VP-IAFSP: Vulnerability Prediction Using Information Augmented Few-Shot Prompting with Open Source LLMs

Mithilesh Pandey[a] and Sandeep Kumar[b]
Department of Computer Science and Engineering, Indian Institute of Technology, Roorkee, India

Keywords: Software Vulnerability Prediction, Large Language Models, Prompt Enhancement, Information Augmentation, Few-Shot Prompting.

Abstract: Software vulnerabilities can cause significant damage to the organization and the user. This makes their timely and accurate detection pivotal during the software development and deployment process. Recent trends have highlighted the potential of Large Language Models for software engineering tasks and vulnerability prediction. However, their performance is often inhibited if they rely solely on plain text source code. This overlooks the critical syntactic and semantic information present in the code. To address this challenge, we introduce VP-IAFSP(**V**ulnerability **P**rediction using **I**nformation **A**ugmented **F**ew **S**hot **P**rompting). Our approach improves the LLMs' efficiency for vulnerability prediction through Prompt Enhancements by augmenting information related to the code and integrating graph structural information from the code to utilize Few-shot Prompting. To assess the proposed approach, we conduct experiments on a manually labeled real-world dataset. The results reveal that the proposed methodology achieves between 2.69% to 75.30% increase in F1-Score for function-level vulnerability prediction tasks when compared to seven state-of-the-art methods. These findings underscore the benefits of combining Information Augmentation with Few-shot Prompting while designing prompts for vulnerability prediction.

1 INTRODUCTION

Software vulnerabilities are weaknesses in software systems or implementation strategies that can be exploited by malicious actors to compromise the system's integrity, privacy, and/or availability. These vulnerabilities when unaddressed pose significant risks leading to unauthorized access, data breaches, and other security damages. Traditional methods for vulnerability detection like manual code reviews and automated static analysis tools are usually expensive, inaccurate, and limited in scope. Therefore, recent years have seen a significant rise in the development of machine learning techniques for vulnerability prediction(Nguyen et al., 2022). Deep learning models have shown promising results in recognizing patterns of vulnerabilities across large codebases(Hanif and Maffeis, 2022), (Fu and Tantithamthavorn, 2022).

Despite these advances, traditional deep-learning approaches come with limitations. They generally rely on supervised learning and require substantial labeled data to train and generalize. This imposes data collection and annotation dependencies. They can also be very time-consuming, as (Li et al., 2021) mention that it took them 9 days and 23 hours to process data and conduct training on the (Fan et al., 2020) dataset. Also, the model's adaptability to new types of vulnerabilities or code structures without retraining on new data is restricted. Few-shot Prompting offers a novel paradigm to address these issues and has been used across various domains(Zhou et al., 2022). It allows LLMs to make predictions based on provided examples within the input prompt. This reduces the need for retraining or modification of the model's parameters(Ma et al., 2023). This approach is particularly appealing for vulnerability prediction as it allows the model to adapt to new contexts and patterns using only a handful of examples. This bypasses the need for massive labeled datasets or complex training procedures.

Few-shot Prompting also holds advantages over traditional fine-tuning. Fine-tuning LLMs is computationally expensive and requires extensive resources(H. Fard, 2024). Fine-tuning can also lead to overfitting and the model may become highly spe-

[a] https://orcid.org/0000-0002-8756-0582
[b] https://orcid.org/0000-0002-3250-4866

Pandey, M. and Kumar, S.
VP-IAFSP: Vulnerability Prediction Using Information Augmented Few-Shot Prompting with Open Source LLMs.
DOI: 10.5220/0013346600003928
In *Proceedings of the 20th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2025)*, pages 592-599
ISBN: 978-989-758-742-9; ISSN: 2184-4895

cialized to the fine-tuned task at the cost of generalization. Few-shot prompting facilitates adaptability without changing the model's weights. This makes it more efficient and cost-effective. Using Few-shot prompting the model works efficiently for tasks where frequent updates or new data are expected like vulnerability prediction.

To this reason, we conduct a study on open-source LLMs using various prompts and explore how information augmentation and few-shot prompting can be used for effective vulnerability prediction. By examining different LLMs under diverse scenarios we assess their performance in identifying software vulnerabilities.

Based on our findings, we introduce VP-IAFSP for vulnerability prediction to leverage the advantages of LLMs. VP-IAFSP aims to improve the LLMs' understanding of vulnerabilities by providing clear task definitions with augmented information on vulnerabilities, and adding illustrative examples of both vulnerable and non-vulnerable functions. This approach uses the LLM's ability to process natural language instructions and apply the provided knowledge to code vulnerability prediction. Through VP-IAFSP, we provide a balance between ease of implementation and the potential for revealing vulnerabilities in source code.

Through a comprehensive evaluation of our methodology on a real-world dataset (Zhou et al., 2019), we address the following three research questions:

- **RQ1:** How effective is VP-IAFSP for vulnerability prediction?
- **RQ2:** Which open-source LLM performs best to utilize VP-IAFSP for vulnerability prediction?
- **RQ3:** How do different prompts impact the performance of various open-source LLMs in the task of Vulnerability Prediction?

2 RELATED WORKS

Traditional rule-based methods like Flawfinder(Ferschke et al., 2012), RATS, and Checkmarx rely on manually crafted patterns to identify vulnerabilities in code. Machine Learning (ML) based methods have emerged as an alternative to automate the process of vulnerability detection. These techniques can be further categorized as sequence-based and graph-based approaches.

Sequence-based methods like SySeVR(Li et al., 2021), and VulDeePecker(Li et al., 2018) treat code as a token sequence and use Natural Language Processing (NLP) techniques to extract features and classify vulnerable or non-vulnerable code. VulDeePecker uses a bidirectional Long Short-Term Memory (LSTM)(Zhou et al., 2016) network to analyze sequences of code. SySeVR extracts both semantic and syntactic code features and employs a bidirectional Gated Recurrent Unit (BGRU) network for vulnerability prediction. It also incorporates program dependence graph (PDG) analysis and program slicing in its analysis to improve its performance. (Russell and et al., 2018) use a custom C/C++ lexer to convert source code into a simplified token sequence. This captures the essential meaning of the code to reduce the vocabulary size and enable transfer learning across different datasets. Then they employ Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) to extract features from the embedded token sequences.

Graph-based methods like IVDetect(Li et al., 2022) and Devign(Zhou et al., 2019) use graph neural networks to capture the structural information present in code. Devign uses a composite graph representation that integrates Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and PDGs to learn comprehensive program semantics. It employs gated graph recurrent layers to capture both node semantics and structural features. IVDetect provides fine-grained interpretations by highlighting PDG subgraphs, statements, and dependencies relevant to detected vulnerabilities. It focuses on learning context-aware representations of vulnerable code. ReVeal(Chakra et al., 2022) utilizes Code Property Graphs (CPGs) which are a rich graph-based representation of source code that captures both semantic and syntactic relationships between code elements. It then makes use of a GGNN to analyze these graphical representations to detect software vulnerabilities.

The recent emergence of LLMs has opened new opportunities for vulnerability detection(Jiarpakdee et al., 2020). BurpGPT(Teyar, 2023) uses ChatGPT and BurpSuite together for detecting vulnerabilities in web applications. GRACE(Lu et al., 2024) is an example of an LLM-based approach that leverages graph structural information and in-context learning to improve vulnerability detection.

3 METHODOLOGY

In this section, we outline the design rationale and architecture of our proposed methodology.

3.1 Prompt Enhancements

Prompt Enhancements involve embedding information and relevant examples within a prompt to help an LLM better understand the task's context and the expected output. In a setup where the LLMs assess a code for vulnerability, the prompt P includes both contextual information C and code snippets X as shown by:

$$f_{\mathrm{P,C}}(X) = \begin{cases} 1 & \text{if } X = \text{vulnerable}, \\ 0 & \text{if } X = \text{non-vulnerable}. \end{cases}$$

A series of labeled examples (vulnerable and non-vulnerable code samples) are incorporated in the prompt to guide the LLM's understanding of what exactly constitutes a vulnerable code pattern. When the LLM receives a new code input, the examples given in the prompt help it evaluate the code with a consistent interpretation of vulnerability. This increases the reliability of the model.

Role-based prompting assigns specific roles to the language model to guide its behavior during the execution of the task. By explicitly defining roles like *"security analyst"* the model can generate more contextually relevant and precise outputs. This technique structures responses according to domain-specific needs and improves the model's behavior with the task's requirements. This is particularly effective where tasks require domain expertise and role-driven outputs. Prompt P1, given in Table 1 is an example of Role-based Prompting.

Table 1: Prompt Enhancements.

Prompt	Prompt Type	Prompt Description
P1	Task + Role Definition	You are a security professional with expertise in finding software vulnerabilities. Check if the code [X] is vulnerable or not.
P2	P1 + Domain Information	This is a C/C++ code from the Product *P*. You are a ... Check if the code [X] is vulnerable or not.
P3	P2 + Code Summary	This is a C/C++ code from the Product *P*. You are a ... Check if the code [X] is vulnerable or not.
P4	P3 + one vulnerable sample	You are a ...vulnerable code: [E]. You are a ... Check if ...
P5	P3 + two vulnerable, one non-vulnerable	You are a ...vulnerable code:[E1,E2], non-vulnerable code:[E3] ... Check if ...

Information-augmented prompting supplements the language model with additional contextual data to improve the reasoning and decision-making of the model. By embedding relevant facts and context into the prompt, the model gains access to the information it might otherwise lack. Information Augmented Prompting can be seen in Prompts P2 and P3 in Table 1. This approach is critical for applications requiring up-to-date knowledge or specific domain details. This enables the model to generate informed and accurate responses while reducing hallucination risks.

Few-shot prompting provides the language model with examples within the prompt to demonstrate the desired behavior of the model(Nashid et al., 2023). By showing input-output pairs, the model can infer patterns and generalize them to similar tasks with limited data. This method is highly effective for tasks involving classification or generating content in a specific style. Few-shot prompting uses the model's pre-trained knowledge and minimizes the need for extensive fine-tuning. Prompts P4 and P5 described in Table 1 are examples of Few-shot Prompting.

3.2 VP-IAFSP

The ensemble of role-based, information-augmented, and few-shot prompting leverages the distinct strengths of each approach and gives more insightful results. Information-augmented prompting provides necessary external knowledge and context to fill potential information gaps. Few-shot prompting then aids in guiding the model's reasoning by using input-output examples. This enables the model to generalize patterns effectively. Each prompting technique has its weaknesses but the ensemble mitigates these limitations. Information-augmented prompting relies heavily on the quality of supplementary data while Few-shot prompting risks overfitting if the examples are too narrow(Ye and Durrett, 2022). By combining these approaches the ensemble creates prompts that are contextually informed and well-structured. This enables the model to handle complex tasks like software vulnerability prediction with improved accuracy.

Figure 1 illustrates the proposed methodology. It comprises basically two modules- Information Augmentation, and Few-Shot Selection.

3.2.1 Information Augmentation

Since we are asking the model to check if a code is vulnerable or not, it is important to provide the model with a concise and standard definition of a vulnerability. For this purpose, we use the definition of vulnerability provided by NVD: *"A vulnerability is a weakness in the computational logic ... when exploited, results in a negative impact to confidentiality, ..."* (Johnson et al., 2011). This increases not only the model's vocabulary but also its comprehension of vulnerabilities.

For the LLMs to perform better on a given task we need to provide it with information related to that particular task. The incorporation of file names in the

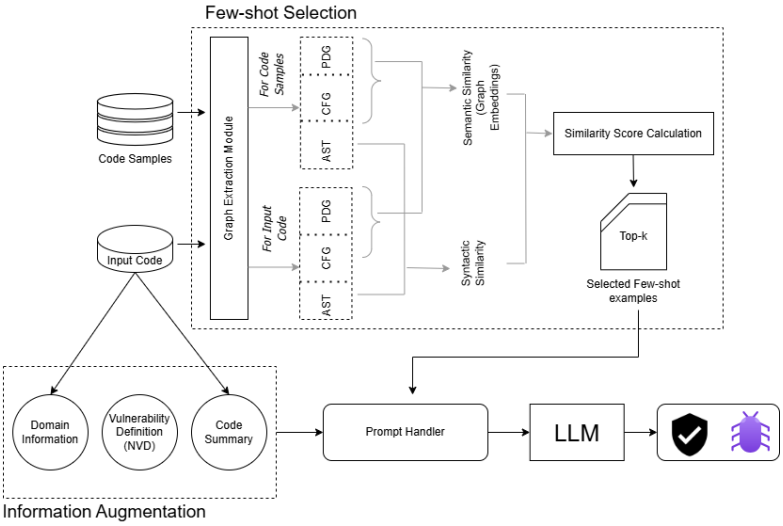

Figure 1: VP-IAFSP architecture and its sub-modules: Information Augmentation and Few-Shot Selection.

prompt can significantly improve the effectiveness of LLMs(Li et al., 2023). We include the project information and the language in which the code is written in the prompt for every input code to help the model understand the code better.

Finally, we provide the final Prompt with a summary of the code generated by an LLM itself. Various approaches have been proposed for code summarization through LLMs (Yun et al., 2024) (Kumar and Chimalakonda, 2024). For the purpose of this research, we use the same LLM for creating a summary of the code and supplying it to another prompt for the final prediction task.

3.2.2 Few Shot Selection

GRACE(Lu et al., 2024) uses few-shot prompting but does not employ CFGs and PDGs while finding the most similar code to include in the prompts. This reduces the quality of the examples that are used in few-shot prompting since these graph structures signify the code's model execution flows, and control and data-flow information respectively(Wu et al., 2021). We use various graph-based representations including ASTs, PDGs, and CFGs to identify similar functions from a set of code samples. By extracting these representations for both the input function and the candidate functions we compute similarity scores based on structural and semantic characteristics. This allows for a more detailed comparison and enables the identification of functions that share similar behavior even if their syntax differs.

After extracting the graph representations of the input code we generate embeddings using Graph Convolutional Networks(GCNs). These embeddings serve as compact representations of the graph structures and help capture essential features to facilitate similarity comparison. Similar graph representations are extracted and embedded for each function in the code samples. The forward pass through a two-layer GCN is defined as follows:

1. First Convolution Layer:

$$\mathbf{H}^{(1)} = \mathrm{ReLU}(\mathbf{A}\mathbf{X}\mathbf{W}_1) \tag{1}$$

Where: - $\mathbf{A}$ is the adjacency matrix, $\mathbf{X}$ is the feature matrix of nodes, $\mathbf{W}_1$ is the weight matrix for the first convolution layer.

2. Second Convolution Layer:

$$\mathbf{H}^{(2)} = \mathbf{A}\mathbf{H}^{(1)}\mathbf{W}_2 \tag{2}$$

Where: - $\mathbf{W}_2$ is the weight matrix for the second convolution layer.

The final graph embedding is:

$$\mathbf{h}_G = \mathrm{mean}(\mathbf{H}^{(2)}) \tag{3}$$

For ASTs comparison to calculate Syntactic Similarity, we use a metric called Jaccard Similarity. The Jaccard similarity coefficient is used to compare the similarity between two graphs based on their nodes, edges, and attributes. The formula for the Jaccard similarity is given by:

$$J(\mathcal{G}_1, \mathcal{G}_2) = w_1 \cdot J_{\text{nodes}} + w_2 \cdot J_{\text{edges}} + w_3 \cdot J_{\text{node attributes}} + w_4 \cdot J_{\text{edge attributes}} \tag{4}$$

Where:

$$J_{\text{nodes}} = \frac{|N_1 \cap N_2|}{|N_1 \cup N_2|} \tag{5}$$

$$J_{\text{edges}} = \frac{|E_1 \cap E_2|}{|E_1 \cup E_2|} \tag{6}$$

$$J_{\text{node attributes}} = \frac{1}{|N_1 \cap N_2|} \sum_{n \in N_1 \cap N_2} \mathbf{1}_{\text{attributes match}} \quad (7)$$

$$J_{\text{edge attributes}} = \frac{1}{|E_1 \cap E_2|} \sum_{e \in E_1 \cap E_2} \mathbf{1}_{\text{attributes match}} \quad (8)$$

We use PDGs and CFGs for the calculation of Semantic Similarity. The Euclidean distance between two embeddings $\mathbf{e}_1$ and $\mathbf{e}_2$ is calculated as:

$$d(\mathbf{e}_1, \mathbf{e}_2) = \sqrt{\sum_{i=1}^{n} (e_{1,i} - e_{2,i})^2} \quad (9)$$

Where $\mathbf{e}_1 = (e_{1,1}, e_{1,2}, \ldots, e_{1,n})$ and $\mathbf{e}_2 = (e_{2,1}, e_{2,2}, \ldots, e_{2,n})$ are the two embedding vectors.

To compute similarity, the distance is transformed using an exponential decay function:

$$\text{Sim}(\mathbf{e}_1, \mathbf{e}_2) = \exp(-d(\mathbf{e}_1, \mathbf{e}_2)) \quad (10)$$

Once the Syntactic and Semantic Similarities are calculated, the final Similarity Score is calculated as :

$$\begin{aligned} \text{Similarity Score} = w \cdot \text{Syntactic Similarity} \\ + (1 - w) \cdot \text{Semantic Similarity} \end{aligned} \quad (11)$$

After the Similarity Score between the Input Code and the Code Samples is calculated, the code samples with the highest Similarity Scores are added to the Prompt.

4 RESULTS AND COMPARATIVE ANALYSIS

4.1 System Settings

We employ 13th Gen Intel(R) Core(TM) i9-13900 with 62Gi RAM and 16 GB Quadro RTX 5000 GPU running on Ubuntu 24.04 LTS operating system for experimentation.

We use Joern for parsing the code into its graph representations and Python's Pytorch library for creating graph embeddings. To access the LLMs we use the Ollama API and the Python's langchain_ollama library. Another Python library used is scikit_learn for evaluation.

4.2 Dataset

(Zhou et al., 2019) is a manually labeled dataset compiled from two open-source C projects: FFmpeg and Qemu. It is a balanced dataset that consists of 10,067 vulnerable functions, alongside 12,294 non-vulnerable functions.

4.3 Evaluation

The performance of the proposed approach is evaluated using Accuracy, Precision, Recall, and F1-Score. Accuracy provides an overall measure of correctness, while Precision evaluates the model's ability to avoid false positives. Recall assesses the detection of true positives. F1-Score balances Precision and Recall to offer a comprehensive performance metric.

4.4 Experimentation

As described in Figure 1, we start by augmenting information about the software vulnerabilities to the prompt. First, we tell the prompt the domain information about the code- what language it is written in and what Project is the code from. Next, we incorporate the vulnerability definition into the prompt to give the prompt a clear definition. To sum up the Information Augmentation part of the approach we provide the Prompt with an LLM-generated summary of the code to add additional context. Detailed methodology for augmenting contextual information is given in Section 3.2.1.

To select the examples to include in the prompt, we make use of the Few-Shot Selection module shown in Figure 1. The exact methodology used is given in Section 3.2.2. By calculating the similarities between the input code and the code samples we find the most similar codes to input to the prompt. Next, we experiment with five state of the art Open Source LLM models- Llama3.2:2b, Gemma2:27b, Llama3.1:70b, CodeLlama:7b, and Codestral:22b. We make use of Ollama API to access these models and input the curated prompts.

4.5 Results

In this subsection, we present the results of our methodology with respect to the proposed research questions.

4.5.1 RQ1: How Effective Is VP-IAFSP for Vulnerability Prediction?

As shown in Table 2, the proposed methodology outperforms all the compared methods in Recall and F1-Score. VP-IAFSP achieves an increment between 2.69% and 75.30% in the F1-score as compared to state-of-the-art approaches. With respect to accuracy, VP-IAFSP outperforms two state-of-the-art sequence-based learning approaches (Li et al., 2018) and (Li et al., 2021). The low value of accuracy is due to the LLM's limited knowledge about software

vulnerabilities and code in general. VP-IAFSP tries to overcome this limitation by providing more context to the models by augmenting information related to the code and few-shot scenarios to increase the model's understanding of the task.

Table 2: Overall performance of VP-IAFSP using different open source LLMs compared to state-of-the-art vulnerability prediction models.

Model	Accuracy	Precision	Recall	F1-Score
(Li et al., 2018)	49.91	46.05	32.55	38.14
(Russell and et al., 2018)	57.60	54.76	40.72	46.71
(Li et al., 2021)	47.85	46.06	58.81	51.66
(Zhou et al., 2019)	56.89	52.50	64.67	57.95
(Chakra et al., 2022)	**61.07**	**55.50**	70.70	62.19
(Li et al., 2022)	57.26	52.37	57.55	54.84
(Lu et al., 2024)	59.78	53.94	82.13	65.11
VP-IAFSP (Llama-3.2)	51.22	50.63	**98.41**	**66.86**
VP-IAFSP (Llama-3.1)	44.41	41.23	51.57	45.82
VP-IAFSP (Gemma 2)	56.42	52.24	55.89	53.46
VP-IAFSP (CodeLlama)	49.91	46.05	77.55	57.79
VP-IAFSP (Codestral)	56.21	55.32	61.29	58.15

4.5.2 RQ2: Which Open-Source LLM Performs Best to Utilize VP-IAFSP for Vulnerability Prediction?

The performance of various LLMs on different prompts is described in Table 3. The conducted experiments show that LLama3.2:2b outperforms other LLMs for Recall and F1-Score. Gemma2:27b shows the greatest accuracy(56.42%) among the LLMs for vulnerability prediction. Codestral:22b(56.21%) performs almost similarly to Gemma2 on accuracy and Codestral has the highest value for Precision among the studied LLMs. Codestral gives a Precision of 55.32%. It is worth mentioning that though Llama3.2's high Recall value is advantageous in recognizing nearly all potential vulnerabilities, there is still a challenge in mitigating the number of false positives. This is a characteristic challenge of using LLMs in vulnerability prediction as previously observed by (Çetin et al., 2024) for GPT-3.5. Though this trade-off may be acceptable in security contexts where addressing potential vulnerabilities is crucial, balancing these metrics can help ensure a robust security management system.

4.5.3 RQ3: How Do Different Prompts Impact the Performance of Various Open-Source LLMs in the Task of Vulnerability Prediction?

Experiments performed on various prompts show that by augmenting information about the code and vulnerability definition, the performance of the LLMs increases, as shown in Table 3. The careful selection of examples to append to the prompt increases the overall efficiency of the LLMs for vulnerability prediction tasks. We also find that the efficiency of the models increases as we append more examples in prompt P5 than P4 and provide the model with concrete examples of what vulnerable and non-vulnerable functions look like.

To conduct investigations beyond Vulnerability Prediction, we test VP-IAFSP's capability for Vulnerability type classification. For this purpose we extract a dataset from (Fan et al., 2020) that spans multiple vulnerability classes between 2002 and 2019. The resulting dataset is a compilation of functions corresponding to 91 different vulnerability types. We compare our method with two state-of-the-art methods for Vulnerability Type Classification- (Zhou et al., 2019) and (Chakra et al., 2022) and the results are shown in Table 4. The results are obtained by calculating Accuracy and weighted F1-score for each model. To calculate the Weighted F1-score we calculate the F1-score for every class and the weights are calculated through the relative frequency of every class present in the dataset.

4.6 Discussion

The improvement of VP-IAFSP over the existing works can be attributed to the additional information and the examples appended to the Prompt. By adding the information about the code to be assessed, we provide the model with the information needed to make it perform better. While telling the model to assess if a code is vulnerable or not, it is imperative to suggest it as to what a vulnerability really is. Similarly, providing the model with the domain information increases the performance of the model. This can be seen in Table 3, where the performance of Prompts with augmented information is better than the Prompts that are only provided the Task Description.

We utilize Few-shot Prompting to help the model make better decisions by supplying it with examples of vulnerable and non-vulnerable code snippets. By carefully selecting the most similar examples to append to the Prompt the model can efficiently distinguish the vulnerable code from the non-vulnerable.

Table 3: Detailed Performance Analysis for Selected Models.

Model	Pro.	Acc.	Prec.	Rec.	F1-Score
Llama3.2	P1	41.77	41.31	96.33	57.82
	P2	43.26	42.73	93.72	58.70
	P3	47.85	46.71	91.47	61.84
	P4	47.34	47.82	95.42	63.71
	P5	51.22	50.63	**98.41**	**66.86**
Gemma2	P1	48.73	46.50	51.53	48.89
	P2	50.96	47.55	52.05	49.70
	P3	51.18	47.54	52.14	49.73
	P4	52.97	48.81	54.56	51.53
	P5	**56.42**	52.24	55.89	53.99
Code-Llama	P1	41.71	42.68	68.71	52.65
	P2	42.48	42.57	69.55	52.81
	P3	45.77	46.36	71.26	56.17
	P4	47.69	47.28	74.31	57.79
	P5	49.91	46.05	77.55	57.79
Codestral	P1	51.49	48.24	57.41	52.43
	P2	51.98	50.24	58.21	53.93
	P3	54.27	51.74	57.96	54.67
	P4	54.77	52.19	59.64	55.67
	P5	56.21	**55.32**	61.29	58.15
Llama3.1	P1	41.36	39.71	46.25	42.73
	P2	42.08	40.21	44.83	42.34
	P3	43.88	42.63	47.24	44.82
	P4	44.23	42.87	48.79	45.64
	P5	44.41	41.23	51.57	45.82

Table 4: VP-IAFSP for Vulnerability Classification.

Model	Accuracy	Weighted F1
(Zhou et al., 2019)	19.69	46.71
(Chakra et al., 2022)	28.36	49.22
VP-IAFSP	31.27	50.38

Table 2 highlights the effectiveness of our method when compared to other state-of-the-art approaches for vulnerability prediction.

We used various open-source LLMs in our study and found that they can be efficiently used for Vulnerability Prediction without any need for fine-tuning. It is important to mention that though LLMs beat the state-of-the-art methods on various metrics using the proposed methodology, their limited knowledge of code still poses as a hurdle for their extensive use for vulnerability prediction tasks, and for software engineering tasks in general.

5 CONCLUSIONS

In this paper, we have proposed a method for vulnerability prediction using Information Augmentation and Few Shot Prompting. The proposed method utilizes CFGs, PDGs, and ASTs to find similar code samples to input to the prompt along with the Input Code that needs to be verified. This enhances the LLM's capability to adapt to a particular task.

We consider various Open Source LLMs and utilize them for the vulnerability prediction task. The experimental results show that the proposed methodology is effective in distinguishing vulnerable and non-vulnerable codes without any need for data preparation or supervised training. This work can be extended by constructing even more appropriate prompts for vulnerability prediction or using techniques like attention steering(Zhang et al., 2024).

REFERENCES

Chakra, S., Krishna, R., Ding, Y., and Ray, B. (2022). Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 48(9):3280–3296.

Fan, J., Li, Y., Wang, S., and Nguyen, T. N. (2020). A c/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 508–512.

Ferschke, O., Gurevych, I., and Rittberger, M. (2012). Flawfinder: A modular system for predicting quality flaws in wikipedia. In *CLEF (Online Working Notes/Labs/Workshop)*, pages 1–10.

Fu, M. and Tantithamthavorn, C. (2022). Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 608–620. ACM.

H. Fard, F. (2024). Technical briefing on parameter efficient fine-tuning of (large) language models for code-intelligence. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ICSE-Companion '24, page 434–435, New York, NY, USA. Association for Computing Machinery.

Hanif, H. and Maffeis, S. (2022). Vulberta: Simplified source code pre-training for vulnerability detection. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE.

Jiarpakdee, J., Tantithamthavorn, C. K., Dam, H. K., and Grundy, J. (2020). An empirical study of model-agnostic techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 48(1):166–185.

Johnson, A., Johnson, A., Dempsey, K., Ross, R., Gupta, S., and Bailey, D. (2011). Guide for security-focused configuration management of information systems. Technical report, US Department of Commerce, National Institute of Standards and Technology.

Kumar, J. and Chimalakonda, S. (2024). Code summarization without direct access to code: Towards exploring federated llms for software engineering. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 100–109.

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., and et al. (2023). Starcoder: May the source be with you! *arXiv preprint arXiv:2305.06161*.

Li, Y., Wang, S., and Nguyen, T. N. (2022). 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*, pages 292–303. ACM.

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

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

Lu, G., Ju, X., Chen, X., Pei, W., and Cai, Z. (2024). Grace: Empowering llm-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software*, 212.

Ma, H., Zhang, C., Bian, Y., Liu, L., Zhang, Z., Zhao, P., Zhang, S., Fu, H., Hu, Q., and Wu, B. (2023). Fairness-guided few-shot prompting for large language models. In *Advances in Neural Information Processing Systems*, volume 36, pages 43136–43155.

Nashid, N., Sintaha, M., and Mesbah, A. (2023). Retrieval-based prompt selection for code-related few-shot learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2450–2462. IEEE.

Nguyen, V.-A., Nguyen, D. Q., Nguyen, V., Le, T., Tran, Q. H., and Phung, D. (2022). Regvd: Revisiting graph neural networks for vulnerability detection. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 178–182. ACM.

Russell, R. and et al. (2018). Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 757–762, Orlando, FL, USA.

Teyar, A. (2023). Burpgpt: Chatgpt powered automated vulnerability detection tool. https://burpgpt.app/#faq.

Wu, Y., Lu, J., Zhang, Y., and Jin, S. (2021). Vulnerability detection in c/c++ source code with graph representation learning. In *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 1519–1524. IEEE.

Ye, X. and Durrett, G. (2022). The unreliability of explanations in few-shot prompting for textual reasoning. *Advances in Neural Information Processing Systems*, 35:30378–30392.

Yun, S., Lin, S., Gu, X., and Shen, B. (2024). Project-specific code summarization with in-context learning. *Journal of Systems and Software*, 216.

Zhang, Q., Singh, C., Liu, L., Liu, X., Yu, B., Gao, J., and Zhao, T. (2024). Tell your model where to attend: Post-hoc attention steering for llms. *arXiv preprint arXiv:2311.02262*.

Zhou, K., Yang, J., Loy, C. C., and Liu, Z. (2022). Conditional prompt learning for vision-language models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16816–16825.

Zhou, P., Shi, W., Tian, J., Qi, Z., Li, B., Hao, H., and Xu, B. (2016). Attention based bidirectional long short-term memory networks for relation classification. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 207–212.

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

Çetin, O., Ekmekcioglu, E., Arief, B., and Hernandez-Castro, J. (2024). An empirical evaluation of large language models in static code analysis for php vulnerability detection. *Journal of Universal Computer Science*, 30(9):1163–1183.