Proceedings of ICSP2024

Detecting Source Code Vulnerabilities Using Fine-Tuned Pre-Trained LLMs

Jin Zhu, Hui Ge, Yun Zhou, Xiao Jin, Rui Luo, Yanchen Sun
China Aerospace Academy of Systems Science and Engineering, Beijing, China
Email: z1004176113@163.com, geh@spacechina.com, cloudc0201@gmail.com, jinxiao@spacechina.com, vonbravochevic@gmail.com, sunchenyan@126.com

2024 IEEE 17th International Conference on Signal Processing (ICSP) | 979-8-3503-8738-4/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICSP62129.2024.10846595

***Abstract*—Pre-trained large language models (LLMs) have advanced capabilities in feature extraction and pattern discovery. Utilizing fine-tuning techniques effectively adapts LLMs to specific scenarios. Detecting vulnerabilities during the coding phase is crucial. In this paper, we collected the Java CWE vulnerability from the SARD dataset and then supervised fine-tuned the open-source Qwen2-7B model using the LoRA technique. We compared the results with vulnerability detection models based on Graph Neural Networks (GNN) and Long Short-Term Memory (LSTM). It is demonstrated that the approach of fine-tuning for LLMs can effectively detect source code vulnerabilities.**

***Index Terms*—Large Language Models, Supervised Fine-Tuning, Vulnerability detection**

I. Introduction

Potential software vulnerabilities can pose serious threats to software security and cause significant economic losses. Static analysis can discover vulnerabilities in programs during the coding phase. Early static vulnerability detection methods were mainly based on rule-based vulnerability analysis [1], code similarity detection [2, 3], and symbolic execution, among others [4–6]. Their detection precision is usually low.

In recent years, with the advancement of computing power, deep learning has experienced rapid development. Researchers have proposed intelligent static vulnerability detection methods based on code metrics [7, 8] and code patterns [9]. Source code vulnerability detection methods based on deep learning technology typically represent code as abstract structures such as Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and others, then embed these abstract structures into high-dimensional vectors. By learning the features of these structures through deep learning networks, the final step is to utilize the learned model to detect vulnerabilities. These methods achieve higher detection accuracy than traditional static analysis, but they require the design of specialized models for specific vulnerabilities to improve detection precision, which can result in lower detection efficiency.

With the emergence of ChatGPT-3.5, large language models (LLMs) have led an artificial intelligence revolution. LLMs possess powerful feature extraction capabilities, and applying pre-trained LLMs to specific domains holds significant potential for development. Fine-tuning techniques act as a bridge connecting LLMs to specific application scenarios. Fine-tuning avoids the enormous cost of training LLMs from scratch and allows pre-trained LLMs to be applied in specific domains by adjusting the model on a particular dataset [10]. The goal of this paper is to explore the effectiveness of open-source LLMs in the field of source code vulnerability detection after fine-tuning on vulnerability datasets. The contributions of this paper are as follows:

- We propose a novel vulnerability detection method using fine-tuned LLMs to detect source code vulnerabilities. Compared to deep learning-based vulnerability detection methods, using fine-tuned LLMs does not require designing specific models for individual vulnerabilities, resulting in higher efficiency in vulnerability detection.
- We fine-tuned open-source LLMs using publicly available CWE vulnerability datasets and provided a paradigm for fine-tuning. The final training results of the models demonstrate that this training paradigm is feasible.
- We compared the fine-tuned LLMs with vulnerability detection methods based on deep learning models, LSTM and GNN, and ultimately proved that fine-tuned LLMs have certain advantages in terms of both efficiency and precision in vulnerability detection.

II. Background

A. Source Code Vulnerability Detection

Currently, mainstream source code vulnerability detection methods can be divided into two categories: those based on code metrics and those based on code patterns. Both of these vulnerability detection approaches involve training specific models for individual vulnerabilities. The process begins by obtaining training samples through data mining, which consist of code snippets contain the specific vulnerability or not. Subsequently, code features are extracted using specific methods. Features extracted by code metrics-based methods include lines of code, number of functions, cyclomatic complexity and code patterns-based methods include Abstract Syntax Trees (ASTs) and Control Flow Graphs (CFGs). These features are used as input for machine learning models, which are trained to produce models tailored to detect specific vulnerabilities, thereby identifying potential vulnerabilities.

B. Large Language Models

The development of large language models (LLMs) can be traced back to statistical machine translation in the 1990s when researchers began to experiment with neural networks for language processing. With the introduction of deep learning theories and the increase in computing power, LLMs began

to gain prominence. In recent years, LLMs represented by the Transformer architecture have made significant progress, such as OpenAI's GPT series [11] and Google's BERT [12]. The core advantage of LLMs lies in their ability to process long texts, capture complex semantic relationships, and achieve cross-lingual universal representation. Open-source LLMs allow researchers and developers to freely use, modify, and extend these models, greatly promoting the development of the NLP field. To date, many research institutions and companies have released their own open-source LLMs, such as Tsinghua University's ChatGLM [13], Meta AI's LLaMA [14], Alibaba's Qwen [15], and others.

C. Supervised Fine-Tuning for LLMs

Fine-tuning of large language models generally falls into two categories: full parameter fine-tuning and parameter-efficient fine-tuning (PEFT). Existing large language models often contain tens of billions of parameters, making full parameter fine-tuning very challenging. Consequently, PEFT has gained popularity and application. PEFT methods mainly consist of three types: addition-based, specification-based, and reparameterization-based. The representative methods of these three types are as follows:

- Addition-based: Adapter-tuning [17], Prefix-tuning [18].
- Specification-based: BitFit [19], Diff-pruning [20].
- Reparameterization-based: LoRA [21], LongLoRa [22].

In this paper, we use LoRA, which is one of the most widely used methods for fine-tuning large models, to fine-tune the model. Let the model weights be $W_0 = R^{d\times k}$. We represent the weight updates via low-rank decomposition [23] as follows:

$$W_0 + \triangle W = W_0 + BA \tag{1}$$

In this equation, $A, B \in R^{d\times r}, r << min(d, k)$. During the training process, only A and B are trained. The model's forward propagation process is as follows:

$$h = W_0 x + \triangle W x = W_0 x + \triangle BAx \tag{2}$$

III. Process Data

A. Vulnerability Datasets

The dataset used in this paper is derived from the Software Assurance Reference Dataset (SARD) [24], which is maintained by the National Institute of Standards and Technology (NIST) in the United States. SARD collects various vulnerabilities according to their CWE (Common Weakness Enumeration) numbers, with CVE-ID (Common Vulnerabilities and Exposures ID) as the unique identifier for each vulnerability, allowing researchers to download and study them. Each vulnerability in the downloaded dataset contains multiple test cases folders, and each test case includes multiple CWE Java source code files. These source code files contain both "good" functions and "bad" functions. During data processing, these functions need to be extracted, with "good" functions labeled as 0 and "bad" functions labeled as their CWE-ID. The CWE vulnerability types used in this paper are as shown in Table I:

TABLE I
Vulnerability Type Information

Type	description	number of test cases
CWE-89	SQL Injection	2156
CWE-113	HTTP Response Splitting	2156
CWE-134	Uncontrolled Format String	784
CWE-470	Unsafe Reflection	720
CWE-789	Uncontrolled Memory Allocation	2537

B. Constructing the Fine-Tuning Datasets

Once vulnerabilities are labeled, the data needs to be processed into a format suitable for LoRA fine-tuning. First, for each piece of data, define a maximum length parameter MAX_LENGTH to control the length of the input sequence. Next, use a tokenizer to process the input and output data. The specific steps are as follows:

1) Instruction Generation: Create a system instruction that describes the role and task of the LLMs——to identify potential vulnerabilities in Java code and output their types. This instruction is processed by the tokenizer to generate the corresponding input IDs and attention masks.
2) Response Processing: Process the response part of the LLMs (i.e., the predicted vulnerability type) through the tokenizer to generate input IDs and attention masks for the response part.
3) Sequence Concatenation: Concatenate the input IDs and attention masks of the instruction part with the input IDs and attention masks of the response part. Additionally, to train the LLMs' output, use the input IDs of the response part as labels, and employ a special token (such as -100) in the instruction part to disregard the loss calculation for these sections.
4) Truncation Processing: If the length of the concatenated sequence exceeds the predefined maximum length MAX_LENGTH, truncate the sequence to ensure that the input data meets the LLMs' input requirements.

Compared to deep learning-based vulnerability detection methods, fine-tuned LLMs do not require training a specific model for each vulnerability when predicting vulnerabilities. Instead, a single model is used to predict any type of vulnerability. The rationale behind this approach is twofold: on one hand, the cost of training and deploying LLMs is significantly higher than that of traditional deep learning models; on the other hand, LLMs have strong feature extraction capabilities, which allow them to maintain high accuracy even when predicting various types of vulnerabilities.

In June 2024, Alibaba released the Qwen2 open-source LLM [16], which achieved leading results on multiple evaluation benchmarks, with greatly enhanced code and mathematical capabilities, and Chinese language skills at the forefront. This paper uses the Qwen2-7b model as the base for fine-tuning.

IV. Result and Conclusion

A. Hardware and Parameters

The GPU used for training is a Tesla V100-PCIE-32GB, and for inference, an Nvidia A10-24G is used. The operating system is Ubuntu 22.04.4 LTS, with a CPU that has 52 cores and 214GB of memory. The LoRA adaptation employs a rank of 8, a scaling factor of 32, and a dropout rate of 0.1. The training parameters involve a batch size of 4, a gradient accumulation steps of 4, and a total of 2 training epochs.

B. Baseline Models

In this paper, a bidirectional Long Short-Term Memory (BiLSTM) Network and Graph Attention Network (GAT) are used as the baseline model. The training parameters of the BiLSTM in this paper are as follows: input dim is set to 200, the embedding dimension is 128, the number of layers in the bidirectional structure network is 10 and dropout is 0.5. The GAT consists of 2 layers, dropout is 0.5. The hidden layer dimensions for input and output are 128 and 64, respectively, followed by another hidden layer with dimensions of 64 and 32, respectively. The input and output dimensions of linear layer are 32 and 2 respectively. The input to GAT is the Abstract Syntax Tree (AST) of the code.

C. Experimental Results and Analysis

The loss function curve and accuracy at each checkpoint during the training process are shown in the Fig. 1.

Methods based on deep learning are typically designed for specific vulnerabilities, and multiple models often work together to detect vulnerabilities in the real world. Therefore, when designing experiments, we use the average accuracy of deep learning models as a benchmark for comparison with LLMs. Let the set of vulnerabilities be:

$$S = \{CWE89, CWE113, CWE134, CWE470, CWE789\}$$

$Accuracy(i)$ is the accuracy for S_i where $i \in S$. The average accuracy $AvgAC$ is defined as:

$$AvgAC = \frac{1}{|S|} \sum_{i}^{|S|} Accuracy(i) \tag{3}$$

The final results are presented in Table II.

TABLE II
Average Accuracy

Model	Accuracy
BiLSTM	0.8339
AST+GAT	0.7692
LLMs	**0.9144**

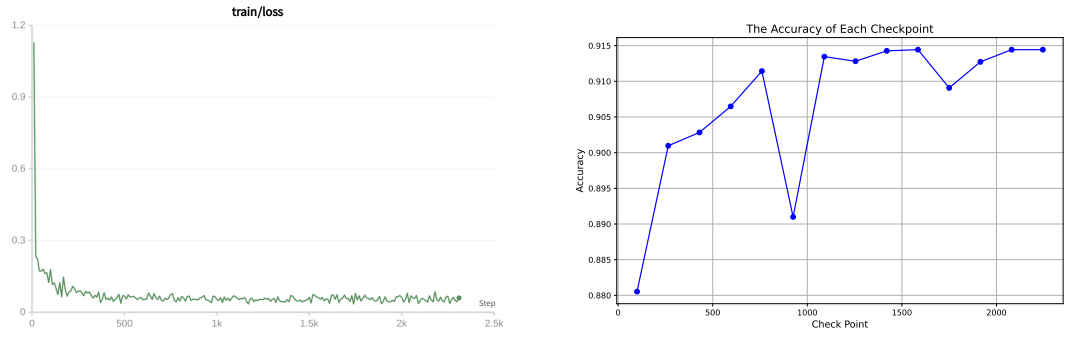
By observing the loss function curve, it is evident that the model's convergence performance is good. Upon examining the Checkpoint curve, it is noted that there is a sudden drop in accuracy at the sixth and eleventh points, but overall, the accuracy is continuously rising. The final results indicate that the average accuracy achieved using LLMs is higher than that of BiLSTM and GAT+AST.

D. Conclusion

Based on the results, it appears that fine-tuned LLMs can converge during the training process, and the final outcomes are also exceptionally good. The overall accuracy of using fine-tuned LLMs to predict vulnerabilities in Java source code is significantly higher than that of directly using conversational tools like ChatGPT. Additionally, the overall accuracy is also higher than that of deep learning models such as BiLSTM and GAT. Therefore, the ultimate conclusion is that the paradigm of using fine-tuning combined with pre-trained models can effectively detect vulnerabilities in source code. In the future, we will further explore methods that integrate pre-trained LLMs with deep learning models for source code vulnerability detection, aiming to enhance the accuracy and efficiency of vulnerability detection.

V. Related Works

There are numerous research achievements in source code vulnerability detection techniques based on deep learning. Russell et al. [25] analyzed the input source code into a variable-length token sequence, embedded it into a variable-length representation, and after passing it through n layers of CNN, input it into a random forest for vulnerability detection. ZHOU et al. [26] performed graph embedding and then used



(a) loss function

(b) checkpoint

Fig. 1. Loss function curve and accuracy at each checkpoint

the Relu function and full pooling layers for processing before applying a convolutional layer for vulnerability detection; Saccente et al. [27] serialized the source code and selected the Long Short-Term Memory (LSTM) neural network model for vulnerability detection. Wang et al.[28] employed an improved R-GGNN (Relational Gated Graph Neural Network) model for representation learning. R-GGNN assigns different learnable weights to different types of nodes and edges, making it more effective for heterogeneous graphs. Li et al.[29] used another improved FA-GCN (Feature Attention Graph Convolutional Network) based on GCN as the main model for vulnerability feature extraction. FA-GCN can handle graphs with heterogeneous features effectively.

Xu Y et al. [30] introduces two effective mechanisms, self-ensemble and self-distillation, to enhance the fine-tuning process of BERT for NLP tasks, demonstrating improved adaptation without the need for external data or knowledge.Tanwisuth K et al. [31] presents an unsupervised fine-tuning framework for pre-trained models, enhancing their zero-shot capabilities by aligning discrete distributions from prompts and target data. Aghajanyan A et al. [32] demonstrates that a small subset of parameters can achieve nearly full performance, and that pre-training reduces intrinsic dimension, with larger models showing even lower dimensions. The findings are connected to task representations and generalization bounds, offering insights into the effectiveness of fine-tuning in the low data regime. Han W et al. [33] introduces adapter layers for fine-tuning BERT, By inserting small bottleneck layers (adapters) into each layer of the model and training these adapters with task-specific unsupervised pretraining followed by supervised training, the method achieves more consistent and robust performance across various NLP tasks. Lee J et al. [34] introduces BioBERT, a domain-specific language representation model pre-trained on biomedical corpora, which significantly outperforms BERT and previous state-of-the-art models in various biomedical text mining tasks.

Currently, some researchers are applying LLMs to vulnerability detection. Cheshkov et al. [35] evaluated the performance of ChatGPT and GPT-3 models in detecting vulnerabilities in Java code, but they found that ChatGPT and GPT-3 have limited capabilities in detecting vulnerabilities. Reference [36] pointed out that ChatGPT is not yet able to fully understand the subtle differences between vulnerabilities and non-vulnerabilities, and the information provided is not always accurate or useful. Therefore, there is a need for further research in using LLMs for vulnerability detection.

REFERENCES

[1] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: a general approach to inferring errors in systems code. SIGOPS Oper. Syst. Rev. 35, 5 (Dec. 2001), 57–72. https://doi.org/10.1145/502059.502041

[2] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: a general approach to inferring errors in systems code. SIGOPS Oper. Syst. Rev. 35, 5 (Dec. 2001), 57–72. https://doi.org/10.1145/502059.502041

[3] Kim S, Woo S, Lee H, et al. Vuddy: A scalable approach for vulnerable code clone discovery[C]//2017 IEEE symposium on security and privacy (SP). IEEE, 2017: 595-614.

[4] Cadar C, Dunbar D, Engler D R. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs[C]//OSDI. 2008, 8: 209-224.

[5] Ramos D A, Engler D. Under-Constrained symbolic execution: Correctness checking for real code[C]//24th USENIX Security Symposium (USENIX Security 15). 2015: 49-64.

[6] Avgerinos T, Cha S K, Rebert A, et al. Automatic exploit generation[J]. Communications of the ACM, 2014, 57(2): 74-84.

[7] Perl H, Dechand S, Smith M, et al. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits[C]//Proceedings of the 22nd ACM SIGSAC conference on computer and communications security. 2015: 426-437.

[8] Shin Y, Meneely A, Williams L, et al. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities[J]. IEEE transactions on software engineering, 2010, 37(6): 772-787.

[9] Dam H K, Tran T, Pham T, et al. Automatic feature learning for vulnerability prediction[J]. arXiv preprint arXiv:1708.02368, 2017.

[10] YOSINSKI J, CLUNE J, BENGIO Y, et al. How transferable are features in deep neural networks?[J]. Advances in neural information processing systems, 2014, 27.

[11] Radford A, Narasimhan K, Salimans T, et al. Improving language understanding by generative pre-training[J]. 2018.

[12] Kenton J D M W C, Toutanova L K.BERT: pre-training of deep bidirectional transformers for language understanding[C]//Proceedings of NAACL-HLT,2019: 4171-4186.

[13] GLM T, Zeng A, Xu B, et al. ChatGLM: A Family of Large Language Models from GLM-130B to GLM-4 All Tools[J]. arXiv preprint arXiv:2406.12793, 2024.

[14] Touvron H, Lavril T, Izacard G, et al. Llama: Open and efficient foundation language models[J]. arXiv preprint arXiv:2302.13971, 2023.

[15] Bai J, Bai S, Chu Y, et al. Qwen technical report[J]. arXiv preprint arXiv:2309.16609, 2023.

[16] "Qwen2 Technical Report," 2024.

[17] Houlsby N, Giurgiu A, Jastrzebski S, et al. Parameter-efficient transfer learning for NLP[C]//International conference on machine learning. PMLR, 2019: 2790-2799.

[18] Li X L, Liang P. Prefix-tuning: Optimizing continuous prompts for generation[J]. arXiv preprint arXiv:2101.00190, 2021.

[19] Zaken E B, Ravfogel S, Goldberg Y. Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked language-models[J]. arXiv preprint

arXiv:2106.10199, 2021.
[20] Guo D, Rush A M, Kim Y. Parameter-efficient transfer learning with diff pruning[J]. arXiv preprint arXiv:2012.07463, 2020.
[21] Hu E J, Shen Y, Wallis P, et al. Lora: Low-rank adaptation of large language models[J]. arXiv preprint arXiv:2106.09685, 2021.
[22] Chen Y, Qian S, Tang H, et al. Longlora: Efficient fine-tuning of long-context large language models[J]. arXiv preprint arXiv:2309.12307, 2023.
[23] Liu H, Tam D, Muqeeth M, et al. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning[J]. Advances in Neural Information Processing Systems, 2022, 35: 1950-1965.
[24] [NIST] Software Assurance Reference Dataset Project [EB/OL]. https://www.nist.gov/itl/ssd/software-quality-group/software-assurance-reference-dataset-sard-manual.
[25] Russell R, Kim L, Hamilton L, et al. Automated vulnerability detection in source code using deep representation learning[C]//2018 17th IEEE international conference on machine learning and applications (ICMLA). IEEE, 2018: 757-762.
[26] Zhou Y, Liu S, Siow J, et al. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks[J]. Advances in neural information processing systems, 2019, 32.
[27] Saccente N, Dehlinger J, Deng L, et al. Project achilles: A prototype tool for static method-level vulnerability detection of Java source code using a recurrent neural network[C]//2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW). IEEE, 2019: 114-121.
[28] Wang H, Ye G, Tang Z, et al. Combining graph-based learning with automated data collection for code vulnerability detection[J]. IEEE Transactions on Information Forensics and Security, 2020, 16: 1943-1958.
[29] Li Y, Wang S, Nguyen T N. Vulnerability detection with fine-grained interpretations[C]//Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2021: 292-303.
[30] Xu Y, Qiu X, Zhou L, et al. Improving bert fine-tuning via self-ensemble and self-distillation[J]. arXiv preprint arXiv:2002.10345, 2020.
[31] Tanwisuth K, Zhang S, Zheng H, et al. POUF: Prompt-oriented unsupervised fine-tuning for large pre-trained models[C]//International Conference on Machine Learning. PMLR, 2023: 33816-33832.
[32] Aghajanyan A, Zettlemoyer L, Gupta S. Intrinsic dimensionality explains the effectiveness of language model fine-tuning[J]. arXiv preprint arXiv:2012.13255, 2020.
[33] Han W, Pang B, Wu Y. Robust transfer learning with pretrained language models through adapters[J]. arXiv preprint arXiv:2108.02340, 2021.
[34] Lee J, Yoon W, Kim S, et al. BioBERT: a pre-trained biomedical language representation model for biomedical text mining[J]. Bioinformatics, 2020, 36(4): 1234-1240.
[35] Cheshkov A, Zadorozhny P, Levichev R. Evaluation of chatgpt model for vulnerability detection[J]. arXiv preprint arXiv:2304.07232, 2023.
[36] Aljanabi M, Ghazi M, Ali A H, et al. ChatGpt: open possibilities[J]. Iraqi journal for computer science and mathematics, 2023, 4(1): 62-64.