithm is determined, and the model is updated with the current strategy algorithm as the core. Next, we construct prompt tuning templates and use the CodeBERT [39] encoder to convert these inputs into vector representations. Following these steps, we obtain a trained model for identifying vulnerabilities during detection. For the detection phase, we input a combination of pure code and prompt templates into the trained model to detect the presence of vulnerabilities in the current code segment.

3.1 Preprocessing Phase

While examining the dataset, we identified noise in the code snippets that could affect the predictions made by PLM. Specifically, as shown in Fig. 1, there are three main types of noise: blank lines (extra indentation or line breaks), comments, and debugging statements. These are defined as noise primarily because PLM's input length is limited. To remove this noise, we implemented the following procedures.

Our experiments aimed to input pure code directly into the model. However, the code snippets in the dataset contained numerous line breaks. These line breaks would be encoded during the input process and mistakenly treated as part of the code, occupying space within the original code. To address this issue, we removed the excess line breaks.

Furthermore, the code snippets in the dataset include some comments. If these comments were input directly into the model, the model might mistakenly recognize them as code, potentially leading to incorrect assumptions about code vulnerabilities. Similarly, some debugging or output statements within code snippets occupy input space, such as the content printed by a *cout* statement in Fig. 2, which is solely used for displaying code execution progress and is unrelated to the variables in the code snippet. To address these noises, we carried out removal operations.

Fig. 2 illustrates three types of noise in a given code: `excess line breaks`, `comments`, and `irrelevant debugging` or `output statements`. Fig. 3 displays the style and structure of the code after our data processing. These processing steps help reduce noise and make the code more suitable for direct input into the model for analysis and prediction.

```
int main ( ) {
    pthread_key_create ( & threadKey , nullptr ) ;
    // Initialize the 'Debug' and 'Error' objects.
    Utils : : init ( & Debug , & Error ) ;
    {
         cout << end << "==================== Test 1" << endl;

         Utils : : HeaderValueList whitelistCookies ;

         whitelistCookies . push_back ( "c1" ) ;
         whitelistCookies . push_back ( "c2" ) ;
         ……
     }
         cout << endl << "All tests passed!" << endl ;
    return 0 ;
}
```

Fig. 2 Code snippets before data processing

```
int main ( ) {
    pthread_key_create ( & threadKey , nullptr ) ;
    Utils : : init ( & Debug , & Error ) ;
    {
         Utils : : HeaderValueList whitelistCookies ;
         whitelistCookies . push_back ( "c1" ) ;
         whitelistCookies . push_back ( "c2" ) ;
         ……
     }
    return 0 ;
}
```

Fig. 3 Code snippets after data processing

3.2 Prompt Tuning Implementation

In our approach, predicting and classifying vulnerable code based on prompt tuning primarily involves two steps. Firstly, selecting an appropriate PLM, considering that most PLMs are more suitable for specific tasks. Secondly, we apply a combination of prompt tuning [40, 41] to the selected PLM to achieve classification predictions for vulnerable code. Next, we will provide a detailed explanation of these steps.

Selecting the suitable PLM as the foundation for prompt tuning is crucial. PLMs can roughly be categorized into two types: autoregressive-based and autoencoder-based. Autoregressive PLMs, such as GPT[42], are better suited for generative tasks like text summarization and dialogue generation. On the other hand, autoencoder-based PLMs like BERT, after pre-training, can be applied to various downstream NLP tasks [43] such as classification and named entity recognition. CodeBERT, as an extension of the BERT [44] model, offers a unique advantage: it undergoes training on both natural language and source code. This training imparts a certain level of code knowledge to CodeBERT, and further fine-tuning with downstream task [45–49] corpora enhances its understanding of specific tasks. Inspired by the above research conclusions, we selected the widely used CodeBERT [39, 50] for the vulnerability classification task in our study.

Below is the specific prompt tuning process. As shown in Fig. 5, the processed source code and prompt template are combined and converted into new input by building a prompt template. Subsequently, these constructed prompt templates are input into the PLM. The model leverages its pre-trained

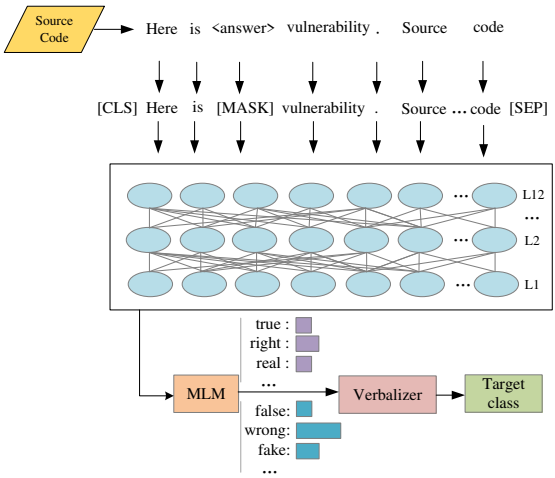

Fig. 4 The model architecture of prompt tuning

knowledge to predict the masked positions within the input. The predicted results are then mapped to the actual labels through the definition of a Verbalizer [45]. This process effectively transforms the downstream task into a masked prediction task, resembling what occurs during the pre-training phase.

The prompt template consists of three components: the input part (the source code in Fig. 5), the answer part ($< answer >$ in the figure), and the prompt words (such as "Here," "is," "vulnerability" in the figure). The input part is filled with the vulnerable code to be predicted. The answer part is filled with the vocabulary ultimately predicted by the PLM. The final predicted vocabulary output is subject to certain constraints, and its output is mapped to the target category labels through a verbalizer.

The construction of prompt words and prompt templates can also exist in various forms. Next, we will introduce the templates and verbalizers used. We use a Hybrid Prompt template as an innovative technique to prompt tuning. It combines the advantages of discrete prompts and continuous prompts. Hybrid prompts provide more flexible and controllable task guidance while maintaining model interpretability and tunability. We used and tested this template, and the results showed that the hybrid prompt performed best in vulnerability detection tasks. Hybrid prompt allows for discrete and continuous information, providing the model with more comprehensive task guidance. The following is the specific template creation process.

Hard prompt templates are often intuitive and simple. An example template is as follows:

$$f = \text{[input] Is the code vulnerable? [answer]} \tag{1}$$

Soft prompt templates are usually relatively abstract, but prompts are more flexible, allowing the model to play freely. An example template is as follows:

$$f = \text{[input] [MASK] [MASK] [MASK] [MASK]? [answer]} \quad (2)$$

Hybrid prompt templates combine the best of both worlds. In this template, vulnerability is a token related to the task. We do not want this token to be replaced.

$$f = \text{[input] [MASK] [MASK] code vulnerable? [answer]} \quad (3)$$

Verbalizer is a label-to-word mapping that aims to project target category labels onto words within the Verbalizer. These label words constrain the Pre-trained Language Model (PLM) output range, meaning that the model's output probabilities are focused exclusively on these label words. Each target category label can correspond to one or multiple label words. By utilizing the Verbalizer, the label word with the highest probability is mapped to the target category label, serving as the model's final prediction output.

In this task, the Verbalizer's corresponding label words can be added as needed or can directly use the target category as the label word. This design ensures that the model's output aligns with the task objective, enabling the model to generate language descriptions related to the target category, thereby enhancing model interpretability and comprehensibility. The Verbalizer is crucial in bridging the gap between the model's output and the task objective through this approach.

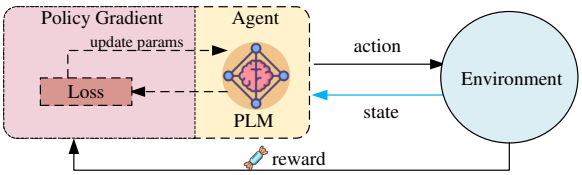

Fig. 5 Agent-environment interaction in RL

3.3 Reinforcement Learning Implementation

In our experiments, we decided not to employ unsupervised methods for fine-tuning the pre-trained model. Unsupervised learning methods often demand more significant computational resources and time for training, both of which were constrained in our setting. Therefore, in the interest of efficiency, we opted to leverage labeled vulnerability information to expedite the pre-trained model's learning process. This approach allowed us to use our existing labeled

data more effectively, achieving rapid improvements in model performance to align with our research and experimentation goals.

Next, we outline how to leverage the principles of reinforcement learning to fine-tune the pre-trained model. In our research, we adopted reinforcement learning to improve the performance of the pre-trained model further. The reinforcement learning approach is based on the following ideas, as shown in Fig. 5. In the classification task of vulnerability detection, we first created an RL environment. This environment included sample source code and their corresponding labels, which were used to define the reward mechanism. Then, we transform the source code and cue templates so that they can be directly input into the pre-trained model, such as the prompt tuning process. We determined the reward based on the model's prediction results. This means we could define a reward function based on the model's output to evaluate its performance at each step. Finally, based on the current state, action, and reward, we computed the gradient of the policy and used these gradients to update the network parameters.

The reinforcement learning (RL) elements in this context can be summarized as follows: (1) Using a pre-trained model as the agent within the RL framework. (2) The transformation of vulnerability code into a format the neural network understands, representing the state. (3) Actions correspond to the predictions made by the pre-trained model in classification tasks. (4) Reward function to measure the vulnerability detection model's performance in performing specific actions in the current state. (5) The policy which determines the agent's behavioral strategy in selecting actions to maximize rewards. (6) Given the discrete nature of classification tasks, the RL environment is effectively constructed based on a training sample pool of samples and their corresponding labels.

Traditional RL typically relies on Markov decision processes where the value of a state (s) depends on the value of the current action chosen and the value of the subsequent state (s'). The value of actions within a state is determined by a combination of rewards (r) and the values of the following state-action pairs. However, in classification tasks, states are often independent of each other. Therefore, we have adopted a different approach using a discrete Markov chain. In this setup, we consider only combinations of given states and available actions without considering logical subsequent states. In essence, our reinforcement learning framework comprises the following elements: the current state, the predicted action, the probability of selecting the current action, and the ultimately determined reward magnitude.

Since states are discrete, each state value or action value may have a limited impact on the final classification task. Therefore, we have introduced a novel approach for training the classification model. The basic idea of this method is to integrate the reward mechanism from reinforcement learning into the training process, employing a reward-based optimization approach. Our method draws inspiration from reinforcement learning, specifically the Vanilla Policy Gradient (VPG) method, which is used to update and train our model

rather than directly using the traditional cross-entropy loss function. VPG is a policy-based optimization algorithm with the primary objective of mapping the policy (or the probability of action selection) to the corresponding labels as effectively as possible, thereby maximizing cumulative rewards. The policy gradient expression is as follows:

$$\nabla J = -\frac{1}{B}\sum_{t=1}^{B}\nabla log\pi(a_t|s_t)R(t), \tag{4}$$

where B represents the batch size given in a single training run. t denotes the example code within the given batch B. s_t stands for the input example code, a_t is the predicted action category made by the agent, and $\pi(a_t|s_t)$ represents the probability of selecting a_t given the current state s_t. $R(t)$ corresponds to the reward function, which determines how well the model's action in state s_t performs based on task-specific criteria. Introducing VPG can better adjust the training process of our model to meet the specific requirements of the classification task.

According to the current gradient strategy, each step increases the log probability of each action, which is proportional to $R(t)$ (the sum of rewards at all past moments). However, the general logic should be that the agent intensifies its actions according to its consequences. Rewards received before taking an action have nothing to do with the quality of the action; only rewards received after the action will impact the agent's behavior. Therefore, the policy gradient expression of this idea is:

$$\nabla J = -\frac{1}{B}\sum_{t=1}^{B}\nabla log\pi(a_t|s_t)\sum_{t'=t}^{B}R(s_{t'}, a_{t'}). \tag{5}$$

The reward function is defined as follows: In each training batch, with a batch size of B, every input in the batch is considered a step. At each step, the agent's predicted action category y_t is compared to the actual class label y_t. If a_t equals to y_t, the agent receives a reward of 1; otherwise, the reward is 0. Throughout this process, the reward function $R(t)$ accumulates continuously, updating based on the prediction results at each step. This reward mechanism provides feedback to the agent, encouraging it to make correct predictions.

$$R(t) = \begin{cases} 1, & a_t = y_t \\ 0, & \text{otherwise.} \end{cases} \tag{6}$$

4 Experimental Evaluation

4.1 Research Questions

To evaluate ProRLearn, we aim to answer the following four research questions:

RQ1: How effective is ProRLearn in vulnerability detection?

To answer this question, we will compare ProRLearn with other approaches, including some latest graph-based and token-based vulnerability detection methods.

RQ2: How effective is prompt tuning for improving ProRLearn's performance on vulnerability detection?

For ProRLearn, the performance of the model is optimized by adjusting the Prompt template to meet the special requirements of vulnerability detection tasks. To answer this question, we will investigate the effectiveness of using different prompt templates.

RQ3: How effective is reinforcement learning for improving ProRLearn's performance on vulnerability detection?

For ProRLearn, the performance of the model is optimized by adjusting the reinforcement learning methods to meet the special requirements of vulnerability detection tasks. To answer this question, we will investigate the effectiveness of using different reinforcement learning methods.

RQ4: What is the effectiveness of ProRLearn with different pre-trained models?

For ProRLearn, the choice of different model architectures may have some impact on the results, and we aim to find a suitable pre-trained model that meets the specific requirements for vulnerability detection. To answer this question, we will investigate the effectiveness of using different pre-trained models.

4.2 Datasets

Our research used two vulnerability datasets, including FFMPeg+Qemu [8] and Reveal [14]. The FFMPeg+Qemu dataset is a balanced dataset widely used in previous studies [8, 20]. It is derived from two open-source C projects and comprises approximately 10k vulnerable entries and 12k non-vulnerable entries, vulnerabilities account for 45.02%. On the other hand, ReVeal is an imbalanced dataset. It originates from two open-source projects: Debian and Chromium. This dataset contains around 2k vulnerable entries and 20k non-vulnerable entries, vulnerabilities account for 9.16%. Table 1 summarizes dataset characteristics. Furthermore, we did not use the BigVul [51] dataset. The accuracy for BigVul was lower, as many of the vulnerability fixing commits used during data extraction for this dataset were large, tangled, or noisy [52].

Table 1 Statistics of the datasets.

Dataset	Samples	#Vul	#Non-vul	Vul Ration(%)
FFMPeg+Qemu	22,361	10,067	12,294	45.02
Reveal	18,169	1,664	16,505	9.16

4.3 Performance Metrics

We used the following four widely used performance metrics for evaluation:

TP: True Positive (TP) refers to the number of instances where the model correctly predicts positive class samples. In vulnerability detection, TP indicates cases where the model accurately identifies code with vulnerabilities.

TN: True Negative (TN) refers to the number of instances where the model correctly predicts negative class samples. In vulnerability detection, TN indicates cases where the model accurately determines that the code does not have vulnerabilities.

FN: False Negative (FN) occurs when the model incorrectly predicts samples that are positive as negative. In vulnerability detection, FN indicates cases where the model mistakenly claims that code has vulnerabilities without vulnerabilities.

FP: False Positive (FP) happens when the model incorrectly predicts samples that are negative as positive. In vulnerability detection, FP indicates cases where the model mistakenly claims that code without vulnerabilities has vulnerabilities.

Accuracy: Accuracy is the proportion of correctly predicted vulnerabilities to all vulnerabilities. TN represents the number of true negatives, and $TP+TN+FN+FP$ represents the total number of vulnerabilities.

$$Accuracy = \frac{TP+TN}{TP+TN+FN+FP} \tag{7}$$

Precision: Precision is the proportion of relevant vulnerabilities among the retrieved vulnerabilities. TP represents the number of true positives, and FP represents the number of false positives.

$$Precision = \frac{TP}{TP+FP} \tag{8}$$

Recall: Recall is the proportion of relevant vulnerabilities among the retrieved vulnerabilities. TP represents the number of true positives, and FN represents the number of false negatives.

$$Recall = \frac{TP}{TP+FN} \tag{9}$$

F1 score: The F1 score is the geometric mean of precision and recall, representing a balance between the two.

$$F1\ score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{10}$$

4.4 Baseline Methods

We compared ProRLearn with six baselines at the function level: four graph-based and two token-based methods. We select these baselines because they represent the most commonly used methods in the current field and can serve as effective controls for our proposed method. These baselines cover

both graph-based and token-based approaches, allowing us to evaluate the performance of our method against different types of baseline methods.

(1) SySeVR [7]: SySeVR is a vulnerability framework that utilizes a bidirectional recursive neural network. This framework extracts syntax and semantic features from the code to be examined and applies them to vulnerability detection.

(2) VulDeePecker [6]: VulDeePecker converts code into an intermediate representation that carries semantic information, such as data and control dependencies. This intermediate representation is then transformed into vectors, which serve as inputs to a bidirectional LSTM-based neural network for vulnerability detection.

(3) IVDetect [9]: IVDetect utilizes a Program Dependence Graph (PDG) to represent the code and extracts information as vector representations. It then employs the Factorized Aggregated Graph Convolutional Network (FA-GCN) to classify the vector representations for vulnerability detection.

(4) Devign [8]: Devign is a source code vulnerability detection model based on Graph Neural Networks (GNN). It utilizes GNN to learn rich semantic information from the source code. The model consists of a Conv module, which extracts valuable features for graph-level classification.

(5) Reveal [14]: Reveal utilizes Code Property Graphs (CPG) and employs the GGNN (Gated Graph Neural Network) to obtain graph embeddings from the CPG. Then, it utilizes a Multi-Layer Perceptron (MLP) for classification and detection.

(6) AMPLE [20]: AMPLE simplifies and enhances the graph based on the code structure diagram, and uses GCN to obtain graph embeddings. Then, it utilizes a Multi-Layer Perceptron (MLP) for classification and detection.

4.5 Experimental Settings

We followed the hyper-parameters and dataset split outlined in the original Baseline papers to ensure accuracy and fairness in our experiments. For each dataset, we followed the same settings as other experiments [20] and divided it into 80% training set, 10% validation set, and 10% test set, as this is a standard testing setup used in prior research [20, 53, 54]. In the case of Devign, since the code was not provided, we replicated the experiments based on the methodology provided by ReVeal.

We used CodeBERT as our model with a maximum input sequence length of 512. Our model was optimized using the Adam optimizer with a batch size of 16 and a learning rate of 2e-5. Additionally, we incorporated hybrid templates during prompt tuning and employed the VPG for reinforcement learning with a reward magnitude of 1.

Our model training was conducted on a server equipped with an NVIDIA GeForce RTX 4090, and each training session consisted of 20 epochs.

5 Experimental Results

5.1 RQ1. Effectiveness of ProRLearn

To demonstrate the effectiveness of ProRLearn, we evaluated its performance by comparing ProRLearn with six baselines on two datasets. The experimental results are presented in Table 2.

Table 2 Comparison between ProRLearn and two datasets for copper leakage detection methods. "-" indicates that the method does not apply to the current dataset. The best results for each metric are highlighted in bold.

Metrics(%) \ Dataset	FFMPeg+Qemu				Reveal			
Baseline	Accuracy	Precision	Recall	F1	Accuracy	Precision	Recall	F1
SySeVR	48.59	47.08	60.02	52.77	73.21	43.56	27.84	33.97
VulDeePecker	50.12	47.89	33.34	39.31	78.51	20.63	14.59	17.09
Devign	57.19	52.41	58.11	55.11	86.38	28.98	34.73	31.60
Reveal	62.73	53.94	71.22	61.39	85.25	29.73	**64.96**	40.79
IVDetect	56.22	55.18	59.94	57.46	-	-	-	-
AMPLE	63.01	53.26	83.21	64.33	90.72	50.06	43.28	46.42
ProRLearn	**64.14**	**55.99**	**86.27**	**67.91**	**92.74**	**53.18**	48.06	**50.49**

Based on the Table 2, we achieve the following findings. First, we can observe that ProRLearn outperforms all the baselines. ProRLearn achieves higher F1 scores, recall, and accuracy on both datasets compared to baselines. Specifically, ProRLearn improves the F1 score by 3.58% and 4.07%, respectively, compared to the current best baseline method. The corresponding relative improvements are 5.57% and 8.77% for the F1 score. Additionally, ProRLearn increases the recall score by 3.06% on FFMPeg+Qemu [8], with relative improvements of 4.76%. Moreover, ProRLearn raises the accuracy score by 1.13% and 2.02%, respectively, with relative improvements of 1.79% and 2.23%. In addition, we also compared with LineVul [53] on the FFMPeg+Qemu and Reveal datasets. On these two datasets, LineVul achieved the F1 scores of 56.54% and 44.79%, respectively.

In other words, the experiment results indicate that the ProRLearn framework surpasses existing works that utilize graph properties and semantic information. In many previous studies, it has been believed that graph-based feature extraction is more effective in detecting code vulnerabilities than semantic and syntactic feature extraction.

However, Table 2 shows that graph-based methods (IVDetect, Devign, Reveal, AMPLE) perform better than token-based methods (SySeVR, VulDeePecker) in three metrics. This is the exact opposite of our experimental results. There could be several reasons for this discrepancy. In past research

on semantic and syntactic features, most studies were based on RNN architectures, which (1) did not address the long-term dependency problem effectively and (2) were trained on specific vulnerability datasets. Our approach successfully addressed the issues as mentioned above, and our research results demonstrate that ProRLearn is more accurate than state-of-the-art methods.

Answer to RQ1: ProRLearn excels beyond all baseline methods across the metrics of accuracy, precision, and F1 score. Specifically, ProRLearn outperforms the best baseline method in F1 scores on the two datasets by 3.58% and 4.07%, respectively.

Table 3 The impact of different prompt methods on the performance of ProRLearn.

Dataset	Method	Accuracy	Precision	Recall	F1 score
FFMPeg+Qemu	non-prompt	58.05	54.11	74.33	62.63
	hard-prompt	63.21	53.19	**89.63**	66.77
	soft-prompt	63.39	54.79	86.25	67.01
	hybrid-prompt	**64.14**	**55.99**	86.27	**67.91**
Reveal	non-prompt	90.98	57.86	32.84	41.90
	hard-prompt	89.79	**65.60**	35.19	45.81
	soft-prompt	91.22	51.72	47.96	49.77
	hybrid-prompt	**91.47**	53.18	**48.06**	**50.49**

5.2 RQ2. Effectiveness of Prompt Tuning

To answer this research question, we initially delved into the contribution of prompt tuning to the performance of ProRLearn and the effectiveness of various prompt learning methods.

Our relevant methods for assessing prompt tuning are hard prompt, soft prompt, and hybrid prompt. We conducted ablation experiments on two datasets to evaluate the effectiveness of prompt tuning. In particular, we carried out four experiments on ProRLearn: (1) ProRLearn without prompt tuning; (2) ProRLearn with the hard prompt; (3) ProRLearn with the soft prompt; and (4) ProRLearn with the hybrid prompt. The results are presented in Table 3.

Compared to ProRLearn without prompt tuning, the hybrid prompt achieved significant improvements in F1 scores of 5.28% and 8.59% on the two datasets. Additionally, the precision score increased by 1.88% on the ReVeal [14]. The recall score also demonstrated substantial improvements of 11.94% and 15.22%. Hard and soft prompts outperformed ProRLearn without prompt learning in terms of F1 and recall scores on the two datasets, indicating

that the prompt tuning module enhances ProRLearn's performance. Furthermore, the hybrid prompt showed slightly superior performance to hard and soft prompts, with F1 scores increasing by 1.14% and 4.68% on the two datasets, respectively. These results underscore the effectiveness of prompt tuning in improving ProRLearn's performance in vulnerability detection tasks.

From Table 3, we can observe that any prompt template enhances ProRLearn's performance. This is because prompt tuning transforms the original classification task into a cloze-style format, similar to the pre-training phase of the PLM. Prompt tuning enables a more comprehensive and effective utilization of the pre-trained knowledge [55] within the PLM. As a result, prompt tuning methods exhibit improved performance in vulnerability detection, underscoring the effectiveness of prompt tuning.

Answer to RQ2: Prompt tuning contributes significantly to the performance of ProRLearn, with an F1 score improvement of 5.28% and 8.59% on the two datasets, respectively.

Table 4 Performance differences between different reinforcement learning strategies.

Dataset	Method	Accuracy	Precision	Recall	F1 score
FFMPeg+Qemu	non-RL	61.54	52.44	74.01	62.57
	Q-RL	59.77	54.02	73.33	62.22
	PG-RL	**64.14**	**55.99**	**86.27**	**67.91**
Reveal	non-RL	88.86	42.92	42.48	42.70
	Q-RL	89.32	47.17	40.08	43.34
	PG-RL	**91.47**	**53.18**	**48.06**	**50.49**

5.3 RQ3. Effectiveness of Reinforcement learning

To answer this research question, we aim to explore reinforcement learning ideas' contribution to ProRLearn performance and evaluate the effectiveness of different reinforcement learning methods.

It should be noted that in reinforcement learning, the two most common training methods are policy-based reinforcement learning and value function-based reinforcement learning. To evaluate the effectiveness of reinforcement learning, we performed ablation experiments on three different versions of ProRLearn for two datasets: ProRLearn without reinforcement learning (denoted as non-RL), ProRLearn with a value function (denoted as Q-RL), ProRLearn (denoted as PG-RL) using the policy function. The results are shown in Table 4. The best results are highlighted in bold.

Performance is similar between contrasting value functions and not using reinforcement learning (only using prompt tuning). Reinforcement learning

using policy functions significantly improves performance. All evaluation indicators of the policy function on both data sets are better than other methods. Among them, compared with the ProRLearn method using non-RL, the f1 score increased by 5.34% and 7.79%, and the accuracy increased by 2.6% and 2.59%, respectively. Analysis of this situation shows that RL of value functions is unsuitable for this task because value functions are suitable for evaluating whether a state is good or bad, while policy functions are suitable for determining the actions that should be taken in each state, similar to classification Tasks. Therefore, the RL of policy functions is more suitable for improving vulnerability detection performance.

Answer to RQ3: Reinforcement learning contributes significantly to the performance of ProRLearn, with an F1 score improvement of 5.34% and 7.79% on the two datasets, respectively.

Table 5 Performance differences between different pre-trained models.

Dataset	Method	Accuracy	F1 score	Method	Accuracy	F1 score
FFMPeg+Qemu	BERT	54.96	55.84	BERT + RL + PT	60.39	59.59
	Roberta	48.68	58.73	Roberta + RL + PT	46.52	63.44
	CodeT5	46.59	56.83	CodeT5 + RL + PT	50.76	63.69
	CodeBERT	59.77	61.59	CodeBERT + RL + PT	61.71	66.10
Reveal	BERT	84.19	33.51	BERT + RL + PT	88.74	36.40
	Roberta	87.58	39.52	Roberta + RL + PT	88.61	46.10
	CodeT5	82.14	35.77	CodeT5 + RL + PT	86.31	37.86
	CodeBERT	89.96	41.98	CodeBERT + RL + PT	89.74	46.15

5.4 RQ4. Effectiveness of Prolearn with Different Models

To answer this research question, we explore the performance of our method on different pre-trained models.

Although our ProRLearn finally applied the pre-trained model CodeBERT. However, during the empirical study, we extended the experiments to verify whether our basic idea is specific to CodeBERT. Specifically, we only replace the PLM in ProRLearn and keep other ideas unchanged to evaluate our method. When evaluating other models, to be able to ensure that all models are functioning correctly. Parameter adjustments have been made for all models. Adjust the original max_len parameter to 256, leaving other parameters unchanged. When evaluating the model, we still use four metrics to measure its performance comprehensively. This helps us better understand the applicability and validity of our ideas to different models.

It is obvious from Table 5 that our method can significantly improve the model's performance no matter which model architecture is used. PT in the table represents a prompt tuning. Specifically, when we apply the idea in the BERT architecture, the F1 score increases by 3.75% and 2.89%. When

we apply the idea in the CodeBERT architecture, the F1 score increases by 4.51% and 4.17%. When applying the idea in the CodeT5 architecture, the F1 score increased by 6.86% and 2.09%. When applying the idea in the Roberta architecture, the F1 score increased by 4.71% and 6.58%. This finding demonstrates the broad applicability of our ideas to larger code bases, as well as to the vulnerability detection domain. We conducted tests on two different datasets, and the results showed that the CodeBERT architecture we adopted outperformed other architectures in performance. Furthermore, CodeBERT significantly outperforms other models regardless of whether our idea is used. This further proves the effectiveness and superiority of CodeBERT in vulnerability detection tasks.

It is worth noting that CodeBERT, Roberta [56], CodeT5, and BERT belong to the same model architecture but use different data sets in the pre-training stage. CodeBERT uses code-related data sets for training in the pre-training stage, which may be one of the reasons why CodeBERT performs better in vulnerability detection tasks. Although CodeT5 also uses code-related data sets in the training phase, CodeT5 adopts a text-to-text architecture and is more suitable for code annotation or translation tasks.

Answer to RQ4: Different choices of pre-trained models can influence the performance of ProRLearn in vulnerability detection. We have experimentally found that using CodeBERT can achieve the best performance.

6 Discussion

In this section, we perform additional analysis to discuss the results of our ProRLearn approach further and provide some recommendations for future researchers.

6.1 How does the size of the reward and verbalizer impact the performance of ProRLearn?

The impact of the verbalizer is shown in Fig. 6. We tried five different numbers of verbalizers, the numbers being 1, 2, 3, 4, and 5 respectively. We form a one-to-many action verbalizer by adding task-related tag words with similar meanings to the target tag to improve the performance of prompt tuning. However, it is important to emphasize that adding more verbalizers is not necessarily better.

According to the results shown in Fig. 6, we can observe that the number of verbalizers is 3, and the model performance reaches the best state. When the number of verbalizers increases to 4 or 5, the F1 score decreases slightly. This means that as the number of prompt words continues to increase, performance may not continue to improve. Selecting an appropriate number of verbalizers for combination can further improve the performance of prompt tuning while reducing the cost of searching for the best performance template.

The impact of reward size: The size of rewards can affect the learning speed of intelligent agents. Greater rewards make it easier for agents to understand

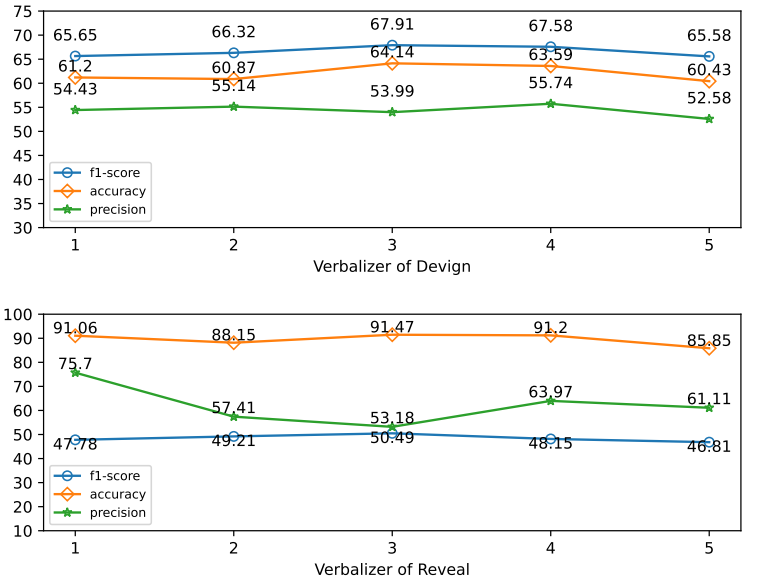

Fig. 6 Comparison of verbalizers based on vulnerability detection.

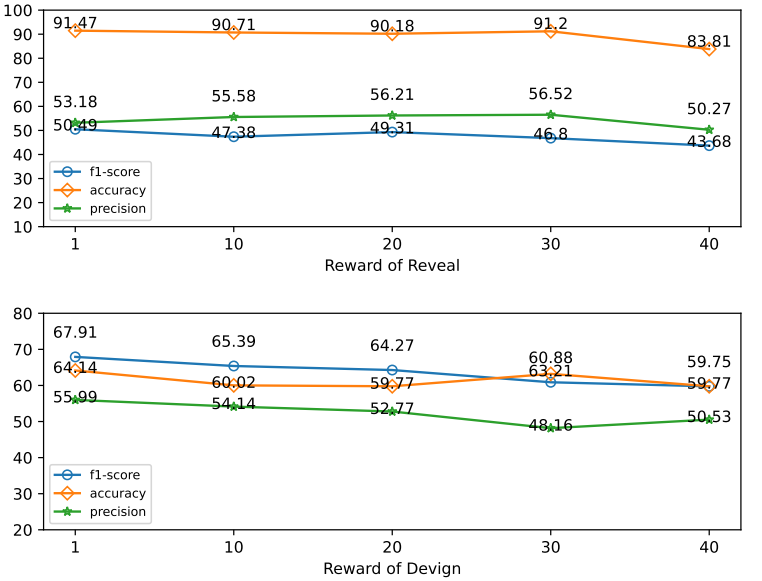

Fig. 7 Performance comparison with different reward sizes.

good behavior and may converge to the optimal strategy faster. The reward is small, and the agent will need more training samples and learning time. However, if the reward is too large, it may lead to unstable training, and the agent may be unable to find the optimal strategy. Therefore, we need to find the right reward size to ensure that the model performs well in the task.

As shown in Fig. 7, we investigated five different reward values, namely 1, 10, 20, 30, and 40. It is worth noting that the model performs best when the reward value is 1, and as the reward value gradually increases, the model's performance gradually decreases. Specifically, it can be observed from the table that when the reward value increases from 1 to 40, the F1 score significantly decreases by 8.16%. This indicates that the reward value set is inappropriate and will have a negative impact on the detection performance of the model. Therefore, careful consideration is needed when choosing reward values to ensure the model performs well.

6.2 How do the different prompt templates impact the performance of ProRLearn?

From the perspective of prompt templates, we analyze the impact of different types of templates on model performance. For all prompt template types, we build three types. They are hard prompts (H1, H2, H3), soft prompts (S1, S2, S3) and mixed prompts (D1, D2, D3). The three templates set are: 1) Prefix prompt template: the prompt word comes first, and the source code comes after; 2) Suffix prompt template: the source code comes first, and the prompt word comes after; 3) Double-fix prompt template: the prompt word comes after Both sides, source code in the middle.

Table 6 The impact of different prompt templates on the performance of ProRLearn.

	Template	FFMPeg+Qemu				Reveal			
		Accuracy	Precision	Recall	F1	Accuracy	Precision	Recall	F1
H1	Here is [answer] vulnerability. [input]	63.36	54.12	76.37	63.35	90.01	51.51	36.63	47.11
S1	[Mask] [Mask] [answer] [Mask] [Mask] [input]	63.54	51.88	83.73	65.20	89.51	**54.65**	37.27	47.27
D1	[Mask] [Mask] [answer] vulnerability [Mask] [input]	65.57	53.32	85.16	65.58	91.27	53.16	43.72	47.98
H2	[input] This code is a vulnerability. [answer]	61.48	58.73	64.37	61.42	87.07	52.52	36.63	43.16
S2	[input] [Mask] [Mask] [Mask] [Mask] [Mask] [Mask] [answer]	62.84	58.07	66.34	61.93	88.15	50.55	38.62	43.79
D2	[input] [Mask] [Mask] [Mask] [Mask] vulnerability [Mask] [answer]	63.39	**59.08**	67.94	63.20	88.73	50.69	39.37	44.32
H3	This code [input] is a vulnerability. [answer]	63.59	53.26	84.63	65.38	90.68	53.39	45.89	49.36
S3	[Mask] [Mask] [input] [Mask] [Mask] [Mask] [Mask] [answer]	62.12	55.34	85.27	67.12	90.47	52.12	47.62	49.77
D3	[Mask] [Mask] [input] [Mask] [Mask] vulnerability [Mask] [answer]	**64.14**	55.99	**86.27**	**67.91**	**91.47**	53.18	**48.06**	**50.49**

According to the results in Table 6, we can observe that the double-fix prompt template has the best effect, which may be because it combines the advantages of the prefix prompt template and the suffix prompt template. The best-performing prompt template (D3) has an improvement of 6.49% relative to the worst-performing prompt template (H2). In addition, prefix prompt templates (H1, S1, D1) generally perform better than suffix prompt templates

(H2, S2, D2), meaning placing the prompt words in front of the input text can achieve better results. This may be because the prompt words in the prefix position can better guide the pre-trained model to focus on learning the target task.

6.3 How does our model improve performance on datasets with different sample sizes?

We evaluate the performance improvement of our method across varying sample sizes. The original dataset was segmented into four scenarios, with 20%, 40%, 60%, and 80% of the data volume as training data, respectively. The test set and training set remain unchanged, and we exclusively utilize the F1 score to measure model performance in this evaluation.

As seen from Table 7, ProRLearn can improve performance with a few samples. As the sample size continues to increase, the improvement effect increases significantly. The performance improvement is the highest when the sample size is increased to 80%. ProRLearn achieves performance improvements with both few and many samples. The reason is that prompt tuning can perform better than fine-tuning in a few samples, and RL can further improve performance as the sample size increases.

Table 7 ProRLearn's performance (in terms of F1-score (%)) in vulnerability detection in scenarios with different sample sizes.

Dataset	Method	20%	40%	60%	80%
FFMPeg+Qemu	fine-tuning	61.14	61.27	54.76	61.94
	prompt-learning	63.07	64.93	57.20	62.55
	ProRLearn	64.93	65.76	62.22	67.91
Reveal	fine-tuning	31.62	36.09	40.80	43.78
	prompt-learning	34.74	40.08	43.22	46.47
	ProRLearn	35.31	41.94	45.37	50.49

7 Threats to Validity

Threats to internal validity mainly relate to minimizing system error. ProRLearn is controlled by multiple parameters, including learning rate, optimizer, batch size, etc. Different settings of these parameters will produce different results. However, exploring optimal parameter settings can be difficult due to the large number of parameters. Our research aims not to seek optimal parameter settings but to demonstrate the performance of our method through fair comparison with baseline models. Therefore, the performance of this paper

can be considered as the lower limit of our method, and the performance can be further improved through parameter optimization.

Threats to external validity mainly relate to the limited size of the experimental dataset. ProRLearn was evaluated on two datasets because these two datasets have been previously used in vulnerability detection-related research work. We only conducted experiments on the C/C++ datasets and did not cover datasets from other programming languages, such as Java and Python. In future work, we plan to expand the scope of experiments and evaluate more datasets to verify and evaluate the effectiveness of ProRLearn.

8 Conclusion and Future work

In this paper, we propose ProRLearn, a novel vulnerability detection framework that combines pre-trained models, prompt tuning, and reinforcement learning. ProRLearn can quickly apply pre-trained models to specific tasks with the help of enhanced prompts. RL also guides the model to optimize specific tasks iteratively. ProRLearn can improve incrementally by interacting with the environment rather than relying solely on static supervisory signals. Compared with state-of-the-art DL-based methods, ProRLearn significantly improves vulnerability detection performance on both datasets, with an F1 score improvement of 3.58% - 28.6%. The results demonstrate the practicality and importance of our ProRLearn in vulnerability detection, reducing the workload of manual review and vulnerability detection, thereby saving time and cost.

In the future, we plan to conduct large-scale experiments to explore various prompt settings and combinations while seeking strategies to optimize model performance. We will take into account factors such as training time and overall performance in our comprehensive evaluation.

References

[1] Nord, R.L.: Software vulnerabilities, defects, and design flaws: A technical debt perspective. In: the 14th Annual Acquisition Research Symposium, vol. 2017, pp. 67–75. Acquisition Research Program, Naval Postgraduate School (2017)

[2] Cherem, S., Princehouse, L., Rugina, R.: Practical memory leak detection using guarded value-flow analysis. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 480–491. Association for Computing Machinery, New York, NY, USA (2007)

[3] Fan, G., Wu, R., Shi, Q., Xiao, X., Zhou, J., Zhang, C.: Smoke: scalable path-sensitive memory leak detection for millions of lines of code. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 72–82. IEEE, Montreal, QC, Canada (2019)

[4] Kroening, D., Tautschnig, M.: Cbmc–c bounded model checker: (competition contribution). In: Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Grenoble, France, April 5-13, 2014., pp. 389–391. Springer, Berlin, Heidelberg (2014)

[5] Heine, D.L., Lam, M.S.: Static detection of leaks in polymorphic containers. In: Proceedings of the 28th International Conference on Software Engineering, pp. 252–261. Association for Computing Machinery, New York, NY, USA (2006)

[6] Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y.: Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681 (2018)

[7] Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z.: Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Transactions on Dependable and Secure Computing **19**(4), 2244–2258 (2021)

[8] Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y.: Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Advances in neural information processing systems **32** (2019)

[9] Li, Y., Wang, S., Nguyen, T.N.: 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, pp. 292–303. Association for Computing Machinery, New York, NY, USA (2021)

[10] Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., McConley, M.: Automated vulnerability detection in source code using deep representation learning. In: 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA), pp. 757–762. IEEE, Orlando, FL, USA (2018)

[11] Lomio, F., Iannone, E., De Lucia, A., Palomba, F., Lenarduzzi, V.: Just-in-time software vulnerability detection: Are we there yet? Journal of Systems and Software **188**, 111283 (2022)

[12] Cao, S., Sun, X., Bo, L., Wu, R., Li, B., Tao, C.: MVD: memory-related vulnerability detection based on flow-sensitive graph neural networks. In: Proceedings of the 44th International Conference on Software Engineering, pp. 1456–1468. Association for Computing Machinery, New York, NY, USA (2022)

[13] Cheng, X., Zhang, G., Wang, H., Sui, Y.: Path-sensitive code embedding

via contrastive learning for software vulnerability detection. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 519–531. Association for Computing Machinery, New York, NY, USA (2022)

[14] Chakraborty, S., Krishna, R., Ding, Y., Ray, B.: Deep learning based vulnerability detection: Are we there yet. IEEE Transactions on Software Engineering **48**(9), 3280–3296 (2021)

[15] Han, X., Zhang, Z., Ding, N., Gu, Y., Liu, X., Huo, Y., Qiu, J., Yao, Y., Zhang, A., Zhang, L., *et al.*: Pre-trained models: Past, present and future. AI Open **2**, 225–250 (2021)

[16] Qiu, X., Sun, T., Xu, Y., Shao, Y., Dai, N., Huang, X.: Pre-trained models for natural language processing: A survey. Science China Technological Sciences **63**(10), 1872–1897 (2020)

[17] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., Liu, P.J.: Exploring the limits of transfer learning with a unified text-to-text transformer. The Journal of Machine Learning Research **21**(1), 5485–5551 (2020)

[18] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., *et al.*: Language models are few-shot learners. Advances in neural information processing systems **33**, 1877–1901 (2020)

[19] Li, X., Ren, X., Xue, Y., Xing, Z., Sun, J.: Prediction of vulnerability characteristics based on vulnerability description and prompt learning. In: 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Taipa, Macao, pp. 604–615 (2023). IEEE

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

[21] Dam, H.K., Tran, T., Pham, T., Ng, S.W., Grundy, J., Ghose, A.: Automatic feature learning for vulnerability prediction. arXiv preprint arXiv:1708.02368 (2017)

[22] Wang, C., Yang, Y., Gao, C., Peng, Y., Zhang, H., Lyu, M.R.: No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 382–394. Association for Computing Machinery, New York, NY, USA (2022)

[23] Wu, Y., Zou, D., Dou, S., Yang, W., Xu, D., Jin, H.: VulCNN: An image-inspired scalable vulnerability detection system. In: Proceedings of the 44th International Conference on Software Engineering, pp. 2365–2376. Association for Computing Machinery, Pittsburgh, USA (2022)

[24] Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., Neubig, G.: Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. ACM Computing Surveys **55**(9), 1–35 (2023)

[25] Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. Robotica **17**(2), 229–235 (1999)

[26] Zeng, A., Song, S., Welker, S., Lee, J., Rodriguez, A., Funkhouser, T.: Learning synergies between pushing and grasping with self-supervised deep reinforcement learning. In: 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 4238–4245. IEEE, Madrid, Spain (2018)

[27] Rosenstein, M.T., Barto, A.G., Si, J., Barto, A., Powell, W., Wunsch, D.: Supervised actor-critic reinforcement learning. Learning and Approximate Dynamic Programming: Scaling Up to the Real World, 359–380 (2004)

[28] Lagoudakis, M.G., Parr, R.: Reinforcement learning as classification: Leveraging modern classifiers. In: Proceedings of the 20th International Conference on Machine Learning (ICML-03), pp. 424–431. AAAI Press, Washington, DC USA (2003)

[29] Kaelbling, L.P., Littman, M.L., Moore, A.W.: Reinforcement learning: A survey. Journal of artificial intelligence research **4**, 237–285 (1996)

[30] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., *et al.*: Human-level control through deep reinforcement learning. nature **518**(7540), 529–533 (2015)

[31] Caicedo, J.C., Lazebnik, S.: Active object localization with deep reinforcement learning. In: Proceedings of the IEEE International Conference on Computer Vision (ICCV), Santiago Chile, pp. 2488–2496 (2015)

[32] Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., *et al.*: Mastering the game of go with deep neural networks and tree search. nature **529**(7587), 484–489 (2016)

[33] Sallab, A.E., Abdou, M., Perot, E., Yogamani, S.: Deep reinforcement learning framework for autonomous driving. Electronic Imaging **29**(19),

70–76 (2017)

[34] Shao, K., Tang, Z., Zhu, Y., Li, N., Zhao, D.: A survey of deep reinforcement learning in video games. arXiv e-prints, 1912 (2019)

[35] Watkins, C.J., Dayan, P.: Q-learning. Machine learning **8**, 279–292 (1992)

[36] Osband, I., Blundell, C., Pritzel, A., Van Roy, B.: Deep exploration via bootstrapped DQN. Advances in neural information processing systems **29**, 4026–4034 (2016)

[37] Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., Riedmiller, M.: Deterministic policy gradient algorithms. In: International Conference on Machine Learning, pp. 387–395. Pmlr, Bejing, China (2014)

[38] Qiu, C., Hu, Y., Chen, Y., Zeng, B.: Deep deterministic policy gradient (ddpg)-based energy harvesting wireless communications. IEEE Internet of Things Journal **6**(5), 8577–8588 (2019)

[39] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., *et al.*: CodeBERT: A pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics, pp. 1536–1547. Association for Computational Linguistics, Online (2020)

[40] Jiang, Z., Xu, F.F., Araki, J., Neubig, G.: How can we know what language models know? Transactions of the Association for Computational Linguistics **8**, 423–438 (2020)

[41] Qin, G., Eisner, J.: Learning how to ask: Querying lms with mixtures of soft prompts. In: Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT), pp. 5203–5212. Association for Computational Linguistics, Online (2021)

[42] Liu, X., Zheng, Y., Du, Z., Ding, M., Qian, Y., Yang, Z., Tang, J.: Gpt understands, too. AI Open (2023)

[43] Howard, J., Ruder, S.: Universal language model fine-tuning for text classification. In: Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 328–339. Association for Computational Linguistics, Melbourne, Australia (2018)

[44] Devlin, J., Chang, M.-W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018)

[45] Schick, T., Schütze, H.: Exploiting cloze-questions for few-shot text classification and natural language inference. In: Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume, pp. 255–269. Association for Computational Linguistics, Online (2021)

[46] Li, X.L., Liang, P.: Prefix-tuning: Optimizing continuous prompts for generation. In: Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), pp. 4582–4597. Association for Computational Linguistics, Online (2021)

[47] Lester, B., Al-Rfou, R., Constant, N.: The power of scale for parameter-efficient prompt tuning. In: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, pp. 3045–3059. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic (2021)

[48] Gu, Y., Han, X., Liu, Z., Huang, M.: Ppt: Pre-trained prompt tuning for few-shot learning. In: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 8410–8423. Association for Computational Linguistics, Dublin, Ireland (2022)

[49] Han, X., Zhao, W., Ding, N., Liu, Z., Sun, M.: Ptr: Prompt tuning with rules for text classification. AI Open **3**, 182–192 (2022)

[50] Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., et al.: Huggingface's transformers: State-of-the-art natural language processing. arXiv preprint arXiv:1910.03771 (2019)

[51] Fan, J., Li, Y., Wang, S., Nguyen, T.N.: Ac/c++ code vulnerability dataset with code changes and cve summaries. In: 2020 IEEE/ACM 17th International Conference on Mining Software Repositories (MSR), pp. 508–512 (2020). IEEE

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

[53] Fu, M., Tantithamthavorn, C.: Linevul: A transformer-based line-level vulnerability prediction. In: Proceedings of the 19th International Conference on Mining Software Repositories, pp. 608–620 (2022). IEEE

[54] Hin, D., Kan, A., Chen, H., Babar, M.A.: Linevd: Statement-level vulnerability detection using graph neural networks. In: 2022 IEEE/ACM

19th International Conference on Mining Software Repositories (MSR), pp. 596–607 (2022). IEEE

[55] Sun, C., Qiu, X., Xu, Y., Huang, X.: How to fine-tune BERT for text classification? In: Proceedings of the 18th China National Conference on Chinese Computational Linguistics, Kunming, China, October 18–20, 2019, pp. 194–206 (2019)

[56] Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., Stoyanov, V.: RoBERTa: A robustly optimized BERT pretraining approach. In: International Conference on Learning Representations, Addis Ababa, Ethiopia (2020)