Final Project: Leveraging a Local Large Language Model For Vulnerability Detection and CWE Classification

Angelo Porcella, Emery Call, Jasmine Vang, Prakriti Baral

November 18th, 2024

1 Introduction

In software security, recognizing vulnerabilities is a highly useful step. Traditional methods like as static and dynamic analysis are being supplemented with Large Language Models (LLMs), such as OpenAI's ChatGPT, which excel at detecting code problems. However, their reliance on proprietary APIs raises questions about privacy, cost, and accessibility.

In this paper, we focus on leveraging a local LLM for vulnerability detection and Common Weakness Enumeration (CWE) classification. By "local LLM," we refer to a language model capable of running efficiently on consumer-grade hardware, without the need for cloud services or specialized infrastructure. This approach offers several advantages, including enhanced privacy for sensitive codebases, cost savings by eliminating subscription fees, and the ability to operate offline in restricted environments.

2 Related Work

2.1 Zhang '17

This paper titled "Prompt-Enhanced Software Vulnerability Detection Using ChatGPT" [3] looked at different prompt engineering/training strategies for use specifically in LLM-based vulnerability detection. The paper discusses

the implementation of "in-context-learning" and "chain-of-thought" prompting wherein the model is guided through an input before giving an answer. In the chain-of-thought method, the model analyzed the intent of the code initially, before making a decision on if the code contained a vulnerability. The paper determined that the chain-of-thought method was the most accurate prompting method. We replicated the chain-of-thought method and in-context-learning method in our experiment. In contrast to the paper, of the two prompting methods we implemented we found in-context-learning to have a higher F1 score than the chain-of-thought method.

2.2 Chen '23

The DiverseVul paper [1] provides some perspective for training LLMs to detect vulnerabilities as opposed to using pre-trained models not purpose built for vulnerability detection as we are in this paper. They achieved an F1 Score of 47.15 and false positive rate of 3.47% on their DiverseVul dataset using a 220M parameter model. They found that even more important than model size and training dataset size was coding related pre-training outside vulnerability detection.

3 Leveraging a Local Large Language Model For Vulnerability Detection and CWE Classification

3.1 Setup

The datasets used in testing were taken from [4] and [1] as mentioned in related works. Our paper aimed to replicate the [4]'s results for vulnerability detection with an LLM but using an easier to run local LLM instead of ChatGPT. To be able easily comparable, we used the same test set as the [4] for testing vulnerability detection. We then also expanded upon this paper by also trying to classify vulnerable code with a top 25 CWE. This second classifier uses a different prompt to get the LLM to respond with a singular CWE from the top 25. It is tested on the DiverseVul [1] dataset instead because the dataset used by the [4] doesn't include CWE labels.

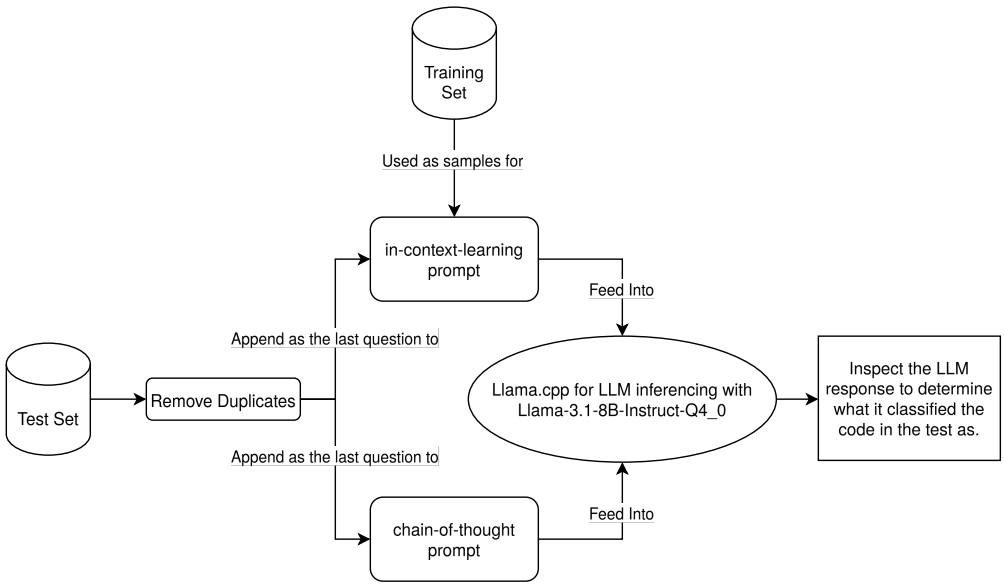

Figure 1: Our LLM Classifier Architecture

Our entire architecture for prompting the LLM to classify code snippets is summarized in the above figure 1. We use two of these, one for vulnerability detection that uses the [4] test and training set and another for the CWE classification which uses the DiverseVul dataset [1]. We created a training and testing set from DiverseVul by taking a random down-sampling of code snippets that had CWEs in the top 25 according to the MITRE 2021 list [2].

3.2 Methods

Utilizing the datasets, tests were run in batches using two different prompting variants. For vulnerability testing, both in context learning (ICL) and chain of thought testing (COT) were run to compare these two prompting styles and to determine the most accurate method between them. For the CWE classification module, the tests were run with the same two prompting variants. Additionally, we found that including the official CWE descriptions help it get some of the CWEs that it couldn't before so we included the descriptions for both variants. By trying multiple prompt variants we cover a wider swath of prompt engineering methods to give a better idea of the overall performance of the model.

Picking which LLM to perform these two classifications with took some

experimentation to see what models had potential. One restriction we had was that it needed to fit well within 16 Gb of RAM in our personal machines so it would avoid using swap space which would massively increase the time it took to run the model. We initially attempted to use the Llama2 based Code Llama Instruct 7 Billion model because it had been trained an extra amount specifically on code. However, after some initial testing, we found it lacked enough general knowledge to be able to perform CWE classification. It seemed to assume any C code likely had some kind of buffer overflow making it unusable. Unfortunately, no new Llama 3 based Code Llama exists at the time of writing that is a similar model size. This lead us to try out the newer, but general purpose, Llama3.1 Instruct 8 Billion model. While lacking specific training on code, in our initial testing, this model showed much greater CWE knowledge and understanding of code. So we opted to use it for this paper.

As for running the LLM, we opted to use llama.cpp, an easy to use LLM inference engine that excels at running on a wide variety of hardware. This means that while potentially slow, the testing we performed can be run on even very mediocre consumer hardware. We ran all of the testing on our personal computers and llama.cpp was only every configured to use the CPU for simplicity.

3.2.1 Test Running

The batches run made sure to ignore any duplicate entries in the dataset and we ran each prompt as a "one shot" test case. This means the model was reset each time as to not bias later tests by having the model train itself via it's own outputs through multiple prompting rounds. This we believe is a better "real world" representation of the performance of the model. Notably, [4] had many duplicates in it's vulnerability testing dataset that as far as we can tell were not ignored.

One problem we had to solve was that in the DiverseVul dataset we used for CWE classification there were multiple CWE labels in the top 25. One solution to this could be to have the LLM produce multiple CWEs it predicts are in the code snippet. However, to keep things simple, we instead opted to have the LLM predict one CWE per code snippet and if it got one of the correct CWEs then we would assume that singular CWE was the correct answer for that code snippet. If the LLM predicted the wrong CWE then we would randomly distribute that singular mis-classification to one of the

correct CWEs.

4 Discussion

Our experiment yielded two distinct findings: one centered on the accuracy of the LLM in classifying code as vulnerable or not, and the other on its ability to identify the CWE associated with the code. The datasets were individually selected for the experiments, each chosen as a point of comparison to similar published research.

4.1 Vulnerability Detection

The data set and prompts from [4], and prompt engineering from [3] construct our experiment set-up. Our prompt utilizes the same role description prompt as [4]. The results of [4] can seen in Table 1 as Method Paper A.1. In addition to providing our LLM with a role description, we analyzed the use of two prompting methods: in-context-learning (ICL) and chain-of-thought (COT). Both prompting methods utilized the same role description. In each of the methods that we implemented, the LLM's objective was to determine if the provided code was vulnerable or not vulnerable.

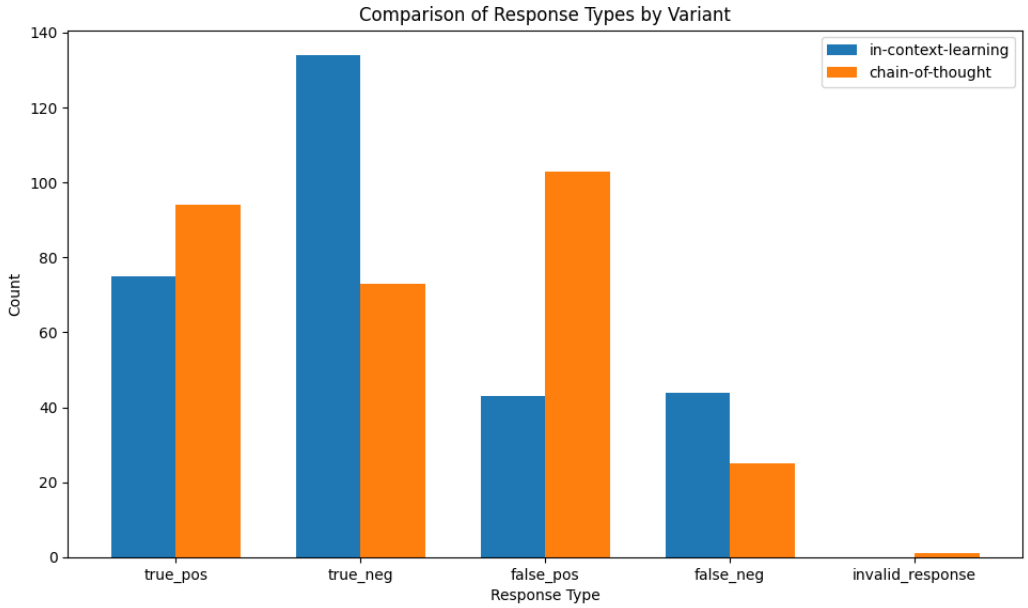

Figure 2: Vulnerability Classification

Figure 2 portrays the LLMs performance provided the prompting method ICL or COT. The results vary, with ICL out performing COT in true negatives, but COT performing better in true positives. The same variances arises with COT having more false positives and ICL having more false negatives. COT contained a single invalid response, in which it defied the instructions for the output format. It is unclear exactly what caused this.

Method	**Precision (%)**	**Recall (%)**	**F1 (%)**
In-Context Learning	63.56	63.03	63.29
Chain-of-Thought	47.72	78.99	59.49
Paper A.1	NaN	0.00	NaN
Paper A.2	80.0	30.3	43.9

Table 1: Comparison of Precision, Recall, and F1 Scores

The performance of each method is accurately represented in 1 where the precision, recall and F1 scores can be easily compared. Method Paper A.1 and Paper A.2 are both from [4], where Paper A.1 prompts are provided with

a role description and Paper A.2 prompts have both a role description and five random code samples from the training data set. These were selected to compare to because Paper A.1 is a minimal baseline prompt and Paper A.2 directly inspired our ICL prompt.

The precision, recall, and F1 scores represented in 1 show that ICL has a balanced performance, which indicates a consistent ability to identify positives and avoid false positives. COT has a high recall but low precision, suggesting the model identifies many true positives but also makes many false positive predictions. Paper A.1 noted in [4] that the method failed to identify any true positives and only identified true negatives, Paper A.2 has a high precision where it is able to avoid false positives, but the recall suggests that it also misses many true positives.

Both methods Paper A.1 and Paper A.2 struggled to identify true positives, where as our methods were both capable of more accurately identifying true positives. The main difference between [4] methods and our methods is the use of additional prompt engineering. This difference shows that the use of ICL or COT methods in conjunction to Paper A.1 and Paper A.2 methods have promising results.

4.2 CWE Classification

The top 25 CWE classification prompts are modified from the vulnerability classification to have context regarding CWE's, the prompts maintain the use of role assignment. The dataset used for CWE classification is sourced from [1].

We collected the following two tables (4, 3) for either prompt variant that contain the micro-average aggregate for all CWEs as well as the individual performance for each CWE classification. Overall, both variants performed similarly poorly around a 0.10 F1 score with only a ˜0.01 difference in their F1 scores. Additionally, both tables are contain mostly None values representing whenever we would get a divide by zero while computing recall and precision. This typically happens because there were no true positives for predicting that CWE. Meaning, the model failed to predict that CWE at all. The exception to this is CWE-306, which had no instances in the DiverseVul dataset so we expect it to never be predicted.

Variant: in-context-learning

CWE	Recall	Precision	F1
All CWEs	0.10776942355889724	0.10804020100502512	0.10790464240903387
CWE-119	0.37037037037037035	0.10638297872340426	0.1652892561983471
CWE-125	0.13333333333333333	0.07692307692307693	0.0975609756097561
CWE-190	None	None	None
CWE-20	0.24242424242424243	0.0975609756097561	0.1391304347826087
CWE-200	None	None	None
CWE-22	0.25	0.4	0.3076923076923077
CWE-276	None	None	None
CWE-287	None	None	None
CWE-306	None	None	None
CWE-352	None	None	None
CWE-416	0.0625	0.25	0.1
CWE-434	None	None	None
CWE-476	0.19047619047619047	0.0625	0.09411764705882353
CWE-502	0.14285714285714285	1.0	0.25
CWE-522	None	None	None
CWE-611	None	None	None
CWE-732	None	None	None
CWE-77	None	None	None
CWE-78	0.2727272727272727	0.14285714285714285	0.18749999999999997
CWE-787	0.15	0.08108108108108109	0.10526315789473685
CWE-79	None	None	None
CWE-798	0.25	0.1875	0.21428571428571427
CWE-862	None	None	None
CWE-89	None	None	None
CWE-918	None	None	None

Figure 3: CWE Classification using In Context Learning Performance Table

Variant: chain-of-thought

CWE	Recall	Precision	F1
All CWEs	0.11027568922305764	0.11027568922305764	0.11027568922305764
CWE-119	0.8235294117647058	0.10606060606060606	0.18791946308724833
CWE-125	None	None	None
CWE-190	None	None	None
CWE-20	0.11764705882352941	0.14814814814814814	0.13114754098360654
CWE-200	None	None	None
CWE-22	0.125	0.6666666666666666	0.21052631578947367
CWE-276	None	None	None
CWE-287	None	None	None
CWE-306	None	None	None
CWE-352	None	None	None
CWE-416	None	None	None
CWE-434	None	None	None
CWE-476	0.1	0.2	0.13333333333333333
CWE-502	None	None	None
CWE-522	None	None	None
CWE-611	None	None	None
CWE-732	None	None	None
CWE-77	None	None	None
CWE-78	0.3333333333333333	0.1282051282051282	0.18518518518518517
CWE-787	None	None	None
CWE-79	0.75	0.12	0.20689655172413793
CWE-798	None	None	None
CWE-862	None	None	None
CWE-89	None	None	None
CWE-918	None	None	None

Figure 4: CWE Classification using Chain of Thought Prompting Performance Table

One difference between the variants is that in context learning had fewer None values indicating that it was able to get at least one true positive for certain CWEs that chain of thought was not able to (19 Nones to 15 Nones). Furthermore, of the CWEs that now had a true positive, half of them (CWE-416 and CWE-787) were in the 5 in context learning samples used. This indicating that using more samples for in context learning could potentially help it cover even more CWEs.

The results that we achieved show some comparisons to [1], particularly in the models' ability to identify certain CWE's more accurately. It was shown in [1] that some CWE's are easier to learn regardless of the training data size. Distinct from our methods, [1] provides pretraining on code and c/c++, but regardless, both struggle with certain CWE's. In [1] there are

several CWE's that have a zero percent true positive rate, each of which our methods also have a zero percent true positive rate. Additionally, the CWE's that [1] had low (below 20%) true positive rates, our methods would typically show a zero percent true positive rate. Although our methods struggled, [1] shows that the results are expected.

Method	**Precision (%)**	**Recall (%)**	**F1 (%)**
In-Context Learning	10.80	10.78	10.79
Chain-of-Thought	11.03	11.03	11.03

Table 2: CWE Comparison of Precision, Recall, and F1 Scores

Table 2 shows the precision, recall, and F1 score for our methods. Reviewing table 2 it is deduced that COT marginally outperforms ICL in all metrics but doesn't show a significant improvement. Both methods struggle to classify CWE's proving that CWE classifying is difficult for GPT models regardless of these applied methods. COT's balanced scores shows potential to continue exploring this method, but regardless, both of the methods perform unsatisfactory.

4.3 Future Work

Our results show the use of local LLM's for vulnerability classification has promising results. However, our experiment only makes use of minimal experiment variants. We use one data set, and two prompts (one for ICL and one for COT). Future work can expand the experiment by using several data sets. Additionally, both ICL and COT used the same prompt for each test case, future projects can expand the ICL and COT methods by experimenting with several prompt variants.

The CWE classification proved to be difficult for the LLM with our two methods. Even though the performance is unsatisfactory, both the COT and ICL methods have balanced scores. With our work only exploring two methods, there is the potential to analyze other methods to determine if there is higher success in CWE classification.

5 Conclusions

Our study shows the use of local Large Language Models (LLMs) for vulnerability identification and CWE classification, with a focus on privacy and computational efficiency on consumer hardware. Using llama.cpp, the Llama3.1 Instruct 8B model demonstrated scalability and usability with increased token limits. In-Context Learning (ICL) surpassed Chain-of-Thought (COT) in terms of precision and F1 scores for vulnerability detection, although COT had a greater recall. However, both techniques struggled with CWE categorization, yielding F1 scores of around 11 percent, showing a limited capacity to reliably identify distinct CWE categories. These findings illustrate the limitations of general-purpose LLMs in handling complicated security tasks, emphasizing the need for domain-specific pretraining and larger datasets to improve classification performance.

References

[1] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection, 2023.

[2] The MITRE Corporation. 2021 CWE Top 25 Most Dangerous Software Weaknesses. MITRE Corporation, 2021. https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html.

[3] Chenyuan Zhang, Hao Liu, Jiutian Zeng, Kejing Yang, Yuhong Li, and Hui Li. Prompt-enhanced software vulnerability detection using chatgpt. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ICSE-Companion '24, page 276–277, New York, NY, USA, 2024. Association for Computing Machinery.

[4] Xin Zhou, Ting Zhang, and David Lo. Large language model for vulnerability detection: Emerging results and future directions. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER'24, page 47–51, New York, NY, USA, 2024. Association for Computing Machinery.