Vulnerability Detection Based on Adapter Tuning and Enhanced Feature Learning

Hui Luo[a,b], Lu Lu[a,c,*], Zhihong Liang[d,e], Siliang Suo[d,e]
[a] School of Computer Science and Engineering, South China University of Technology, Guangzhou, China
[b] Pazhou Laboratory, Guangzhou, China
[c] PengCheng Laboratory, Shenzhen, China
[d] Electric Power Research Institute, CSG, Guangzhou Guangdong, China
[e] Guangdong Provincial Key Laboratory of Power System Network Security, Guangzhou Guangdong, China
[*] Corresponding author email: lul@scut.edu.cn

***Abstract*—Pre-trained code models have achieved promising results in the vulnerability detection field. The prevailing approach is to adapt these models with vulnerability datasets using inefficient full-model fine-tuning. These pre-trained models primarily capture the semantic features of code, while neglecting its structural characteristics. To address these issues, this paper proposes a vulnerability detection method with adapter tuning and enhanced feature learning. First, adapter modules are introduced to UniXcoder and tune only the parameters in the adapters to extract semantic features. This significantly reduces the number of training parameters and better adapts the model to downstream tasks. Then, structural features, including control and data flow information, are extracted from the Program Dependence Graph (PDG) to compensate for the limitations of pre-trained models that rely solely on semantic features. Finally, the semantic and structural features are integrated to train the detection model. The experimental results demonstrate that our proposed method outperforms state-of-the-art approaches and is highly efficient in terms of both training parameters and training data.**

***Index Terms*—Vulnerability Detection, Adapter Tuning, Enhanced Feature Representation, Code Pre-trained Models**

I. Introduction

Deep learning (DL) has been widely applied in industry for detecting both industrial defects and software defects [1]–[3]. With the rise of large pre-trained code models such as CodeBERT [4] and CodeT5 [5], many researchers adopt the *pre-training and finetuning* paradigm to detect vulnerabilities and acquire promising results. However, these approaches still have some limitations.

Firstly, most large pre-trained code models are adapted to vulnerability detection through full-model fine-tuning [6] [7]. However, these pre-trained models have not been trained specifically for vulnerability detection, so full fine-tuning struggles to bridge the gap with vulnerability detection [8]. Moreover, fine-tuning billions of parameters results in high resource consumption and suboptimal trained models on downstream tasks.

Secondly, code pre-trained models focus on the semantic features of the code and ignore the rich structural information in the source code. There is abundant structural information in source code, including data and control dependencies, which have been demonstrated to be an important part of vulnerability features [9]. Many researchers convert code into intermediate representations, such as Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Program Dependence Graph (PDG) [10], and then extract relevant features from these representations for vulnerability detection.

To solve the above problems, we propose ATEFL, a function-level software vulnerability detection method with adapter tuning and enhanced feature learning. To tackle the first issue, ATEFL employs adapter tuning to fine-tune UniXcoder for semantic features extraction. Adapter tuning is one of the parameter-efficient fine-tuning(PEFT) methods and achieves comparable or even superior performance to full-model fine-tuning. To solve the second issue, we extracted data and control flow paths from PDG to offer additional structural features. By integrating semantic and structural information, ATEFL learns more comprehensive features related to vulnerabilities. In summary, the major contributions of our work are as follows:

- To alleviate the resource consumption of fine-tuning pre-trained models for vulnerability detection, we introduce adapter-tuning, significantly reducing the training parameters while achieving a slight performance improvement.
- We propose ATEFL, a vulnerability detection method based on adapter tuning and enhanced features comprised of semantic and structural features.
- We conducted extensive experiments to demonstrate the effectiveness of ATEFL. The experiment results show that ATEFL outperforms many current vulnerability detection methods and can learn rich feature representations even with limited data.

II. Related work

A. DL-Based Software Vulnerability detection

Due to the powerful representation capabilities of deep learning, many DL methods have been employed to automatically learn vulnerability patterns and detect them from historical data.

Dam et al. [1] parse source code into AST sequences and use LSTM networks to learn the vector representations for

* is the corresponding author.

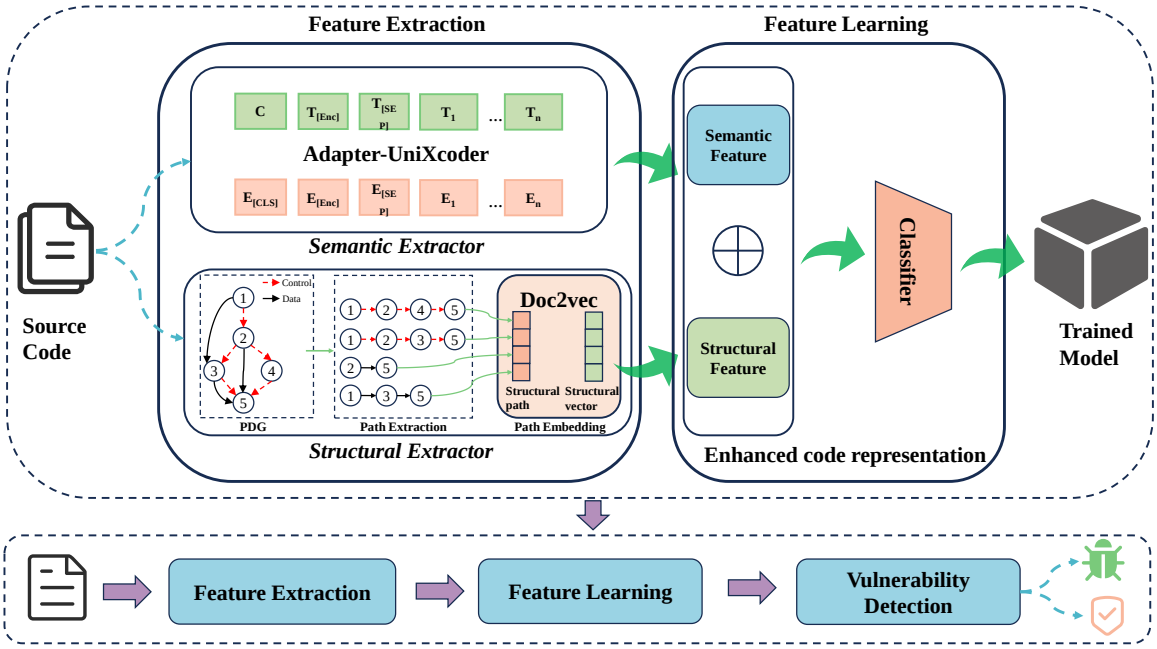

Fig. 1. Overview of Our Proposed ATEFL.

vulnerability detection. However, the method treated source code as flat sequences, overlooking the rich structural and semantic information in code graphs.

Graph Neural Network(GNN) has been demonstrated excellent performance in graph learning and widely applied in software vulnerability detection. Cao et al. [11] propose MVD, which utilizes GNN to learn flow information from both PDG and Call Graph (CG) to detect memory-related vulnerabilities.

Pre-trained code models have achieved significant success in various software engineering tasks, such as code understanding and code generation. Recently, many pre-trained code models have been employed for vulnerability detection. Fu et al. [6] downloaded the pre-trained CodeBERT tokenizer and weights and fine-tune them on the Big-Vul dataset, achieving the state-of-the-art performance. Wang et al. [5] introduce CodeT5, an pre-trained encoder-decoder model. CodeT5 outperformes both CodeBERT and GraphCodeBERT on the devign dataset.

B. Parameter-Effecient Fine-Tuning

Parameter-efficient fine-tuning (PEFT) aims to alleviate the cost and time required for fine-tuning pre-trained models on downstream tasks. Xie et al. Wang et al. [12] introduce adapter tuning to pre-trained code models for code search and summarization tasks. Ayupov [13] study the use of adapter tuning and low-rank adaptation (LoRA) across four code processing tasks and find that these methods achieve better performance than full-model fine-tuning in code understanding tasks.

III. Method

A. Overall Framework

As illustrated in Fig. 1, ATEFL consists of two main components: (i) feature extraction, where semantic features are extracted by applying adapter tuning to the UniXcoder model and structural features are extracted from the paths of the Program Dependence Graph (PDG). (ii) model training and detection, where the two types of features extracted earlier are combined for learning. Once the model has been trained with optimal parameters, it can be employed to detect vulnerabilities in the code.

B. Feature extraction

The purpose of feature extraction is to convert the semantic and structural information of the code into vector representations, which can then be fed into neural networks to learn features related to vulnerabilities.

1) Semantic Feature: Given the success of pre-trained models in various software engineering domains such as code search and code summarization, the code pre-trained model UniXcoder is selected to extract the semantic features of the code. UniXcoder is a cross-modal programming language pre-trained model that has been pre-trained on code and Abstract Syntax Tree (AST), thereby providing a strong understanding of code semantics. It uses prefix adapters and mask attention matrices to control the behavior of the model, including three modes: encoder-only, decoder-only, and encoder-decoder. The encoder-only mode is set to generate semantic vector representation for code.

The semantic feature extractor is shown in Fig. 2, which takes function code as input. The code is tokenized to tokens through the Byte Pair Encoding algorithm. Subsequently, positional embedding and word embedding techniques are employed to obtain the initial vector representations. Then, the initial vector is fed into UniXcoder, which employs 12 Transformer layers to generate hidden layer representations of the input code, denoted as $H^N = h_0^N, h_1^N, ..., h_n - 1^N$. Each

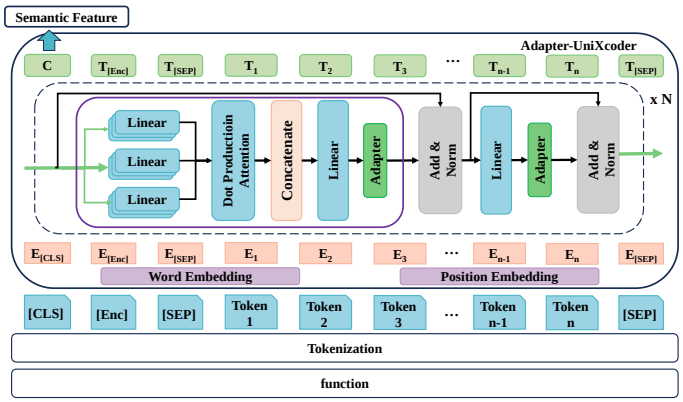

Fig. 2. The Structure of Semantic Extractor.

Transformer layer consists of a multi-headed self-attention layer and a feed-forward layer. The multi-head attention mechanism enables the model to learn various aspects of information from different subspaces of code representations. For Q, K, and V with a dimension of d, these vectors are divided into heads, each with a dimension of d/h. After the self-attention operation, all heads are concatenated back to the original dimension and then fed into the feed-forward layer. For the *i-th* layer, the output of the multi-headed self-attention is computed using the following formula:

$$Q = H^{i-1}W^Q, K = H^{i-1}W^K, V = H^{i-1}W^V \quad (1)$$

$$head_i = softmax(\frac{QK^T}{\sqrt{d_k}} + M)V \quad (2)$$

$$MultiHead = Concat(head_1, ..., head_h)W^O \quad (3)$$

where Q, K, and V are obtained by mapping the hidden layer output from the previous layer, and d_k is the dimension of each head. M is a mask matrix that controls the context a token can attend to. If the *i-th* token can attend to the *j-th* token, then $M_{ij} = 0$; otherwise, $M_{ij} = \infty$. During fine-tuning, the prefix is set to encoder-only mode, allowing all tokens to attend to each other. W^O is used to linearly project to the expected dimension following concatenation.

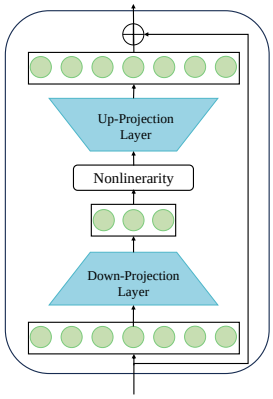

Fig. 3. The Structure of Adapter.

The 125 million parameters of UniXcoder burdens the fine-tuning process and hinders the potential performance on vulnerability detection. Therefore, the adapters are introduced to obtain better vector representations. Fig. 3 illustrates the structure of the adapter. The adapter consists of two projection layers and a non-linear layer, employing residual connections across the adapter. As presented in Fig. 1, for each Transformer layer in UniXcoder, the adapter is inserted after the attention mechanism and the feed-forward layer. During fine-tuning, only the parameters in the adapters are updated while keeping the other pre-trained parameters frozen. Given a hidden input vector , the output of the adapter is:

$$output_{adapter} = W_{up}(\alpha(W_{down}h)) + h \quad (4)$$

where α is a non-linear activation function, $W_{up} \in R^{d \times m}$, $W_{down} \in R^{m \times d}$ are the parameters of the two projection layers, d is the dimension of the transformer hidden layer, and m is the dimension of the adapter layer. Typically, m is smaller than d.

2) Structual Feature: UniXcoder is used to capture the semantic information of code, but it does not utilize the structural information in the source code. Structural information is a crucial part of the source code, typically containing data dependencies and control dependencies between statements. PDG is introduced to extract the rich structural information from the source code. The PDG of a function can be denoted as $g = (V, E)$, where V denotes a set of nodes corresponding to statements in the function, and E represents the data or control dependencies between the nodes.

The Joern is used to convert functions into their corresponding PDGs. Inspired by vagavolu [14], we extract node paths from the PDG as the structural features, including data dependency paths and control dependency paths. Each node in the PDG converted by Joern is represented by two attributes: node type d and node code token t. A path is represented as a sequence of nodes, where all edges on this path have the same label. Specifically, paths with data dependency labels are classified as data dependency paths, while those with control dependency labels are classified as control dependency paths. The path representation is a sequence $p_i = (t_1, l_1, t_2, l_2, ..., t_k, l_k)$, where t_i is the type of node i, $l_i \in (data, control)$ represents the type of edge between node i and node $i-1$. In this section, the structural features of a function are represented as $S = (s_1, s_2, ..., s_m)$, where $s_i = (s, p, e)$ represents a data flow path or control flow path between node s and node e. After obtaining the structural features of a function, they need to be converted into vector representations for input into neural networks. Since the number and length of extracted paths from each function vary, we use Doc2Vec to convert the structural features of each function into vector representations.

C. Training

In this section, the semantic and structural features are trained together to effectively learn the combined feature representation of code.

The structural feature vectors obtained through Doc2Vec do not have the same dimension as the semantic feature vectors generated by UniXcoder, so these vectors are aligned to the same dimension. Specifically, for each function, the semantic feature vector output by adapter-tuned UniXcoder is $V_{semantic} \in R^{512\times768}$. In classification tasks, the final hidden layer representation of the [CLS] token is taken as the semantic feature of the input code. It has learned the semantics of all tokens through the attention mechanism. The structural feature vector generated by Doc2Vec is denoted as $V_{structual} \in R^{512}$. To prevent the semantic vectors from having greater weight than the structural vectors due to the difference in length during training, the structural feature is mapped into R^{768} by a linear layer. The final feature vector is $V = concat(V_{semantic}, V_{structual}) \in R^{1536}$.

After learning the composite features of the code, the final step is to train a vulnerability detection model. First, the learned composite features are fed into a multi-layer perceptron (MLP) for binary classification task. The model is iteratively trained on the training set, monitoring the loss function and optimizing the feature weights through backpropagation. The optimization continues for several epochs, and early stopping is employed during training to avoid overfitting. For the loss function, since vulnerability detection is a binary classification task, the binary cross-entropy loss function is used:

$$L_{ce} = -y_i \log(p_i) + (1 - y_i) \log(1 - p_i) \tag{5}$$

where y_i represents the true label, p_i is the predicted probability.

Secondly, many dropout layers are used in the pre-trained model to prevent overfitting, resulting in slight variations in the output for the same input. Therefore, KL divergence is applied to penalise the model for inconsistent outputs given the same input.

$$KL(P \parallel Q) = \sum_{i=1}^{N} P(x_i) \log \frac{P(x_i)}{Q(x_i)} \tag{6}$$

where P and Q represent two probability distributions.

The KL-loss is computed as follows:

$$L_{kl} = \frac{1}{2}[KL(P_1 \parallel P_2) + KL(P_2 \parallel P_1)] \tag{7}$$

where P_1 and P_2 are the two output probability distributions of the same input.

The final loss function is denoted as:

$$Loss = L_{ce} + \beta L_{kl} \tag{8}$$

After obtaining the trained model, it can used for vulnerability detection.

IV. Experiment Design

A. Datasets

The benchmark dataset used in the experiments is the FFmpeg+Qemu dataset provided by Zhou et al [9], which is derived from two real-world projects FFmpeg and Qemu. It contains 27k C language functions, with a defect rate of 45%. The dataset is divided into training, validation, and test sets with a ratio of 8:1:1.

B. Experimental Setting

The code is implemented in Pytorch. The Unixcoder model architecture and pre-trained weights are loaded throughout the Transformers library. Only the parameters in the adapters are adjusted during fine-tuning while other pre-trained weights are frozen. In code structural feature extraction, code is transformed to PDG by Joern, and Scikit-learn is used to train doc2vec to obtain the corresponding structural feature vectors. All models were trained on an AMD210 with 64GB of VRAM. The dimension of the adapter is set to 96. During training, we used the AdamW optimizer with a batch size 32. The learning rate is 3e-4. The maximum number of epochs was set to 30, and early stopping was employed to obtain the best model. The model with the best F1-score on the validation set was saved for testing.

C. Evaluation Indicators

To comprehensively evaluate the performance of our method in vulnerability detection, we use accuracy, precision, recall, and F1-score as evaluation metrics. Precision measures how many of the functions predicted to be defective are defective; recall indicates how many of the actual vulnerabilities are correctly classified; and F1-score is often used as a comprehensive measure of model performance. The formulas for these metrics are as follows:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{9}$$

$$Precision = \frac{TP}{TP + FP} \tag{10}$$

$$Recall = \frac{TP}{TP + FN} \tag{11}$$

$$F1 - score = \frac{2 \times Precision \times Recall}{Precison + Recall} \tag{12}$$

D. Research Questions

To evaluate ATEFL, the experiments focus on the following research questions:

RQ1: *How effective is ATEFL in detecting vulnerabilities at the function level?*

RQ2: *What is the effect of the adapter and structural features on the performance of ATEFL?*

RQ3: *How does ATEFL perform under low-data conditions?*

To answer **RQ1**, the following vulnerability detection baselines are compared with ATEFL:

1) **LineVul** [6] is a transformer-based vulnerability prediction method. It leverages the pre-trained model CodeBERT to capture long-term dependencies in code sequences, enabling it to learn rich semantic features from source code.

2) **CodeT5** [5] is a variant of the text-to-text transfer transformer, which has been pre-trained on large code corpora, including CodeSearchNet and additional C language datasets. CodeT5-base with 220M parameters is selected as the baseline.

3) **AMPLE** proposed by Wen et al [15] simplifies code structure graphs and uses an edge-aware GCN module along with a multi-head attention mechanism to learn long-range dependencies in code structure graphs.

4) **Devign** proposed by Zhou et al [9] constructs a composite code property graph with AST, CFG, and DFG. It employs a Gated Graph Neural Network to extract structural features from these rich code representations.

Additionally, ablation experiments are conducted to investigate the impact of each component on ATEFL. To answer **RQ2**, the follow models are set up:

1) ATEFL *wo A and S*: The method using full-finetune and semantic features only.

2) ATEFL *wo S*: The method uses adapter tuning but only uses the semantic features from UniXcoder.

3) ATEFL *wo A*: The method using full fine-tuning instead of adapter-tuning to fine-tune the model with semantic and structural features.

4) ATEFL: The method with adpater tuning and semantic and structural features.

Finally, the use of pre-trained models for vulnerability detection typically requires a large number of training samples to achieve good performance. **RQ3** is set up to evaluate the performance of ATEFL under low-data conditions. ATEFL is trained on subsets comprising 1%, 10%, 20%, 30%, and 50% of the training data from RQ1. The trained model is then tested on the RQ1 test set and compared with LineVul and CodeT5.

V. Experiment Result

TABLE I
RESULTS VERSUS BASELINE METHODS

method	Acc	F1-score	Precision	Recall
LineVul	66.69	61.83	63.93	59.86
CodeT5-base	62.72	66.50	55.88	82.11
AMPLE	61.28	65.16	53.36	79.82
Devign	53.37	58.29	51.38	67.35
ATEFL	**88.75**	**87.45**	**87.99**	**86.91**

RQ1: *To what extent can the function-level vulnerability detection performance ATEFL achieve?*

The experiment results are reported in Table I and the best performances are highlighted in bold. It is found that ATEFL outperforms all the baselines in all metrics. Specifically, the acc, F1-score, recall, and precision of ATEFL are 88.75%, 87.45%, 87.99%, and 86.91%, respectively. ATEFL relatively improves over the baselines from 33.08%-66.29% in acc, from 31.5%-50% in F1-score, from 37.63%-71.25% in Precision, and from 5.8%-52.84% in Recall. The results indicate that ATEFL outperforms other methods based on graph neural networks and pre-trained models. This is because graph neural networks typically only use structural features from the code graph representations, while pre-trained models only leverage the semantic features of the code. Neither approach can fully capture the rich features related to vulnerabilities in the code.

TABLE II
PERFORMANCE COMPARISION FOR DIVERSE VERSIONS OF ATEFL

method	Acc	F1-score	Precision	Recall
ATEFL *wo A and S*	64.41	62.21	59.65	65.01
ATEFL *wo S*	65.12	64.82	59.42	71.29
ATEFL *wo A*	70.27	69.34	64.77	74.61
ATEFL	88.12	86.96	87.87	86.07

TABLE III
F1-SCORE COMPARISON OF LINEVUL, CODET5, AND ATEFL UNDER DIFFERENT DATA PORTIONS.

Portion	data	F1		
		LineVul	CodeT5	ATEFL
1%	203	54.22	53.34	54.63
10%	2034	55.45 *(+1.23)*	56.73 *(+3.39)*	**66.18** *(+11.55)*
20%	4068	57.37 *(+1.92)*	58.08 *(+1.35)*	**72.10** *(+5.92)*
30%	6102	61.26 *(+3.89)*	61.28 *(+3.2)*	**77.33** *(+5.23)*
50%	10170	61.52 *(+0.26)*	62.50 *(+1.22)*	**82.85** *(+5.52)*

RQ2: *How do the adapter, and structural features perform in our method?*

Table II reports the performance of four versions of ATEFL on FFmpeg+Qemu. The baseline is $\text{ATEFL}_{\text{wo A and S}}$, which only relies semantic features extracted from UniXcoder throughout full fine-tuning. In the training phase, $\text{ATEFL}_{\text{wo A and S}}$ needs to update about 127M parameters.

Compared with $\text{ATEFL}_{\text{wo A and S}}$, $\text{ATEFL}_{\text{wo S}}$ achieves an absolute improvement of 0.71%, 2.61%, and 6.28% in accuracy, F1-score, and recall. It should be noted that the training parameters of the model are only 3.67M, which is 2.89% of the UniXcoder. The results demonstrate the efficiency of applying adapter tuning in vulnerability detection. It significantly reduces the number of training parameters, saves computational resources, and achieves superior performance.

To evaluate the contribution of structural features, $\text{ATEFL}_{\text{wo A}}$ integrates structural and semantic features for vulnerability detection. It improves the baseline by 5.86%, 7.13%, 12%, and 9.6% in terms of accuracy, F1-score, precision, and recall, respectively. The improvements indicate the effectiveness of structural features in vulnerability detection, which contain critical information related to vulnerabilities and greatly improve the performance of the model.

Comparing ATEFL with the previous three models, combining the adapter and structural features has better performance than adding them separately. This suggests that full fine-tuning may hinder the effectiveness of structural features, which may be attributed to the huge number of parameters updated during full fine-tuning. In cases of insufficient data, full fine-tuning may fail to adequately learn optimal feature representations in the specific dataset. It primarily leverages the prior knowledge from pre-training, which is not readily transferable to downstream tasks. In contrast, training only

the adapter parameters enables the model to effectively learn the specific knowledge of the studied dataset.

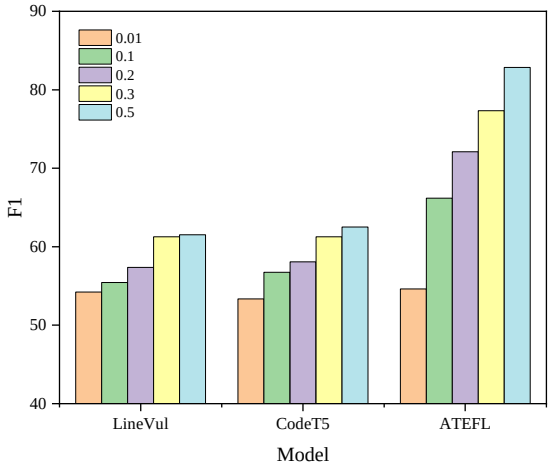

Fig. 4. F1 Comparision for LineVul, CodeT5 and ATEFL on limited data

RQ3: *How does ATEFL perform with a limited amount of data?*

Table 3 presents the F1-score of ATEFL on the reduced training set compared with state-of-the-art methods, LineVul and CodeT5. The value in parentheses following the F1-score indicates an improvement in comparision with the F1-score of the previous line. The column 'data' indicates the number of samples in the training subsets. Fig. 4 presents the performance of ATEFL over reduced training dataset sizes compared to LineVul and CodeT5. It can be observed that ATEFL outperforms LineVul and CodeT5 across all portions. Additionally, ATEFL demonstrates the highest data efficiency. Using the 1% portion dataset as the baseline for all methods, the F1 scores for Linevul, CodeT5, and ATEFL are 54.22%, 53.34%, and 54.63%, respectively. As the size of dataset increases to 10%, the F1 scores of LineVul and CodeT5 improve by 1.3% and 3.4%, respectively, while ATEFL shows an 11.6% improvement. As the portion of data increases from 10% to 50%, ATEFL achieves an average improvement of 5.5%, whereas LineVul and CodeT5 show improvements of only 2% and 1.9%, respectively.

VI. Conclusion

This paper proposes ATEFL, a novel vulnerability detection framework with adapter tuning and enhanced feature learning. ATEFL leverages adapter tuning to fine-tune UniXcoder for extracting semantic features from the code and integrates the structural features from the PDG for joint learning in vulnerability detection. Extensive experiments demonstrate that the model exhibits superior performance and high efficiency. It outperforms many state-of-the-art methods and showcases efficiency in terms of computational resources and training data utilization. Even in a low-data scenario, ATEFL maintains satisfactory performance.

In the future, we will explore more PEFT methods for various code-related tasks. Incorporating code structural features into pre-trained models also remains a key research direction.

Acknowledgement

This work is supported in part by the Natural Science Foundation of Guangdong Province (NO. 2024A1515010204), the second batch of cultivation projects of Pazhou Laboratory (NO. PZL2022KF0008) and Southern Power Grid Science and Technology Project: Research on the construction method of AI trusted model and performance and safety evaluation technology (NO. ZBKJXM20232483).

References

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

[2] F. Zhou, H. Zhang, M. Zhe, G. Wen, H. Pan, Z. Lan, and Z. Zhang, "A power transmission line and its defect detection method based on data enhancement,augmentation and neural network," *Southern Power System Technology*, vol. 16, no. 09, pp. 131–142, 2022.

[3] P. Li, R. Liu, J. Zhou, and T. Zhao, "X-ray image intelligent recognition of crimping defects of strain clamps based on deep learning," *Southern Power System Technology*, vol. 16, no. 03, pp. 126–133, 2022.

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

[5] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 8696–8708.

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

[7] S. Liu, B. Wu, X. Xie, G. Meng, and Y. Liu, "Contrabert: Enhancing code pre-trained models via contrastive learning," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2476–2487.

[8] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, "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*, 2022, pp. 382–394.

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

[10] Y. Mirsky, G. Macon, M. Brown, C. Yagemann, M. Pruett, E. Downing, S. Mertoguno, and W. Lee, "Vulchecker: graph-based vulnerability localization in source code," in *Proceedings of the 32nd USENIX Conference on Security Symposium*, 2023, pp. 6557–6574.

[11] S. Cao, X. Sun, L. Bo, R. Wu, B. Li, and C. Tao, "Mvd: memory-related vulnerability detection based on flow-sensitive graph neural networks," in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 1456–1468.

[12] D. Wang, B. Chen, S. Li, W. Luo, S. Peng, W. Dong, and X. Liao, "One adapter for all programming languages? adapter tuning for code search and summarization," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 5–16.

[13] S. Ayupov and N. Chirkova, "Parameter-efficient finetuning of transformers for source code," *arXiv preprint arXiv:2212.05901*, 2022.

[14] D. Vagavolu, K. C. Swarna, and S. Chimalakonda, "A mocktail of source code representations," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 1296–1300.

[15] X.-C. Wen, Y. Chen, C. Gao, H. Zhang, J. M. Zhang, and Q. Liao, "Vulnerability detection with graph simplification and enhanced graph representation learning," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2275–2286.