Automated Software Engineering (2024) 31:38
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] · **Hao Shen**[1]

Received: 12 January 2024 / Accepted: 4 April 2024

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: 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 three datasets (FFMPeg+Qemu, Reveal, and Big-Vul) indicate that ProRLearn achieves performance improvement of 3.27–70.96% over state-of-the-art baselines in terms of F1 score. 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 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

Extended author information available on the last page of the article

Published online: 20 April 2024

1 Introduction

Software security issues (2020. "The exactis breach: 5 things you need to know."; Nord 2017) 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. The number of disclosed vulnerabilities constantly increases, causing more and more significant concerns in the field of software industry and cybersecurity. The National Vulnerability Database in the USA (NIST, National Vulnerability Database) released 25,096 vulnerabilities in 2022, with an increase of 25% over the last year (Vulnerability and Threat Trends Report 2023). In the interconnected world today, developing accurate automatic vulnerability detection techniques for these threats has become an utmost priority.

Currently, detecting source code vulnerabilities can be divided into two broad categories: traditional vulnerability detection methods (Cherem et al. 2007; Fan et al. 2019; Kroening and Tautschnig 2014; Heine and Lam 2006) and machine learning/deep learning-based vulnerability detection methods (Li et al. 2018, 2021a, b; Zhou et al. 2019; Russell et al. 2018; Lomio et al. 2022). Previous vulnerability detection methods (Cherem et al. 2007; Fan et al. 2019; Kroening and Tautschnig 2014; Heine and Lam 2006) mainly use rules defined in advance by experts to analyze the code. However, such an analysis method (Li et al. 2018, 2021a) cannot easily find some deeply hidden vulnerabilities (Cao et al. 2022; Cheng et al. 2022). Deep learning (DL) has gained widespread usage in recent years for detecting source code vulnerabilities via automatic feature extraction.

Recently, several vulnerability-identifying frameworks (Li et al. 2018, 2021a) utilize DL to detect and learn source code vulnerabilities have been proposed. For example, Devign (Zhou et al. 2019) and ReVeal (Chakraborty et al. 2021) use Graph Neural Network (GNN) (Chakraborty et al. 2021; Li et al. 2021b) on attribute graphs that integrate control flow, data dependencies, and Abstract Syntax Trees (ASTs) (Zhou et al. 2019). VulDeePecker (Li et al. 2018) 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. (2021a) used the Bi-LSTM to detect vulnerabilities. These DL-based techniques intelligently assist developers in programming and improving their developing efficiency. The state-of-the-art DL-based approaches for code intelligence exploit the pre-training and fine-tuning paradigm (Han et al. 2021; Qiu et al. 2020; Raffel et al. 2020; Brown et al. 2020), in which language models are first pre-trained on a large unlabeled text corpus and then fine-tuned on downstream tasks. 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.

Specifically, as shown in Fig. 1a, pre-trained models generally employ Masked Language Modeling (MLM) objectives for the pre-training phase. The input for MLM consists of representations of tokens randomly masked in natural language

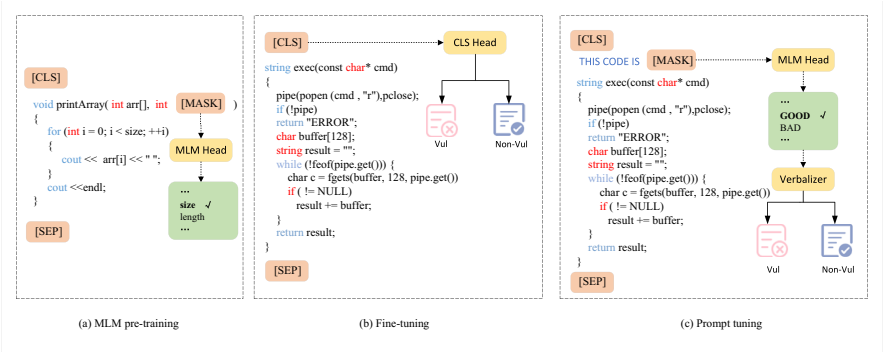

Fig. 1 Illustration on the process of pre-training, fine-tuning, and prompt tuning

text, and the model is trained to predict the masked tokens via the MLM head. However, during the fine-tuning phase for downstream tasks, the input is natural language text, and the training objective shifts to a classification problem. As illustrated in Fig. 1b, the pre-trained model computes representations for input text and predicts labels through the CLS head. The inconsistency between the inputs and objectives in pre-training and fine-tuning makes it challenging to fully leverage the knowledge of the pre-trained model, resulting in sub-optimal outcomes for downstream tasks (Schick and Schütze 2021; Wang et al. 2022). As shown in Fig. 1c, during the prompt tuning phase for downstream tasks, convert them into cloze-style tasks (Nie et al. 2022). This phase is consistent with the pre-training phase to fully utilize the knowledge of the pre-training model. In addition, natural language prompts inserted during the prompt tuning phase can involve knowledge of specific tasks to facilitate adaptation to downstream tasks (Li and Liang 2021; Schick and Schütze 2021).

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, aligning with the pre-training process. Secondly, as the sample data increases (Li et al. 2023), the performance gap between fine-tuning and prompt tuning gradually narrows. To alleviate this limitation, we introduce a novel reward mechanism leveraging policy gradients, inspired by the adaptive learning capabilities of reinforcement learning. This approach allows for continuous improvement in model performance as more data becomes available, as demonstrated in previous studies (Plaat et al. 2023; Arulkumaran et al. 2017).

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 three widely used datasets for vulnerability detection: FFMPeg+Qemu (Zhou et al. 2019), Reveal (Chakraborty et al. 2021), and Big-Vul (Fan et al. 2020). We conducted a comparative experiment between ProRLearn and seven existing software vulnerability detection methods, namely Sysevr, VulDeePecker, IVDetect, Devign (Zhou et al. 2019), Reveal (Chakraborty et al. 2021), AMPLE (Wen et al. 2023) and LineVul (Fu and Tantithamthavorn 2022). The experimental results on three datasets indicate that ProRLearn can improve F1 scores by 3.58%, 4.07%, and 3.27%, respectively, while improving accuracy by 1.13%, 2.02%, and 2.96%, respectively.

The main contributions of this study are as follows:

- 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.
- We compare ProRLearn with seven state-of-the-art baselines. Experimental results show that it outperforms baselines by 3.58%, 4.07%, and 3.27% in F1 score and by 1.13%, 2.02%, and 2.96% in accuracy, respectively.
- We design a set of ablation experiments, which can explore the effectiveness of each component of ProRLearn.
- We also 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 (Li et al. 2018; Russell et al. 2018; Dam et al. 2017; Wang et al. 2022). For example, VulDeePecker (Li et al. 2018) embeds source code through word2vec. Then it is fed into a Bi-LSTM neural network with an attention mechanism. SySeVR (Li et al. 2021a) extracts code statements, program dependencies, and program slices as features, and inputs them into a bidirectional recurrent neural network. LineVul (Fu and Tantithamthavorn 2022) leverages CodeBERT to embed the whole sequence of tokens in a function for vulnerability detection. 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 (Li et al. 2021a, b; Wu et al. 2022; Chakraborty et al. 2021), such as Abstract Syntax Trees (AST), Control Flow Graphs (CFG), Data Flow Graphs (DFG), and Program Dependency Graphs

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

(PDG). Devign (Zhou et al. 2019), IVDetect (Li et al. 2021b), and AMPLE (Wen et al. 2023) utilize graphs (such as AST CFG, DFG, PDG) as inputs and extract information from them or simplify them, ultimately inputting them into a graph neural network for vulnerability detection. 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 (Li et al. 2023; Liu et al. 2023) is a method for pre-trained language models (Liu et al. 2023) 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 (Han et al. 2022). 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, is different 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.

Recently, Wang et al. (2022) conducted an empirical study on applying prompt tuning to code intelligence tasks based on research in the field of NLP. They conducted prompt tuning on popular pre-trained models (such as CodeBERT and CodeT5) and considered three code intelligence tasks, including defect prediction, code summarization, and code translation. Furthermore, Li et al. (2023) applied a combination of vulnerability code description and prompt tuning to vulnerability assessment. Yu et al. (2023) proposed a smart contract slicing method to reduce irrelevant code while combining the sliced code with prompt tuning. These studies have demonstrated the effectiveness of prompt tuning. In our work, we combined RL and prompt tuning to exploit knowledge about programming languages captured by pre-trained models to detect vulnerability.

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 (Sutton

and Barto 1999). Instead, RL (Zeng et al. 2018; Rosenstein et al. 2004; Lagoudakis and Parr 2003; Kaelbling et al. 1996) optimizes its strategy through multiple rounds of environment exploration and experience mining. RL performs well without prior expert knowledge and exhibits the following characteristics (Mnih et al. 2015; Caicedo and Lazebnik 2015; Silver et al. 2016; Sallab et al. 2017; Shao et al. 2019). 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. The value-based method focuses on learning and optimizing the state value function, which measures the quality of taking different actions in different states. Typical representatives of value function methods include Q-learning (Watkins and Dayan 1992), Deep Q-Networks (Osband et al. 2016), 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 (Silver et al. 2014) and deep deterministic policy gradient methods (Qiu et al. 2019) are examples of policy optimization methods.

3 Approach

The overall architecture of ProRLearn is shown in Fig. 2, 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 only considers code, that is without mixing with other aspects(such as comments) and focuses on performing specific tasks (De Luca and Restivo 1980). For the training phase, the reinforcement learning environment is set up. The 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 (Feng et al. 2020) 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 ode 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. 2, there are three main types of noise: blank lines (extra indentation or line breaks),

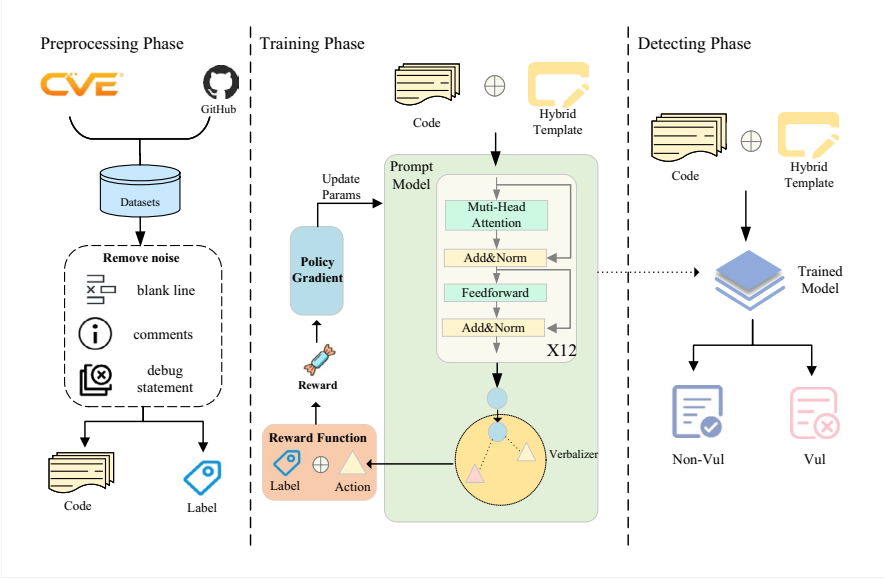

Fig. 2 Overview of ProRLearn

Fig. 3 Code snippets before data processing

```
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 ;
}
```

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 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. 3,

Fig. 4 Code snippets after 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 ;
}
```

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.

Figure 3 illustrates three types of noise in a given code: `excess line breaks`, `comments`, and `irrelevant debugging or output statements`. Figure 4 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.

3.2 Prompt tuning implementation

In our method, predicting and classifying vulnerable code based on prompt tuning primarily involves two steps. Firstly, selecting an appropriate PLM, as different PLMs are suitable for different tasks. Secondly, we apply a combination of prompt tuning (Jiang et al. 2020; Qin and Eisner 2021) to the selected PLM to achieve classification predictions for vulnerable code. Next, we will provide a detailed explanation of these steps.

Carefully selecting the Pretrained Language Model (PLM) for prompt tuning is of utmost importance. We have chosen the CodeBERT model (Feng et al. 2020), an extension of BERT designed specifically for handling source code. BERT is a bidirectional pre-training model based on the transformer architecture (Vaswani et al. 2017), which captures the global bidirectional context by considering the contextual information of each position. Meanwhile, the pre-training process of BERT includes a Masked Language Model, which is similar to our prompt tuning. However, GPT is an autoregressive pre-training model based on transformer architecture. Autoregressive pre-training means that the model generates text gradually, one token at a time, and relies on the previously generated tokens. This approach is commonly applied in text-generation tasks. Therefore, in handling classification tasks, CodeBERT inherits the bidirectional nature of BERT, enabling the model to better understand the overall context of the source code text. CodeBERT, as an extension of the BERT (Devlin et al. 2018) 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 (Schick and Schütze 2021; Li and Liang 2021; Lester et al. 2021; Gu et al. 2022; Han et al. 2022) corpora enhances its understanding of specific tasks. Therefore, we selected the widely used CodeBERT (Feng et al. 2020; Wolf et al. 2019) for the vulnerability classification task in our study.

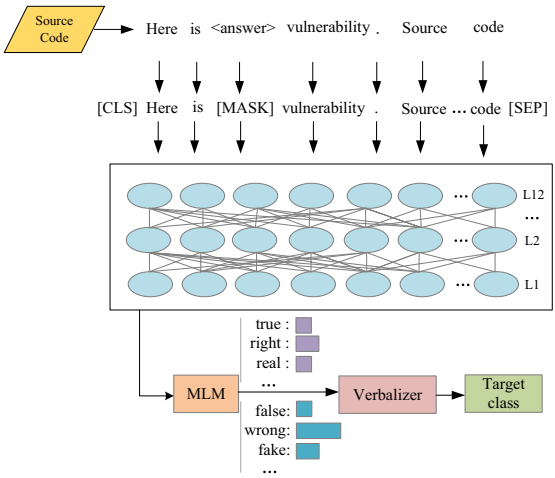

Fig. 5 The model architecture of prompt tuning

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 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 (Schick and Schütze 2021). 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 a novel technique to prompt tuning. It combines the advantages of hard prompts and soft prompts, consisting of hard tokens and soft tokens. Hard tokens are usually task-related important tokens that are not tuned during training, while soft tokens are tunable embeddings that combine the two to generate richer and more diverse results.

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.

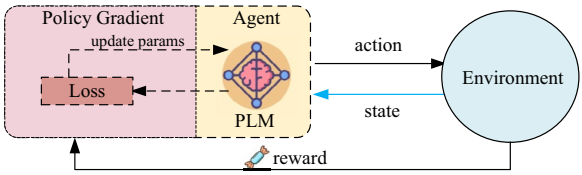

Fig. 6 Agent–environment interaction in RL

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

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

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

$$f = [input]\ \text{[MASK] [MASK] [MASK] [MASK]?}[answer] \tag{2}$$

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.

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

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.

3.3 Reinforcement learning implementation

In our experiments, we did not employ unsupervised methods for prompt tuning the pre-trained model. 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.

In our study, we leveraged reinforcement learning principles to adjust pre-trained models to further improve their performance in vulnerability detection tasks. The

method is based on the following ideas, as shown in Fig. 6. Firstly, we created an RL environment of vulnerability detection comprising vulnerability source codes, their corresponding ground true labels, and prompt template, labels were utilized to define the reward mechanism. Second, we merged the prompt template and code to new input for RL, aiming to provide additional contextual vulnerability information and task guidance for the pre-trained model. This process corresponds to the state in the diagram. Next, The pre-trained model is designed as an Agent in the RL framework, enabling it to interact with the code to learn vulnerability information. In the current task, when the Agent reaches the verbalizer mapping target class stage, we define a reward function. This reward function is based on the accuracy of the mapping, generates reward signals, and encourages the model to generate accurate detection actions for vulnerabilities through the feedback of the reward signals. Finally, based on the current state, action, and reward, we calculated the gradient of the policy and used these gradients to update the parameters of the pre-trained model. This iterative process enables the model to continuously optimize itself through feedback signals from reinforcement learning to better adapt to vulnerability detection tasks. The whole method framework combines the language understanding of pre-trained models and the task orientation nature of reinforcement learning, providing a novel approach to improve vulnerability detection performance. In our experimental study, we further explore different reward functions and prompt templates to optimize model performance.

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 core 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 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 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 in 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 of the vulnerability detection task. To answer this question, we will investigate the effectiveness of using different pre-trained models.

4.2 Datasets

Our research used three vulnerability datasets, including FFMPeg+Qemu (Zhou et al. 2019), Reveal (Chakraborty et al. 2021) and Big-Vul (Fan et al. 2020). The FFMPeg+Qemu dataset is a balanced dataset widely used in previous studies (Wen et al. 2023). 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%. Big-Vul collected C/C++ functions from 348 open-source GitHub projects spanning from 2002 to 2019. This dataset contains approximately 10k vulnerable entries and 177k non-vulnerable entries (i.e., vulnerabilities account for 5.88% of all the entries). Table 1 summarizes dataset characteristics.

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	1664	16,505	9.16
Big-Vul	179,299	10,547	168,752	5.88

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 situations where the model fails to recognize actual 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 retrieved. *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 seven baseline methods: four graph-based and three token-based methods.

(1) *SySeVR* (Li et al. 2021a): 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* (Li et al. 2018): 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* (Li et al. 2021b): 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* (Zhou et al. 2019): 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* (Chakraborty et al. 2021): 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* (Wen et al. 2023): 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.

(7) *LineVul* (Fu and Tantithamthavorn 2022): LineVul uses a transformer to better capture the long-term dependencies in source code. Additionally, the SAC of the transformer model is used to calculate the contribution of each input token to the prediction result, thereby obtaining fine-grained information on vulnerable code lines.

4.5 Experimental settings

For each dataset, we followed the same settings as other experiments (Wen et al. 2023) and randomly partitioned the dataset into disjoint training, validation, and test sets with the ratio of 8:1:1, as this is a typical test setup in previous studies (Fu and Tantithamthavorn 2022; Wen et al. 2023; Hin et al. 2022). Notice we used the same dataset split for all the experiments mentioned. We followed the hyperparameters and dataset split outlined in the original Baseline papers to ensure accuracy and fairness in our experiments. 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. We optimized our model using the Adam optimizer with a batch size of 16 and a learning rate of 2e-5. We incorporated hybrid templates during prompt tuning. We employed the VPG for reinforcement learning with a reward magnitude of 1. We train our model for a maximum of 50 epochs on a server with NVIDIA GeForce RTX 4090 with 20-epoch patience for early stopping.

5 Experimental results

5.1 RQ1: effectiveness of ProRLearn

To demonstrate the effectiveness of ProRLearn, we evaluated its performance. We compared ProRLearn with seven baseline methods on three datasets. The experimental results are presented in Table 2. Based on the results in the table, we draw the following conclusions.

From Table 2, we observe that our proposed method outperforms all the baselines. ProRLearn achieves higher F1 score, recall, and accuracy on three datasets than baselines. Specifically, ProRLearn improves the F1 score by 3.58%, 4.07%, and 3.27%, respectively, compared to the current best baseline method. The corresponding relative improvements are 5.57%, 8.77%, and 3.77% for the F1 score. Additionally, ProRLearn increases the recall score by 3.06% on FFMPeg+Qemu (Zhou et al. 2019), a relative increase of 4.76%, and the recall score on Big-Vul (Fan et al. 2020) increased by 8.43%, a relative increase of 10.12%. Moreover, ProRLearn raises the accuracy score by 1.13%, 2.02%, and 2.96%, respectively, with relative improvements of 1.79%, 2.23%, and 3.09%.

In other words, our 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 two token-based methods (SySeVR, VulDeePecker) in three metrics. However, LineVul and our method performed better on three datasets.

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 method successfully addressed the aforementioned issues, and our research results demonstrate that ProRLearn is more accurate than state-of-the-art methods.

Answer to RQ1: ProRLearn outperforms all baseline methods in terms of accuracy, precision, and F1 score. ProRLearn in F1 scores on three datasets were 3.58%, 4.07%, and 3.27% higher than the best baseline method, respectively.

Table 2 Comparison between ProRLearn and three datasets for copper leakage detection methods

Dataset												
Metrics (%)	FFMPeg+Qemu				Reveal				Big-Vul			
Baseline	Accuracy	Precision	Recall	F1	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	90.10	30.91	14.08	19.34
VulDeePecker	50.12	47.89	33.34	39.31	78.51	20.63	14.59	17.09	81.19	38.44	12.75	19.15
Devign	57.19	52.41	58.11	55.11	86.38	28.98	34.73	31.60	56.25	50.84	70.79	59.24
Reveal	62.73	53.94	71.22	61.39	85.25	29.73	**64.96**	40.79	53.45	48.01	70.07	56.98
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	93.14	29.98	34.58	32.11
LineVul	62.75	**63.98**	50.51	56.45	92.43	48.86	41.35	44.79	95.82	90.68	83.31	86.84
ProRLearn	**64.14**	55.99	**86.27**	**67.91**	**92.74**	**53.18**	48.06	**50.49**	**98.78**	**93.44**	**91.74**	**90.11**

"–" indicates that the method does not apply to the current dataset. The best results for each metric are highlighted in bold

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 three datasets to evaluate the effectiveness of prompt tuning. We conducted 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 difference between these four models lies in input. The model without prompt directly inputs the code and returns the detection results, while the models with hard prompt, soft prompt, or hybrid prompt construct different prompts based on the code and use the verbalizer to return the detection results. The results are presented in Table 3.

Compared to ProRLearn without prompt tuning, the hybrid prompt achieved improvements in F1 scores of 5.28%, 8.59%, and 6.08% on the three datasets. Additionally, the precision score increased by 1.88% on the ReVeal (Chakraborty et al. 2021) and 10.64% on the Big-Vul (Fan et al. 2020). The recall score also demonstrated substantial improvements of 11.94%, 15.22%, and 6.45%. Hard and soft prompts outperformed ProRLearn without prompt learning in terms of F1 and recall scores on the three 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%, 4.68%, and 1.77% on the three datasets, respectively. These results underscore the effectiveness of prompt tuning in improving ProRLearn's performance in vulnerability detection tasks.

From Table 3, we observe that any prompt template enhances ProRLearn's performance. This is because prompt tuning transforms the original classification task into a cloze-style format (Nie et al. 2022), similar to the pre-training phase of the PLM. Prompt tuning enables a more comprehensive and effective utilization of the pre-trained knowledge (Sun et al. 2019) 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 to improving the performance of ProRLearn, with an F1 score improvement of 5.28%, 8.59%, and 6.08% on the three datasets, respectively.

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.

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

Metrics (%)												
Dataset	FFMPeg+Qemu				Reveal				Big-Vul			
Baseline	Accuracy	Precision	Recall	F1	Accuracy	Precision	Recall	F1	Accuracy	Precision	Recall	F1
Non-prompt	58.05	54.11	74.33	62.63	90.98	57.86	32.84	41.90	96.11	82.80	85.29	84.03
Hard-prompt	63.21	53.19	**89.63**	66.77	89.79	**65.60**	35.19	45.81	98.69	91.28	85.58	88.34
Soft-prompt	63.39	54.79	86.25	67.01	91.22	51.72	47.96	49.77	98.34	90.65	92.14	89.21
Hybrid-prompt	**64.14**	**55.99**	86.27	**67.91**	**91.47**	53.18	**48.06**	**50.49**	**98.78**	**93.44**	**91.74**	**90.11**

Bold values indicate that the value is the maximum of the current column

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**
Big-Vul	Non-RL	97.90	83.43	83.74	83.59
	Q-RL	98.23	85.38	83.28	85.38
	PG-RL	**98.78**	**93.44**	**91.74**	**90.11**

Bold values indicate that the value is the maximum of the current column

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 three 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 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%, 7.79%, and 6.52%, while the accuracy increased by 2.6%, 2.59%, and 0.89%, 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 to improving the performance of ProRLearn, with an F1 score improvement of 5.34%, 7.79%, and 6.52% on the three datasets, respectively.

5.4 RQ4: effectiveness of prolearn in different models

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

We use the pre-trained model CodeBERT in ProRLearn. However, we extended our experiments to verify whether this 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, the RoBERTa model occupied

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
Big-Vul	BERT	53.58	15.41	BERT+RL+PT	94.63	46.51
	RoBERTa	56.13	50.71	RoBERTa+RL+PT	97.57	76.01
	CodeT5	62.14	35.77	CodeT5+RL+PT	97.05	71.66
	CodeBERT	64.25	55.64	CodeBERT+RL+PT	97.81	83.96

The original max_len of all pre-trained models has been changed to 256

too much memory. To make a fair comparison, 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. Reducing the input of data information may lead to a decrease in model performance. However, 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 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%, 2.89%, and 31.1%. When we apply the idea in the CodeBERT architecture, the F1 score increases by 4.51%, 4.17%, and 24.32%. When applying the idea in the CodeT5 architecture, the F1 score increased by 6.86%, 2.09%, and 35.89%. When applying the idea in the RoBERTa architecture, the F1 score increased by 4.71%, 6.58%, and 25.3%. 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 three different datasets, and the results showed that the CodeBERT architecture we adopted outperformed other architectures in performance. Furthermore, CodeBERT 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 (Feng et al. 2020), RoBERTa (Liu et al. 2020), CodeT5, and BERT belong to the same model architecture but use different datasets 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 datasets in the training phase, CodeT5 adopts a text-to-text architecture and is more suitable for code annotation or translation tasks.

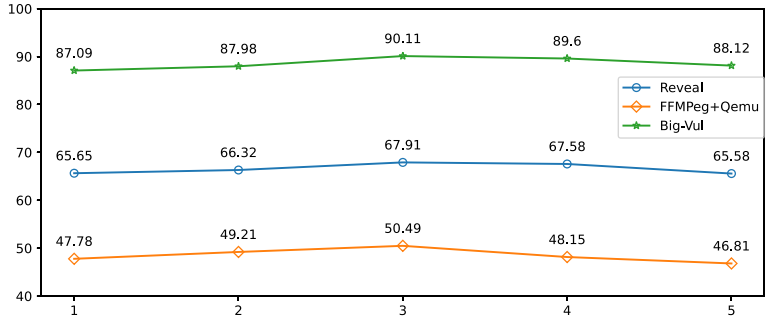

Fig. 7 Comparison of verbalizers based on vulnerability detection

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 results.

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 reflected in Fig. 7. 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 data in Fig. 7, we 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 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.

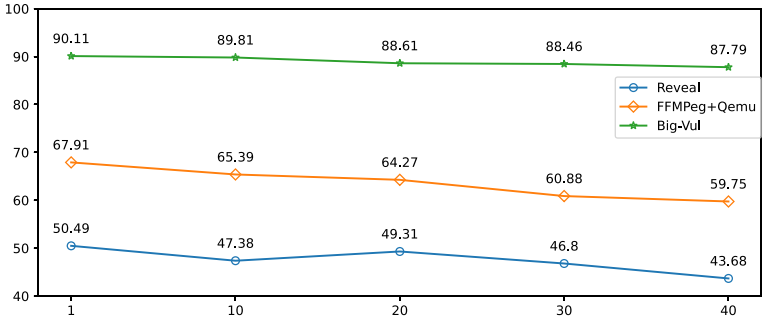

Fig. 8 Performance comparison with different reward sizes

As shown in Fig. 8, we tried 5 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 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.

According to the results in Table 6, we 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.

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

Template		FFMPeg+Qemu		Reveal		Big-Vul	
		Recall	F1	Recall	F1	Recall	F1
H1	Here is [answer] vulnerability. [input]	76.37	63.35	36.63	47.11	85.12	86.55
S1	[Mask] [Mask] [answer] [Mask] [Mask] [input]	83.73	65.20	37.27	47.27	83.75	87.53
D1	[Mask] [Mask] [answer] vulnerability [Mask] [input]	85.16	65.58	43.72	47.98	85.23	87.72
H2	[input] This code is a vulnerability. [answer]	64.37	61.42	36.63	43.16	82.37	82.14
S2	[input] [Mask] [Mask] [Mask] [Mask] [Mask] [Mask] [answer]	66.34	61.93	38.62	43.79	83.74	83.59
D2	[input] [Mask] [Mask] [Mask] [Mask] vulnerability [Mask] [answer]	67.94	63.20	39.37	44.32	83.28	85.38
H3	This code [input] is a vulnerability. [answer]	84.63	65.38	45.89	49.36	84.95	88.02
S3	[Mask] [Mask] [input] [Mask] [Mask] [Mask] [Mask] [answer]	85.27	67.12	47.62	49.77	86.01	89.10
D3	[Mask] [Mask] [input] [Mask] [Mask] vulnerability [Mask] [answer]	**86.27**	**67.91**	**48.06**	**50.49**	**91.74**	**90.11**

Bold values indicate that the value is the maximum of the current column

Table 7 ProRLearn's performance (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 tuning	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 tuning	34.74	40.08	43.22	46.47
	ProRLearn	35.31	41.94	45.37	50.49
Big-Vul	Fine-tuning	76.96	79.30	80.60	82.50
	Prompt tuning	82.28	83.09	83.56	84.47
	ProRLearn	84.84	87.41	86.04	90.11

6.3 How does our model improve performance on dataets 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. 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.

6.4 Evaluation on AUC and MCC performance metrics

In addition to four metrics, we also consider the Area Under the Curve(AUC) and Matthews Correlation Coefficient(MCC). These two metrics are used to evaluate the performance of the model trained on the imbalanced datasets (Tanha et al. 2020). AUC metrics the classification performance of the model at different thresholds, indicating the probability that positive ranks higher than negative across all possible classification thresholds. AUC is the area under the Receiver Operating Characteristic(ROC) curve and its values range from 0 to 1. A higher AUC indicates better model performance, where 0 represents the poorest detection, 0.5 represents the classifier's ability to predict vulnerabilities equivalent to random guessing, and 1 represents the model's perfect detection ability. MCC is a metric used to evaluate the performance of binary classification models. It considers the counts of all four categories (TP, TN, FP, FN) in the confusion matrix and evaluates them comprehensively. The MCC values range from -1 to 1, where 1 represents perfect prediction,

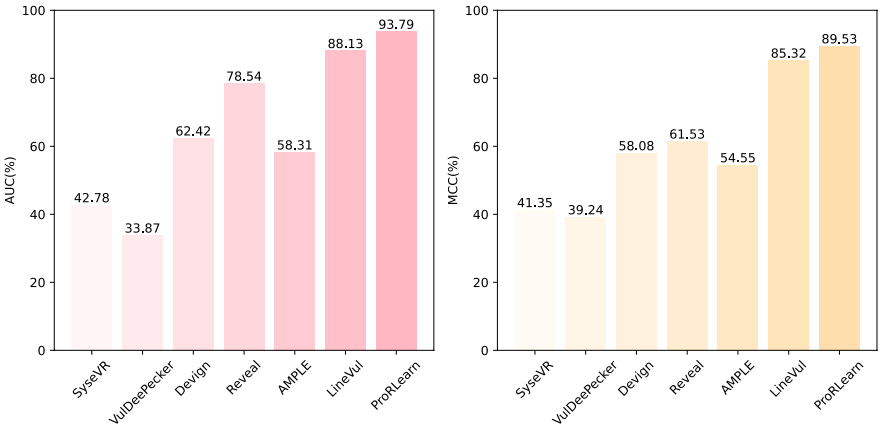

Fig. 9 The comparison results between ProRLearn and six baselines in AUC and MCC

0 represents equivalent to random prediction, and −1 represents opposite prediction. The specific formula is as follows:

$$\text{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{11}$$

Figure 9 shows the performance comparison of ProRLearn with six baselines on the Big-Vul dataset between AUC and MCC. The Big Vul dataset comes from the real world, and the evaluation of this dataset can better reflect the performance of the model in practical application scenarios. In this figure, we find that our proposed approach ProRLearn can outperform the baselines by 4.21% to 50.29% in MCC and 5.66% to 59.92% in AUC. Therefore, the effectiveness of ProRLearn can also be demonstrated in both AUC and MCC performance metrics.

6.5 Statistical analysis on the performance of ProRLearn

We used the Wilcoxon test and Cliff's Delta to compare whether there was a significant difference in F1 scores between ProRLearn and baseline methods on the Big-Vul dataset. The Cliff's Delta is within the range of [−1, 1]. 0 indicates that there is no difference between these two sets of data. It is equal to 1 when all values of one group are higher than the values of the other group and −1 when the reverse is true. When 0.148 ≤ ||d|| 0.33, the effect size is small, when 0.33 ≤ ||d|| 0.474, the effect size is medium, and when ||d|| ≥ 0.474, the effect size is large. As shown in Table 8, we calculated that the p values of the Wilcoxon test were all less than 0.01, and the effect size of Cliff's Delta was all greater than 0.474, indicating a large effect size. The results indicate that the performance improvement of ProRLearn shows significant improvement compared to the six baselines. Furthermore, we employ the McNemar test with Odds Ratios (ORs). An odds ratio of 1 indicates that the detected vulnerability has the same likelihood in both models. An odds ratio greater than 1 indicates that the first model may

Table 8 Analyze the comparison results of ProRLearn and six baselines in terms of F1 metrics

Method	Wilcoxon	Cliff's delta	McNemar	
	P value	Effect size	*P* value	Odds ratio
SySeVR	**	Large	***	735.89
VulDeePecker	**	Large	***	348.10
Devign	**	Large	***	103.73
Reveal	**	Large	***	92.87
AMPLE	**	Large	***	1097.12
LineVul	**	Large	***	1852.29

*** means p value < 0.001, ** means p value < 0.01, * means p value < 0.05

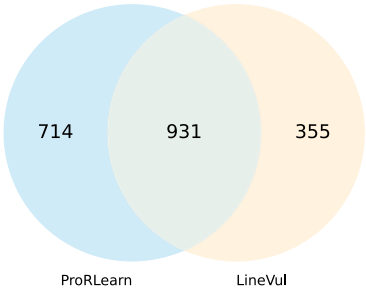

Fig. 10 The complementary analysis of the vulnerabilities correctly detected by ProRLearn via the Venn diagram

detect more vulnerabilities. An odds ratio of less than 1 indicates that the first model may detect fewer vulnerabilities. Under the McNemar test, all *P* values were less than 0.001, indicating statistical significance, while ORs were very large. We can conclude that ProPLearn has achieved a significant performance improvement for detecting vulnerabilities.

In addition, we use Venn diagrams to further analyze the performance of ProRLearn. In this experiment, since the Big-Vul dataset was collected from the real world, we selected 20% of the Big-Vul dataset containing vulnerability samples as the test set (2109 samples) and the rest as the training set to find the number of samples that successfully detected vulnerabilities. Figure 10 shows the detection results, which only show the detection results of ProRLearn and LineVul. The reason is the detection accuracy of the remaining baseline methods on the test set is all below 30%, while the accuracy of these two methods exceeds 60%. Therefore, we only compare the performance of the current best-performing method. In this figure, we can see that both methods can jointly detect 931 vulnerabilities. More importantly, our method can detect an additional 714 vulnerabilities, while LineVul can only detect 355 vulnerabilities. ProRLearn has better performance than baselines.

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 three datasets because these three 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.

Threats to construct validity lie in the suitability of our selected performance metrics. We use common four performance metrics to evaluate vulnerability detection performance since these metrics have been used in previous studies (Fu and Tantithamthavorn 2022; Wen et al. 2023). However, there is a class imbalance issue in the dataset we evaluated, so we also considered additional metrics MCC and AUC to improve the comprehensiveness and rigor of our research.

8 Conclusion and future research

This paper proposes 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 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

2020. The exactis breach: 5 things you need to know. https://blog.infoarmor.com/individuals-and-families/the-exactis-breach-5-things-you-need-to-know

Arulkumaran, K., Deisenroth, M.P., Brundage, M., Bharath, A.A.: Deep reinforcement learning: a brief survey. IEEE Signal Process. Mag. **34**(6), 26–38 (2017)

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A.: Language models are few-shot learners. Adv. Neural. Inf. Process. Syst. **33**, 1877–1901 (2020)

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)

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)

Chakraborty, S., Krishna, R., Ding, Y., Ray, B.: Deep learning based vulnerability detection: are we there yet. IEEE Trans. Softw. Eng. **48**(9), 3280–3296 (2021)

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)

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)

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)

De Luca, A., Restivo, A.: On some properties of very pure codes. Theor. Comput. Sci. **10**(2), 157–170 (1980)

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)

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)

Fan, J., Li, Y., Wang, S., Nguyen, T.N.: A C/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. IEEE (2020)

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

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. IEEE (2022)

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)

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

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

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)

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. IEEE (2022)

Jiang, Z., Xu, F.F., Araki, J., Neubig, G.: How can we know what language models know? Trans. Assoc. Comput. Linguist. **8**, 423–438 (2020)

Kaelbling, L.P., Littman, M.L., Moore, A.W.: Reinforcement learning: a survey. J. Artif. Intell. Res. **4**, 237–285 (1996)

Kroening, D., Tautschnig, M.: Cbmc-c bounded model checker: (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5–13, 2014. Proceedings 20, pp. 389–391. Springer, Berlin (2014)

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)

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)

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)

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)

Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z.: Sysevr: a framework for using deep learning to detect software vulnerabilities. IEEE Trans. Dependable Secure Comput. **19**(4), 2244–2258 (2021a)

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 (2021b)

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. IEEE (2023)

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)

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 Comput. Surv. **55**(9), 1–35 (2023)

Lomio, F., Iannone, E., De Lucia, A., Palomba, F., Lenarduzzi, V.: Just-in-time software vulnerability detection: Are we there yet? J. Syst. Softw. **188**, 111283 (2022)

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

Nie, E., Liang, S., Schmid, H., Schütze, H.: Cross-lingual retrieval augmented prompt for low-resource languages. arXiv e-prints, 2212 (2022)

NIST, National Vulnerability Database. https://nvd.nist.gov/

Nord, R.L.: Software vulnerabilities, defects, and design flaws: a technical debt perspective. In: Fourteenth Annual Acquisition Research Symposium, p. 451. Acquisition Research Program, Boston, USA (2017)

Osband, I., Blundell, C., Pritzel, A., Van Roy, B.: Deep exploration via bootstrapped DQN. In: Advances in Neural Information Processing Systems, vol. 29 (2016)

Plaat, A., Kosters, W., Preuss, M.: High-accuracy model-based reinforcement learning, a survey. Artif. Intell. Rev. **56**(1), 1–33 (2023)

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)

Qiu, C., Hu, Y., Chen, Y., Zeng, B.: Deep deterministic policy gradient (DDPG)-based energy harvesting wireless communications. IEEE Internet Things J. **6**(5), 8577–8588 (2019)
Qiu, X., Sun, T., Xu, Y., Shao, Y., Dai, N., Huang, X.: Pre-trained models for natural language processing: a survey. Sci. China Technol. Sci. **63**(10), 1872–1897 (2020)
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. J. Mach. Learn. Res. **21**(1), 5485–5551 (2020)
Rosenstein, M.T., Barto, A.G., Si, J., Barto, A., Powell, W., Wunsch, D.: Supervised actor-critic reinforcement learning. In: Learning and Approximate Dynamic Programming: Scaling Up to the Real World, pp. 359–380 (2004)
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)
Sallab, A.E., Abdou, M., Perot, E., Yogamani, S.: Deep reinforcement learning framework for autonomous driving. Electron. Imaging **29**(19), 70–76 (2017)
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)
Shao, K., Tang, Z., Zhu, Y., Li, N., Zhao, D.: A survey of deep reinforcement learning in video games. arXiv e-prints, 1912 (2019)
Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., Riedmiller, M.: Deterministic policy gradient algorithms. In: International Conference on Machine Learning, Bejing, China, pp. 387–395. PMLR (2014)
Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M.: Mastering the game of go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016)
Sun, C., Qiu, X., Xu, Y., Huang, X.: How to fine-tune BERT for text classification? In: Chinese Computational Linguistics: 18th China National Conference. CCL 2019, Kunming, China, October 18–20, 2019, Proceedings 18, pp. 194–206. Springer, Cham (2019)
Sutton, R.S., Barto, A.G.: Reinforcement learning: an introduction. Robotica **17**(2), 229–235 (1999)
Tanha, J., Abdi, Y., Samadi, N., Razzaghi, N., Asadpour, M.: Boosting methods for multi-class imbalanced data classification: an experimental review. J. Big Data **7**(1), 1–47 (2020)
Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. In: Advances in Neural Information Processing Systems, vol. 30 (2017)
Vulnerability and Threat Trends Report 2023. https://www.skyboxsecurity.com/resources/report/vulnerability-threat-trends-report-2023/
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)
Watkins, C.J., Dayan, P.: Q-learning. Mach. Learn. **8**, 279–292 (1992)
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)
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)
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, Pennsylvania (2022)
Yu, L., Lu, J., Liu, X., Yang, L., Zhang, F., Ma, J.: Pscvfinder: a prompt-tuning based framework for smart contract vulnerability detection. In: 2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE), pp. 556–567. IEEE (2023)
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)

Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y.: Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Advances in Neural Information Processing Systems, vol. 32 (2019)

Authors and Affiliations

Zilong Ren[1] · Xiaolin Ju[1] · Xiang Chen[1] · Hao Shen[1]

✉ Xiaolin Ju
ju.xl@ntu.edu.cn

✉ Xiang Chen
xchencs@ntu.edu.cn

Zilong Ren
zilongren23@gmail.com

Hao Shen
shenhyc@gmail.com

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