Software Vulnerability Detection Using LLM: Does Additional Information Help?

Samiha Shimmi[1], Yash Saini[1], Mark Schaefer[1], Hamed Okhravi[2], Mona Rahimi[1]
[1]Department of Computer Science, Northern Illinois University, IL, USA
Email: {sshimmi, z1949906, z1927111, rahimi}@niu.edu
[2]MIT Lincoln Laboratory, MA, USA, Email: hamed.okhravi@ll.mit.edu

***Abstract*—Unlike conventional machine learning (ML) or deep learning (DL) methods, Large Language Models (LLM) possess the ability to tackle complex tasks through intricate chains of reasoning, a facet often overlooked in existing work on vulnerability detection. Nevertheless, these models have demonstrated variable performance when presented with different prompts (inputs), motivating a surge of research into prompt engineering – the process of optimizing prompts to enhance their performance. This paper studies different prompt settings (zero-shot and few-shot) when using LLMs for software vulnerability detection. Our exploration involves harnessing the power of both natural language (NL) unimodal and NL-PL (programming language) bimodal models within the prompt engineering process. Experimental results indicate that LLM, when provided only with source code or zero-shot prompts, tends to classify most code snippets as vulnerable, resulting in unacceptably high recall. These findings suggest that, despite their advanced capabilities, LLMs may not inherently possess the knowledge for vulnerability detection tasks. However, few-shot learning benefits from additional domain-specific knowledge, offering a promising direction for future research in optimizing LLMs for vulnerability detection.**

***Index Terms*—Software vulnerability detection, software vulnerability detection using LLM**

1. Introduction

Software vulnerabilities are steadily growing year over year, and despite decades of effort in improving sanitization capabilities, there is still a need for more robust vulnerability detection frameworks [1], [2]. Large Language Models (LLMs) have become popular recently and utilized in multiple security domains including intrusion detection [3], anomaly detection [4], and program repair [5], among others. LLMs are also used in software vulnerability detection [6], [7], [8], [9], although more research is needed to thoroughly understand their capabilities. Some of the existing efforts have shown that LLMs were not performing well compared to State-of-The-Art (SOTA) techniques [10], [8]. Others have used various prompts to analyze their results among different prompt settings [11], [12]. However, the nascence of research in this area means that additional work is needed to fully understand the capability of LLMs for software vulnerability detection. For example, existing work focuses on a limited number of samples [8], [7], [11], and larger-scale studies are necessary.

Prompt engineering is a specific aspect of AI engineering, which involves the design and optimization of instructions given to an LLM to achieve desired outputs or behaviors [13]. Prompt Engineering is commonly applied for the purpose of improving downstream natural language processing (NLP) tasks and LLM applications [14]. Prompt engineering encompasses various techniques, such as experimenting with different phrasing, specifying desired answer formats, providing context, or adding instructions to guide the model's behavior. This iterative process empowers developers to fine-tune the model's behavior and enhance its performance without relying solely on labeled data.

Zero-shot and few-shot are terms used in the context of ML and NLP models, especially for LLMs [15]. They refer to different ways of utilizing the model's capabilities to perform tasks for which it was not explicitly trained [16]. Zero-shot learning refers to the ability of a model to perform a task without any specific training data or examples for that task. Instead, it relies on general knowledge learned during its training on a diverse range of tasks. The model understands the task's requirements from the provided input and attempts to generate a relevant response. Few-shot learning is a step beyond zero-shot, referring to the model's ability to perform a task with only a few examples or demonstrations, rather than having a complete training dataset [17]. The model can generalize from these limited examples to grasp the essence of the task and produce reasonable results [18].

In this paper, our objective is to shed light on whether LLMs inherently possess the knowledge required for identifying vulnerabilities in source code effectively. We undertake a systematic exploration of this question, initially

. DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.
This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering.

evaluating LLMs in their raw form without any supplementary knowledge. Later, we delve into the process of extracting domain knowledge (auxiliary information to be provided to the LLM) when it is not readily available. Our experimentation involves utilizing both unimodal and bimodal deep learning models to extract domain information for LLMs, aiming to determine if any of these approaches are more effective than the basic prompt. We conducted a large-scale analysis with 17,761 samples from the BigVul [19] dataset and our experiments reveal several limitations and difficulties (Sec.5) while using LLMs in real-world practical scenarios.

Furthermore, we introduce multiple strategies for optimizing the utilization of LLMs in classifying vulnerable methods in code. Our efforts focus on enhancing both zero-shot and few-shot learnings, aiming to elevate the overall performance of automatic vulnerability detection.

The primary goal of this paper is to serve as a resource that inspires and aids future researchers and engineers to appropriately exploit LLMs for the classification of vulnerabilities. Our experimental result on a large-scale dataset shows that applying the above-mentioned strategies improves accuracy over the basic prompt. All artifacts of this study are available[1].

2. Related Work

This section provides a summary of the related efforts that use LLM for software vulnerability detection. There are several studies that show well-structured prompts lead to improved LLM performance in various downstream tasks [20], [21], [22]. Vulnerability detection using prompt engineering is still a nascent research area.

Chen et al. [23] investigated the effectiveness of vulnerability detection of smart contracts using ChatGPT. To find the optimized prompt, they used ChatGPT itself as a prompt optimizer and later used those prompts for vulnerability detection. Cheshkov et al. [10] worked on a performance comparison of two widely used LLMs ChatGPT and GPT-3. They used several prompts and tested their impact, finding that the results were consistent among different prompts.

Lu et al. [9] introduced GRACE, where they provided LLM with several auxiliary information, including a Code Property Graph (CPG) along with several similar code snippets. In their analysis, providing additional information was helpful and yielded better results. In another work, Zhou et al. [8] provided auxiliary information—role description, Top CWE information, randomly sampled similar neighbors, and similar neighbors based on CodeBERT neighbors—expecting a better result.

Hu et al. [24] proposed a new prompting paradigm "open-ended prompting" in addition to existing binary class and multi-class prompt. The idea of their approach is to detect any potential vulnerability and describe them in natural language without any predefined vulnerability information.

1. https://github.com/research7485/vulnerability_detection

Two recent efforts have delved into the application of LLMs to vulnerability [7], [11]. The former primarily concentrates on evaluating the performance of multiple LLMs for this task, without making alterations to the fundamental prompts used [7], while the latter focuses on the addition of API calls to ChatGPT4 to improve its performance.

In contrast, we aim to design prompts to be automatically augmented with the most useful additional information about the most relevant CWEs to a given function according to the function's context. A recent work, Vul-RAG [25], uses a retrieval-augmented generation (RAG) framework to facilitate vulnerability detection by retrieving external knowledge from a pre-built knowledge base. While Vul-RAG relies on external knowledge retrieval, our approach focuses on optimizing prompts with domain-specific information directly within the LLM context, particularly improving few-shot learning performance without the need for external retrieval systems.

3. Basic Prompt

For our experiment, we used the closed-source LLM Gemini-pro[2]. We used the API it provides to collect the information. We used Gemini-pro because its API supports querying on large-scale data and it has been used recently in the literature for other security-related tasks [26], [27], [28]. Another advantage of Gemini-pro API is that it accepts around 32k tokens which helps us provide additional information to the LLM while prompting it[3].

For our preliminary experiment, we used the **Big-Vul** [19] dataset. Big-Vul is a C/C++ code vulnerability dataset curated from 348 open-source projects on GitHub. The vulnerable code is identified by the public CVE database and CVE-related source code repositories. The dataset incorporates descriptive information about the vulnerabilities from the CVE database, such as CVE IDs, related CWE, CVE severity scores, and CVE summaries. In total, Big-Vul contains 11,823 vulnerable methods and 253,096 safe methods. We took 8,921 vulnerable methods that have corresponding CVE and CWE information and to create a balanced dataset, we took 8,740 non-vulnerable methods.

In this preliminary analysis, our objective was to assess whether LLMs inherently possess the ability to detect vulnerability in source code. In our experiment, we queried a language model about vulnerabilities from the set of methods in the BigVul dataset. The precise prompt that we used is *"Is the following function vulnerable to any Common Weakness Enumeration (CWEs)?" + [function]*? We then calculated the number of actual vulnerable methods correctly identified as such (true positives – TP), those incorrectly classified as vulnerable (false positives – FP), those incorrectly classified as safe (false negatives – FN), and those correctly classified as safe (true negatives – TN).

2. https://gemini.google.com/app
3. https://ai.google.dev/gemini-api/docs/tokens

TABLE 1: Big-Vul: Results from the Basic Prompt.

Input	TP	TN	FP	FN	Acc.	Pre.	Rec.	F1	F2
Method	8,500	176	8,729	210	49.25%	49.33 %	97.59%	65.54%	81.62%

The results of this experiment are detailed in Table 1. The overall accuracy of the model is 49.25%, which is comparable to a dummy classifier. Additionally, the basic prompt led the LLM to classify most instances as vulnerable, resulting in an unacceptably high recall rate of 97.59%. This result might be an indication that LLMs do not inherently possess the ability to detect vulnerability in source code without auxiliary information.

Another observation is that LLM did not provide a direct ”yes” or ”no” answer. This led us to formulate a method to correctly identify FP, FN, TP, and TN. We manually analyzed the answers provided by the LLM, and, to detect if the label is vulnerable, we only considered the records that contain the following words *”yes”, ”is vulnerable”, ”does not validate”*, etc. To detect if the label is safe, we used the following keywords: *”is not vulnerable”, ”appears to be non-vulnerable”*, etc. The complete list of the keywords is provided in Appendix A.

It is worth mentioning that despite formulating these conditions, we encountered a small number of records that could not be categorized based on our keywords. As a result, the summation of TP, TN, FP, and FN is not equal across all experiments. Additionally, in all the experiments, while asking for responses through the API, we encountered errors for a small number of records. These records are not consistent, causing the count to vary among different experiments.

4. Proposed Prompt Designs

As we observed in the previous section, providing only the basic prompt did not show a good performance accuracy-wise, rather recognizing most of the records as vulnerable regardless of their actual vulnerability status. In this section, we provided domain-specific information to analyze the output provided by the model. Additionally, we try both zero-shot and few-shot scenarios hoping for a better result in few-shot learning.

For this purpose, we sought methods to acquire domain knowledge with better accuracy. This section delves into our two distinct designs, denoted as D_1 and D_2 with the primary objective of automatically extracting pertinent domain knowledge.

In D_1, we harnessed a text-code bimodal neural network (CodeBERT) to enrich prompts with pertinent code examples, thereby furnishing additional information to the language model. In our study, we employed CodeBERT [29] for bimodal-based similarity comparison, leveraging its transformer-based architecture to effectively capture relationships between natural language and programming language elements. Due to its bimodal nature, CodeBERT can generate embeddings from both source code and natural language. Since the original CodeBERT model was not trained in the C language, we fine-tuned it on the Devign dataset [30] for C. In D_2, we acquired domain knowledge through an unimodal neural network (SBERT), leveraging semantic analysis to identify the most relevant vulnerabilities. For unimodal semantic comparison, we utilized SBERT [31], a transformer-based model well-suited for semantic analysis tasks and used in the literature [32], [2]. The choice of these models was motivated by the performance of transformer architectures in understanding and processing complex relationships within code snippets and natural language expressions. Our objective centered on evaluating and comparing the efficacy of these two prompt designs in identifying code vulnerabilities at the granularity of methods.

As illustrated in Figures 1 and 2, both designs incorporate experimentation with both zero-shot and few-shot learning. In zero-shot experiments, we simply tried to identify the most similar CWEs and provide the description of those CWEs in natural language to the LLM. While in few-shot learning after gaining this information we perform an additional approach and find multiple method examples that map to the found CWE along with an equal number of non-vulnerable examples (so that the model is not biased towards vulnerable examples) and pass this information to the LLM. The underlying assumption here is that the model will utilize the provided examples to better recognize an existing vulnerability within a given method. To detect similar examples, we used the widely used cosine similarity metric that is used in the literature to compare similarity between embeddings [33], [34], [35].

During this step, records without a matching CWE in the function description data file were discarded (around 20% depending on different settings). At that time, the ‘Vulnerability Assessment’ step described below was not planned. The main goal was to compare how closely the CWEs returned by SBERT and CodeBERT matched the actual CWEs. To avoid skewing the results, those records were excluded. Even though the ‘Vulnerability Assessment’ step was later added, these records were not reinstated. We accept that as a limitation of our approach.

4.1. D_1 - Augmenting with Code Examples

While “bimodal” denotes a model that integrates information from two different modalities, such as natural language and programming language, “unimodal” refers to a model that operates within a single mode, specifically natural language in this context [36].

In this particular design, we leveraged an NL-PL deep neural model to measure the similarity between the CWE descriptions in NL and the methods in PL as a measure of similarity. Unlike an unimodal model, the bimodal model is mutually trained on both, programming and natural languages, overcoming potential limitations associated with using a single modality. As such, the model is adept at comparing artifacts from both modalities, enabling it to learn implicit relationships between NL and PL components.

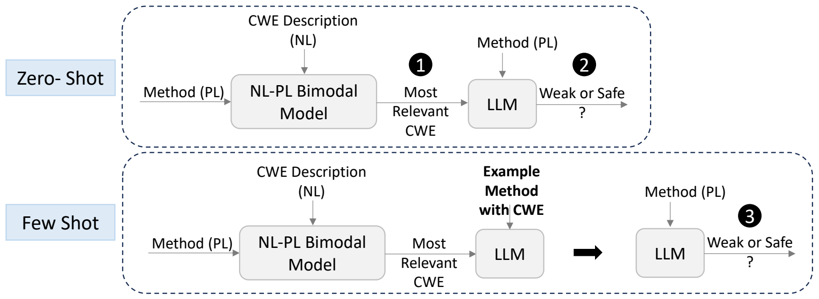

Figure 1: An overview of the design using a bimodal neural model in zero- and few-shot learning.

The assumption here is that the model's versatility in processing both natural language and code within a shared space allows for a direct comparison between the method's source code and the English descriptions of CWE entities. As such, bimodality offers a more direct and efficient means of comparing code and English descriptions to assess security vulnerabilities in the method. This modification is expected to enhance the precision of identifying relevant CWEs for a given method. This design consists of two steps:

- **Step 1 - Bimodal Information Space Construction:** In the initial step, we created a bimodal space that incorporates both NL and PL information. Each sample is evaluated and placed in this space based on both NL and PL attributes. This process yields the most relevant CWEs (based on cosine similarity) for a given function, taking into account semantic aspects as well as semantic and syntactic information from provided code examples. The bimodal model allows for a comprehensive comparison of data across different modalities. Integrating semantic and syntactic properties are proposed in the literature to enhance the ability to detect vulnerability [2], [37].
- **Step 2 - Vulnerability Assessment:** The method, together with the identified CWEs, is presented to the LLM again for detecting vulnerability. This time the LLM assesses whether the method's code or implementation is susceptible to the most probable vulnerabilities associated with its function.

In this design, within our zero-shot experiments, the focus is on identifying the most similar CWEs and conveying the description of those CWEs in natural language to the LLM. However, in the few-shot learning approach, after acquiring this information, we adopted an additional strategy. We sought out multiple method examples that correspond to the identified CWE along with an equal number of non-vulnerable methods and presented this enriched information to the LLM. This approach aims to provide the model with a more comprehensive understanding of the identified CWE and its manifestations in various code instances, potentially enhancing the model's ability to discern and classify vulnerabilities in a broader context. The reason we incorporate the non-vulnerable methods is to reduce any bias caused by only providing vulnerable information to the LLMs.

4.2. D_2 - Integrating Semantic Knowledge

In this approach, as illustrated in Figure 2, we devised an automated pipeline that initially harnesses an LLM to extract a description of a method's functionality and purpose in natural language. Later, we utilize a pre-trained neural network to semantically identify vulnerabilities that share similar terminologies with the given method's objectives in natural language. The resulting set of CWEs, along with the method itself, are then fed into an LLM to ascertain the method's vulnerability to the most probable ones associated with its function. This design, therefore, contains the three primary steps as follows:

- **Step 1 - Automated Objective Identification:** The method body is passed to the LLM, leveraging the model to extract a description of the functionality. This description provides the model with contextual knowledge describing what the method is supposed to do.
- **Step 2 - Semantic Vulnerability Matching:** Once the method's description is extracted, a pre-trained neural network, SBERT is used to semantically identify vulnerabilities that share similar terminology with the method based on the cosine similarity metric. This involves analyzing the language used to describe the method and finding vulnerabilities with terminology or context that is related to the method's functionality.
- **Step 3 - Vulnerability Assessment:** The method along with the list of identified CWEs are then passed to the LLM again. The LLM is used here for the second time to assess whether or not the method is vulnerable with respect to the most likely vulnerabilities associated with that function. In other words, it evaluates whether the method's code or implementation is susceptible to the vulnerabilities identified in Step 3. Similar to the bimodal approach, we are passing an equal number of non-vulnerable methods to the LLM to minimize the bias caused by only providing vulnerable samples.

We then, first provide the source code along with CWE descriptions for zero-shot implementation. In the few-shot learning variant of this design, after obtaining the initial information, we augment it with multiple method-level code examples, containing the same CWE associated with them along with a similar number of non-vulnerable methods, and pass this enriched information to another instance of

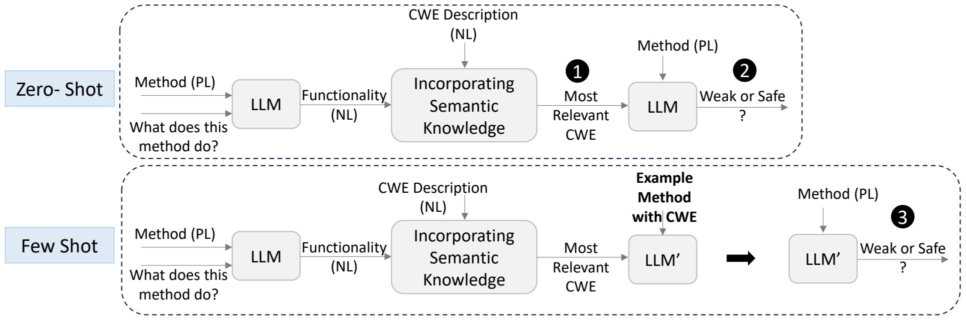

Figure 2: An overview of the design using an unimodal neural model in zero- and few-shot learning.

the same LLM, albeit through a new API and conversation thread. The adoption of an independent model here is to mitigate potential biases in the initial LLM, ensuring a more unbiased and comprehensive evaluation of the model's accuracy in identifying the method's vulnerability.

5. Evaluation

For evaluation purposes, we used the same BigVul dataset that we used in Section 3. We also chose the Gemini-pro API that we used in our preliminary study.

5.1. Interplay of Precision and Recall

We employed two distinct strategies to investigate the trade-offs between precision and recall, which are labeled as **Precision Focus** and **Precision Focus** approaches in Tables 2.

In the former strategy, we applied a stringent threshold for similarity measures, excluding all samples showing any degree of semantic similarity (above zero) to the method under examination, with the exception of the top three most similar samples. In this scenario, regardless of the magnitude of the similarity score, ranging from zero to one, we only retained those with the highest similarity measures in the method. Consequently, this experiment places a stronger emphasis on precision than recall.

In the latter strategy, we relaxed the threshold to some extent, opting to sacrifice precision in order to enhance recall. This approach aims to potentially decrease the likelihood of retrieving fewer false negatives while increasing the likelihood of retrieving a larger number of false positives. This threshold was defined as those instances with similarity scores within the top 90% of all similar CWEs for all methods. Consequently, some CWEs with insufficient similarity to the method were excluded. In this situation, certain methods, which had lower similarity to the collection of CWEs and were considered safe in the previous experiment, were now categorized as vulnerable.

It is worth noting that, for the few-shot experiments, we are looking for samples from the same BigVul dataset and when we do not find the relevant CWE samples in this dataset, we are randomly selecting a CWE sample to provide to the LLM.

Note that the implementation of this variant was with the intention of achieving a balance in the trade-off between precision and recall. Lowering the similarity threshold may increase recall as more CWEs are included, but it might reduce precision by introducing more false positives. On the other hand, raising the similarity threshold can improve precision by reducing false positives, but it might lower recall as fewer CWEs are included, potentially missing some true positive cases.

5.2. Results and Discussion

The final results of the LLM classification of methods are reported in Table 2, for both bimodal and unimodal DL models.

If we compare Table 2 with Table 1, we can notice an improved accuracy in most cases compared to the model provided with only source code. Additionally, the number of true negatives and false negatives increases while providing more examples, specifically in the few-shot scenario. While this increase gives a worse recall rate, it indicates that the model is using the provided auxiliary information to change its decision and shows improved accuracy.

5.2.1. Zero-shot vs. Few-shot Learning. Analyzing Table 2 against Table 1 reveals an overall improvement in accuracy for most cases when models are provided with additional context, such as source code paired with examples or semantic knowledge. Notably, the few-shot learning examples demonstrated superior accuracy compared to zero-shot examples. This suggests that incorporating domain knowledge can enhance the model's reasoning capabilities. However, the difference in accuracy between zero-shot and few-shot examples was minimal.

5.2.2. Threshold Adjustment and Sample Size. The impact of adjusting the similarity threshold to balance recall and precision was also examined. The results indicate that such adjustments did not significantly alter outcomes, implying that increasing the sample size does not necessarily yield more true positives. This finding highlights the complexity of fine-tuning model performance to balance precision and recall, suggesting that simply providing more data or lowering the similarity threshold does not straightforwardly improve model performance.

TABLE 2: The accuracy of LLM in detecting method-level vulnerability with *"bimodal-generated"* and *"unimodal-generated"* domain information. The circled numbers are associated with circled numbers in Figure 2.

Approach		TP	TN	FP	FN	Acc.	Pre.	Rec.	F1	F2
Bimodal										
Zero-Shot ❷	**Precision Focus**	4,681	1,493	4,726	729	45.58%	49.76%	63.91%	55.96%	60.47%
Zero-Shot ❷	**Recall Focus**	4,646	1,045	4,940	624	50.56%	48.47%	88.16%	62.55%	75.75%
Few-Shot ❸	**Precision**	2,896	4,169	2,883	2655	56.06%	50.11%	52.17%	51.12%	51.75%
Few-Shot ❸	**Recall**	3,603	4,047	2,944	2,643	57.79%	55.03%	57.68%	56.33%	57.13%
Unimodal										
Zero-Shot ❷	**Precision Focus**	4,829	1,049	5,394	702	49.09%	47.24%	87.31%	61.31%	74.64%
Zero-Shot❷	**Recall Focus**	4,927	937	5,373	563	49.69%	47.83%	89.74%	62.41%	76.37%
Few-Shot ❸	**Precision Focus**	3,247	4,190	2,888	3,088	55.45%	52.93%	51.25%	52.08%	51.58%
Few-Shot ❸	**Recall Focus**	3,369	3,873	2,905	2,576	56.92%	53.70%	56.67%	55.14%	56.05%

5.2.3. Unimodal vs. Bimodal Models. Comparison between unimodal and bimodal models revealed minimal differences. The unimodal model, which uses method descriptions provided by the LLM to detect similarity with corresponding CWEs, was nearly as effective as the bimodal model that directly compares source code with CWE descriptions. This reinforces that both types of additional information help the model to detect vulnerability.

5.2.4. Experimental Limitations and Practical Considerations. The decision not to explore other LLMs in this study was influenced by time limitations. Querying LLMs through an API is computationally expensive and time-consuming. Conducting the experiments in this paper took approximately 218 hours on three NVIDIA A100 Tensor Core GPUs with a memory bandwidth of 2TB.

It is important to note that the total number of instances might vary among different experiments due to certain challenges encountered during the process. Specifically, for both unimodal and bimodal scenarios, we used the LLM twice and encountered more vague results or errors in some instances. As a result, these instances were removed from the experiments. These exclusions were not intentional but were necessitated by technical issues encountered during the experimentation process. This issue should be considered when planning to utilize LLMs in real-world practical scenarios.

6. Threats to Validity

We acknowledge certain threats to the validity of our work. First, we conducted our experiments solely with one LLM. We thus cannot claim the generalizability of our findings to all LLMs. Not all LLMs permit automated querying processes, owing to the absence of an API. To mitigate this limitation, we carefully selected an LLM that has strong performance and has been used widely in the research community.

Another limitation of our approach is that we evaluated only one dataset in a specific language. Therefore, we cannot claim the generalizability of our results to other settings. However, our research serves as a starting point, and additional studies are needed to further explore and understand this area.

Additionally, because of ambiguous responses from the LLM, we were unable to generate true or false results for every sample. Importantly, no records were intentionally omitted from our study. A portion of the records was lost during the LLM steps due to the safety settings of the LLM, which were unchangeable at the time of the experiment. Further, some records were lost during the cosine similarity step as explained earlier.

In addition, we acknowledge the potential concern of data leakage, as the BigVul dataset was curated in 2020, and closed-source LLMs like Gemini-pro may have been trained on similar data sources. However, we did not observe any direct evidence of data leakage influencing the results of our experiments.

7. Conclusion and Future Work

In summary, our exploration of vulnerable code classification using LLMs has yielded insightful revelations regarding the intricacies of prompt engineering. Few-shot learning demonstrated superior performance compared to zero-shot learning, while the basic prompt detected most records as vulnerable, irrespective of their actual vulnerability status. Our study highlights the importance of domain knowledge and emphasizes the need for incorporating such information in prompt design.

Moving forward, multiple avenues for future research emerge from our current study. First, it is informative to expand the experimentation with multiple LLMs to provide a more comprehensive understanding of their generalizability in vulnerability detection tasks. Second, exploring the scalability of these models and their effectiveness across diverse programming languages and codebases is important. Third, qualitative analysis of false negatives to understand the drop in recall can provide additional insights.

The insights gained from this study lay the foundation for future endeavors in refining vulnerability detection methodologies using LLMs and advancing the understanding of prompt engineering in the context of code analysis.

8. Acknowledgement

The work in this paper was partially funded by the Office of Naval Research (ONR) (Grant#: G2A62826).

References

[1] "Cve details," https://www.cvedetails.com/, 2024, accessed: Aug 2024.

[2] S. Shimmi, A. Rahman, M. Gadde, H. Okhravi, and M. Rahimi, "VulSim: Leveraging similarity of Multi-Dimensional neighbor embeddings for vulnerability detection," in *33rd USENIX Security Symposium*, 2024, pp. 1777–1794.

[3] A. Khediri, H. Slimi, A. Yahiaoui, M. Derdour, H. Bendjenna, and C. E. Ghenai, "Enhancing machine learning model interpretability in intrusion detection systems through shap explanations and llm-generated descriptions," in *Proceedings of the 6th PAIS*, 2024, pp. 1–6.

[4] L. Zanella, W. Menapace, M. Mancini, Y. Wang, and E. Ricci, "Harnessing large language models for training-free video anomaly detection," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024.

[5] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "Inferfix: End-to-end program repair with llms," in *Proceedings of the 31st FSE*, 2023.

[6] M. D. Purba, A. Ghosh, B. J. Radford, and B. Chu, "Software vulnerability detection using large language models," in *the 34th ISSREW*, 2023, pp. 112–119.

[7] M. Fu, C. Tantithamthavorn, V. Nguyen, and T. Le, "Chatgpt for vulnerability detection, classification, and repair: How far are we?" *arXiv preprint arXiv:2310.09810*, 2023.

[8] X. Zhou, T. Zhang, and D. Lo, "Large language model for vulnerability detection: Emerging results and future directions," in *Proceedings of the 44th ICSE-NIER*, 2024, pp. 47–51.

[9] G. Lu, X. Ju, X. Chen, W. Pei, and Z. Cai, "Grace: Empowering llm-based software vulnerability detection with graph structure and in-context learning," *Journal of Systems and Software*, vol. 212, p. 112031, 2024.

[10] A. Cheshkov, P. Zadorozhny, and R. Levichev, "Evaluation of chatgpt model for vulnerability detection," *arXiv preprint arXiv:2304.07232*, 2023.

[11] C. Zhang, H. Liu, J. Zeng, K. Yang, Y. Li, and H. Li, "Prompt-enhanced software vulnerability detection using chatgpt," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, 2024, pp. 276–277.

[12] Y. Nong, M. Aldeen, L. Cheng, H. Hu, F. Chen, and H. Cai, "Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities," *arXiv preprint arXiv:2402.17230*, 2024.

[13] J. Gu, Z. Han, S. Chen, A. Beirami, B. He, G. Zhang, R. Liao, Y. Qin, V. Tresp, and P. Torr, "A systematic survey of prompt engineering on vision-language foundation models," *arXiv preprint arXiv:2307.12980*, 2023.

[14] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language model: Survey, landscape, and vision," *arXiv preprint arXiv:2307.07221*, 2023.

[15] W. Wang, V. W. Zheng, H. Yu, and C. Miao, "A survey of zero-shot learning: Settings, methods, and applications," *ACM TIST*, vol. 10, no. 2, pp. 1–37, 2019.

[16] S. Kadam and V. Vaidya, "Review and analysis of zero, one and few shot learning approaches," in *Proceedings of the 18th ISDA*, 2020.

[17] Y. Wang, Q. Yao, J. T. Kwok, and L. M. Ni, "Generalizing from a few examples: A survey on few-shot learning," *ACM computing surveys (csur)*, vol. 53, no. 3, pp. 1–34, 2020.

[18] W. Ren, Y. Tang, Q. Sun, C. Zhao, and Q.-L. Han, "Visual semantic segmentation based on few/zero-shot learning: An overview," *IEEE/-CAA Journal of Automatica Sinica*, 2023.

[19] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "Ac/c++ code vulnerability dataset with code changes and cve summaries," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 508–512.

[20] Z. Zhao, E. Wallace, S. Feng, D. Klein, and S. Singh, "Calibrate before use: Improving few-shot performance of language models," in *International Conference on Machine Learning*. PMLR, 2021, pp. 12 697–12 706.

[21] S. Pitis, M. R. Zhang, A. Wang, and J. Ba, "Boosted prompt ensembles for large language models," *arXiv preprint arXiv:2304.05970*, 2023.

[22] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.

[23] C. Chen, J. Su, J. Chen, Y. Wang, T. Bi, Y. Wang, X. Lin, T. Chen, and Z. Zheng, "When chatgpt meets smart contract vulnerability detection: How far are we?" *arXiv preprint arXiv:2309.05520*, 2023.

[24] S. Hu, T. Huang, F. İlhan, S. F. Tekin, and L. Liu, "Large language model-powered smart contract vulnerability detection: New perspectives," *arXiv preprint arXiv:2310.01152*, 2023.

[25] X. Du, G. Zheng, K. Wang, J. Feng, W. Deng, M. Liu, B. Chen, X. Peng, T. Ma, and Y. Lou, "Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag," *arXiv preprint arXiv:2406.11147*, 2024.

[26] T. Quinn and O. Thompson, "Applying large language model (llm) for developing cybersecurity policies to counteract spear phishing attacks on senior corporate managers," 2024.

[27] S. Pal, "Words worth millions: Decrypting financial text with gemini," *Available at SSRN 4861479*, 2024.

[28] S. M. Taghavi and F. Feyzi, "Using large language models to better detect and handle software vulnerabilities and cyber security threats," 2024.

[29] 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," *arXiv preprint arXiv:2002.08155*, 2020.

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

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

[32] X. Deng, Y. Gu, B. Zheng, S. Chen, S. Stevens, B. Wang, H. Sun, and Y. Su, "Mind2web: Towards a generalist agent for the web," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[33] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[34] P. Wu, L. Yin, X. Du, L. Jia, and W. Dong, "Graph-based vulnerability detection via extracting features from sliced code," in *the 20th QRS-C*, 2020, pp. 38–45.

[35] B. Mosolygó, N. Vándor, P. Hegedűs, and R. Ferenc, "A line-level explainable vulnerability detection approach for java," in *ICCSA Workshop*, O. Gervasi, B. Murgante, S. Misra, A. M. A. C. Rocha, and C. Garau, Eds., 2022, pp. 106–122.

[36] J. Ngiam, A. Khosla, M. Kim, J. Nam, H. Lee, and A. Y. Ng, "Multimodal deep learning," in *Proceedings of the 28th ICML*, 2011, pp. 689–696.

[37] S. Shimmi and M. Rahimi, "Mining software repositories for patternizing attack-and-defense co-evolution," in *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security*, 2022.

Appendix

1. Keywords

1.1. Keywords to detect vulnerable label. ”yes”, ”is vulnerable”, ”does not validate”, ”doesn’t validate”, ”doesn’t allocate”, ”input validation”, ”does not check”, ”is likely to be applicable”, ”likely vulnerable”, ”doesn’t perform”, ”doesn’t seem to perform”, ”function does not perform”, ”could be a security issue”, ”has a potential issue”, ”may be vulnerable”, ”is associated with”, ”is potentially vulnerable to”, ”cwe-”, ”*cwe-”, ”**cwe-”, ”***cwe-”, ”###cwe-”, ”####cwe-”, ”could be vulnerable”, ”seems to be vulnerable”, ”doesn’t check”, ”appears to be vulnerable”, ”potential vulnerability”, ”potentially be vulnerable”, ”are vulnerable to cwe are”, ”might be vulnerable”, ”is a potential cwe”, ”is a potential vulnerability”, ”vulnerable to cwe”, ”violation of cwe-”, ”Possibly contains a”, ”could occur”, ”could lead to”, ”can lead to”, ”might lead to”, ”has a potential cwe”, ”could be exploited”, ”without checking”, ”This function has a”, ”following Common Weakness Enumeration”, ”suffer from cwe”, ”related to cwe”, ”it’s vulnerable to”, ”is susceptible to”, ”can be a security risk”, ”potentially be vulnerable”, ”To mitigate this vulnerability”, ”to fix this issue”, ”list of vulnerable cwe’s”, ”potential violation of cwe”, ”contains a cwe”, ”the following cwe”

1.2. Keywords to detect non-vulnerable label. ”not vulnerable”, ”appears to be non-vulnerable”, ”doesn’t exhibit any vulnerabilities”, ”doesn’t seem to have any vulnerabilities”, ”doesn’t seem to be vulnerable”, ”doesn’t appear to be vulnerable”, ”doesn’t have the issue”, ”doesn’t appear to have”, ”doesn’t contain any”, ”doesn’t present any vulnerabilities”, ”doesn’t seem vulnerable”, ”doesn’t have a vulnerability”, ”doesn’t seem to have any obvious vulnerabilities”, ”doesn’t look vulnerable”, ”doesn’t have any”, ”looks fine”, ”appears to be free of any”, ”doesn’t meet the criteria for cwe”, ”unlikely to be vulnerable”, ”does not appear to be vulnerable”