VuLLM: Vulnerability detection with Large Language Models

Ivar Nelson
Blekinge Institute of Technology
Blekinge, Sweden
inve20@student.bth.se

Oscar Andersson
Blekinge Institute of Technology
Blekinge, Sweden
osan20@student.bth.se

***Abstract*— In this work, we present VuLLM, an adapter for the CodeLlama (7B) model that is fine-tuned on vulnerability detection. We investigate the performance of using zero shot, instruction tuning and fine-tuning on a custom classification head. We compare the classifier to the state of the art on the REVEAL dataset which consists of 22,734 labeled examples. Recent works has focused on applying graph neural networks for vulnerability detection. We revisit the use of token based models for this purpose and hypothesise that these large models have learned a rich representation for code. We beat the best token based method with an absolute F1 of 14.2%. This is an early work in applying LLMs to this problem, and we believe that more progress can be made. All code is made publicly available at https://github.com/ivos-projects/VuLLM**

I. Introduction

This paper explores the use of pre-trained language models for the task of vulnerability detection. The problem can be represented as a binary classification problem. Given a piece of code, the task is to determine if it contains a vulnerability or not.

This problem is important because vulnerabilities in software can be exploited by attackers to gain access to sensitive information or to take control of the system. The consequences of such attacks can be severe. For example, in 2017 a vulnerability in the Apache Struts framework was exploited to steal personal information from 147 million people[9]. The vulnerability was discovered and patched, but only after the attackers had gained access to the data.

There are two main approaches to the problem of vulnerability detection. The difference lies is in how the code is represented. The first approach is to represent the code as a graph and use graph neural networks to classify the code[8][10]. The motivation to this approach is that the source code has structure does not have to be learned without prior knowledge. The second approach is to represent the code as a sequence of tokens and use language models to classify the code[5]. Recently the graph based methods have been shown to outperform the token based methods. However, the token based methods have not been evaluated using the latest pre-trained language models.

Large language models that are trained on large amounts of code may have gained the intrinsic structure of code and can therefore make better use of the second approach to vulnerability detection.

The recent development of pre trained language models has made it possible to use them for a wide range of tasks. The pre-trained language models are trained on huge amounts of data and can be fine tuned to perform well on a specific task. In this work we focus on a specific pre-trained language model, CodeLlama[6], which is trained very large amounts of code and text.

II. Background

When fine-tuning a pre-trained language model, there are several choices to be made. The first choice is which pre-trained language model to use. The second choice is how to fine tune the pre-trained language model. To limit the scope of this paper, we focus on the pre-trained language model CodeLlama which comes in several configurations. The main difference is in the number of parameters used. We will focus on the 7B variant in two configurations, namely CodeLlama and CodeLlama-instruct and experiment with different ways of fine-tuning the pre-trained language models. To lay the foundation for the experiments, we will first describe the different fine-tuning methods as well some theory associated with LLMs.

A. Language Modeling

Language modeling refers to the task of predicting the next word or sequence of words in a given context. It is a fundamental problem in natural language processing (NLP) and plays a crucial role in various NLP applications such as machine translation, text generation, and sentiment analysis.

One of the key advancements in language modeling is the transformer architecture. The transformer is a deep learning model that has revolutionized NLP tasks. It was introduced in a seminal paper called “Attention is All You Need”[7].

The transformer architecture is based on the concept of self-attention, which allows the model to weigh the importance of different words in a sentence when making predictions. Unlike traditional recurrent neural networks (RNNs) that process words sequentially, the transformer can consider the entire context simultaneously. This parallel processing capability makes the transformer highly efficient and enables it to capture long-range dependencies in the text.

The transformer consists of an encoder and a decoder. The encoder takes the input sequence and generates a representation called the ”contextualized word embeddings” These embeddings capture the meaning and context of each word in the sequence. The decoder then uses these embeddings to generate the next word in the sequence.

B. Fine tuning

Fine tuning is a technique used to adapt a pre-trained language model to a specific task. The pre-trained language model is first trained on a large corpus of text. The pre-trained language model is then fine tuned on a specific task by training it on a dataset for that task. There are multiple ways to fine-tune a pre-trained language model. We will explore them in the following sections.

1) Causal vs Masked Fine tuning: Causal language modeling refers to the task of predicting the next word in a sequence given only the previous words. Masked language modeling uses both context to the right and left of the word to predict the word. When the model is fine tuned on a specific dataset, the model is trained to predict the next word in the sequence. One can then construct a template where the word to be predicted is masked out. This usually requires less examples and can be used in cases of zero shot classification.

2) Custom classification head: Another approach is the use a custom classification head. For binary classification, this can be mapping to two logits for the last layer. There are two main choices, either to freeze the pre-trained language model and only train the classification head, or to train the entire model. This approach is called sequence classification.

3) Lora, PEFT and Quantization: The process of fine-tuning a language model has one major drawback: it is computationally expensive and requires a of memory. The base-model of CodeLlama has 7 billion parameters. Using fp16 precision, the model requires 13 GB of VRAM. For fine-tuning, you also need to store the weights for the backward pass, which doubles the memory requirement. The larger batch size you can use, the more GPU memory you need. This makes it difficult to fine-tune the model on a single GPU. However, recently a paper called “Lora: A Low-Rank Adaptive Method for Efficient Large Language Models”[4] was published. The paper introduces a method for reducing the memory requirement of fine-tuning a language model. The method hypothesizes that the weight updates of the language model can be approximated by a lower rank. Instead of calculating the gradients with respect to the full weight matrix, the gradients are calculated with respect to two vectors, x and y, which are then multiplied to approximate and summed with the original weights. This makes the the entire process diffrentiable. If x and y are of size n, the memory requirement is reduced from n^2 to $2n$.

Under the fine-tuning phase, the original weights of the base-model is freezed and the new lora weights are optimized. Adapter is another work to describe this concept. Using this approach we don’t need to store the entire 7B parameters for each specialized model. Instead we can store different adapters and add them on top of the base model. For a 7B model, 300MB is a reasonable size for on adapter compared to 13BG for the entire model.

To further reduce the memory requirement, the original weights can be quantized.

III. VuLLM

A. Model

Figure 1 illustrates the two distinct configurations of the VuLLM (Vulnerability Language Model). In both instances, the CodeLlama 7B serves as the foundational model but in two different levels of fine-tuning. For sequence classification the base version of CodeLlama 7B is used and for causal classification CodeLlama 7B instruct is used. It is a version that has been further fine-tuned on instructions and chat like interactions and is more well suited to answering questions.

Our hypothesis posits that CodeLlama inherently encapsulates the intrinsic structural attributes of code, thereby enabling the proficient handling of long-range dependencies within a given code section. In other words, we contend that CodeLlama has acquired the capacity to discern the relationships among various components of code, obviating the necessity for additional representations beyond the source code itself. The deployment of two distinct configurations is used for experiments. From these two configurations, the following reseach objectives are formulated.

- **RQ1:** How does a fine-tuned sequence classifier compare to previous work on the REVEAL dataset?
- **RQ2:** How does the causal classifier compare to previous work on the REVEAL dataset, with varying degrees of fine-tuning?

We now go into more detail of the two different configurations that will address the research questions.

1) Sequence Classification Model: The Sequence Classification Model is characterized by a base model with 32,016 output logits, each corresponding to individual tokens. In the context of sequence classification, the conventional language modeling head is substituted with a configuration featuring only two newly initialized logits, as illustrated in figure 1. Notably, the entire model undergoes quantization to 4 bits, and Linearity Rectified Attention (LoRA) weights are applied to all linear layers, excluding the final layer in the architectural hierarchy. The classifier will undergo training with the maximal amount of available data, since the language modeling head is replaced by only two logits that represent vulnerable and not vulnerable. Comparing the results of a fine-tuned sequence classifier with previous work can shed light on whether or not our base model has knowledge of the intrinsic structure of code and thus is able to handle long range dependencies.

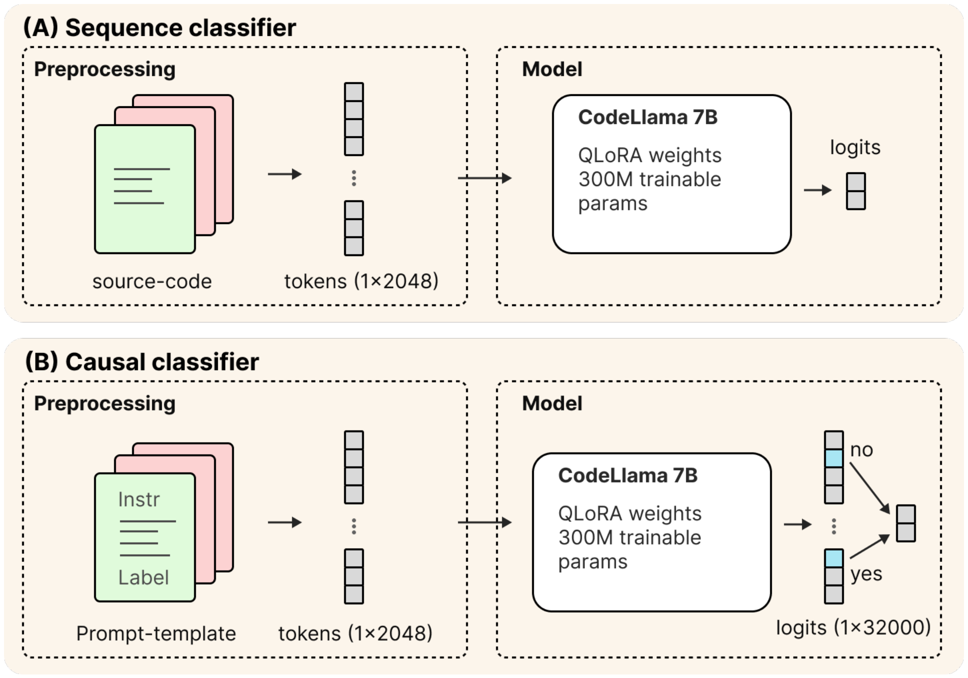

Fig. 1. Architecture of the two different models used in VuLLM. (A) shows the sequence classifier which is trained with a custom classification head. (B) shows the causal classifier which outputs a probability distribution over all tokens, this is them used to make a prediction.

2) Causal Classification Model: This methodology employs the base model featuring the original language model head comprising one logit for each of the 32,016 tokens. During the classification process, the model designates target tokens and selects the logit associated with the highest value as the predictive outcome. To guide the model towards specific tokens, a predetermined prompt template is employed, as elucidated in the Appendix. During inference, subsequent tokens are omitted, and the model generates a probability distribution for the subsequent tokens. The template can be utilized for zero-shot classification or employed in fine-tuning through multiple templates prior to engaging in the inference phase. Moreover, the model undergoes quantization to 4 bits, and Linearity Rectified Attention (LoRA) weights are applied to all layers, excluding the language modeling head.

B. Dataset

The REVEAL dataset[1] is a dataset of C/C++ source-code functions labelled as either vulnerable or not vulnerable. The dataset has 22,734 instances, however it is highly imbalanced with 90% of the examples belonging to the non-vulnerable class, thus making it easy to achieve high accuracy by just predicting the non-vulnerable class. Therefore, F1 score is used as the main metric for evaluating the models. The dataset is split into a training set, validation set and a test set in the same manner as previous work using a ratio of 90% of instances for training and reserving 10% each for validation and testing. The split is made in a stratified fashion to preserve the class distribution and the dataset has been made available on the Hugging Face platform to facilitate accessibility.

C. Model evaluation

Precision assesses positive prediction accuracy, recall gauges identification of actual positives, and the F1 score balances precision and recall. Accuracy measures overall correctness.

$$\textbf{Precision} = \frac{\text{True Positives}}{\text{True Positives + False Positives}}$$

$$\textbf{Recall} = \frac{\text{True Positives}}{\text{True Positives + False Negatives}}$$

$$\textbf{F1 Score} = 2 \times \frac{\text{Precision}\times\text{Recall}}{\text{Precision + Recall}}$$

$$\textbf{Accuracy} = \frac{\text{True Positives + True Negatives}}{\text{Total Instances}}$$

These metrics collectively provide a nuanced evaluation of the model's predictive performance.

1) Context size: The context window for training and inference can differ on

D. Implementation details

During training a batch size of 4, coupled with 8 gradient accumulation steps was used, effectively simulating a batch size of 32. The learning rate is initialized at $2e{-}4$ and diminishes linearly as training progresses. All experiments were conducted on an Nvidia RTX-3090 with 24GB of VRAM.

The causal model has been tested in three configurations; No fine-tuning (causal-zero-shot), fine-tuning on 50 examples for each class (causal-100) and 500 examples for each class (causal-1000).

The sequence classifier version has been trained on either a dataset that has been down sampled to be balanced or a dataset with the equivalent number of examples (3000) but with the overall class distribution intact. Since some examples contains more tokens than the model is able to handle, we sample only examples that are shorter than 2048 tokens for training. For evaluation and testing any examples that are longer than the context window are truncated not to skew the comparison with other models.

IV. Results

The results in table III have not been reproduced by us. We used the results from AMPLE [8]. To compare our results to related work we use the same 80-10-10 training-validation-test split as in AMPLE but we were unable to find the specific random seed. Therefore the results should not be seen as incontrovertible.

Table I shows the result on all the configurations of VuLLM.

Table II illustrates the performance of a hypothetical classifier on the test set. Either acting as a equal weighted random classifier or one that has learned the class distribution of the training set.

In table III the results from the best VuLLM model is presented along side the results of previous work.

Model	Accuracy	Precision	Recall	F1
causal-zero-shot	0.89	0.29	0.09	0.14
causal-100	0.14	0.09	0.87	0.17
causal-1000	0.54	0.15	0.76	0.25
sequence-balanced	0.752	0.258	0.804	0.390
sequence-unbalanced	0.893	0.425	0.254	0.318

TABLE I
Metrics for all the configurations of VuLLM on the REVEAL dataset. All the results are performed on a test set that has not been seen by the model during the training or evaluation process

Classifier	Accuracy	Precision	Recall	F1
50/50	0.518	0.097	0.469	0.161
Weighted	0.825	0.112	0.112	0.112

TABLE II
Results on test set with dummy classifier.

Model	Accuracy	Precision	Recall	F1
VulDeePecker	0.764	0.211	0.131	0.162
Russel et al.	0.685	0.162	0.527	0.248
SySeVR	0.743	0.401	0.249	0.307
Devign	0.875	0.316	0.367	0.339
Ample	0.927	0.511	0.462	0.485
sequence balanced	0.752	0.258	0.804	0.390

TABLE III
Comparing performance of models on REVEAL dataset. Note that the random seed from which to generate the splits have not been obtained.

V. Discussion

In this section we will address the research questions RQ1 and RQ2 as well as future work. This work has showed a proof of concept. There is a vast space of hyperparameters and models to choose from. Because of time constraints and lack of computational resources, we have not explored the whole space, but still produced some impressive results.

A. *RQ1, The Effectiveness of Sequence classifier*

The results from the two versions of a sequence classifier that was trained are shown in I. We argue that the sequence classifier that has been trained on a balanced training set has the best performance. This is mostly due to the high recall it exhibits. For the task of detecting vulnerable code we certainly want to catch as many as positives as possible, which is why we emphasize recall specifically. Table III shows a comparison the other works and best performing VuLLM model, 'sequence-balanced', version fares relatively well. While not generating the highest accuracy or F1 score the model has the highest recall and a ratio of recall and precision that is less unbalanced than some other models with high recall, such as the model from Russel et al. As discussed in section III a prerequisite for a functioning sequence classifier is that the base model already has captured the structure of the computer code that you are trying to analyze and since the results are fully comparable with previous work this appears to be the case.

Answer to RQ1: A fine-tuned token based sequence classifier appears to have the capacity to reach state of the art performance

B. *RQ2, The Effectiveness of Causal Classifier*

Table I shows the result three different causal models that where trained. One of the model was tested in zero shot manner, where no previous training was done. As depicted by the table, more training resulted in a higher F1 score. One of the underlying hypotheses of RQ2 was that the model would have a concept of vulnerable code, and would not need to be fine-tuned to make predictions. The results indicate that this is not true and that more training results in favourable performance. To get a deeper understanding of how the causal model works, the 5 tokens with the highest probability were analyzed. For the zero shot classification, the most probable token was always the representation of the word "This". This is due to the fact that we are using the instruct version of CodeLlama where the goal is not just to output one token but instead formulate a coherent answer. It is possible that the model would achieve a higher F1 score if the complete generation was analyzed instead of only selecting the probabilities for the True and False tokens. Further analysis of the most probable tokens, as well as the result from table IV shows that the causal model trained on 1000 examples show much better performance. It shows an accuracy of 0.54 compared to 0.14 for causal-100. Therefore it is not unlikely that the causal-100 model simply predicts the positive class.

Compared to the other models causal 1000 performs the best with regards to F1, however it is lacking in the accuracy compared to other models. This is most probably how the data for the training is selected. Since The LLM is expensive to train, we have selected a balanced split, which makes the distribution different in the training setting and the validation and test setting.

Answer to RQ2: The Causal model does not perform close to the state of the art on the REVEAL dataset. The model might be to small to capture the concept of vulnerable code.

C. Future Work

The speed at which the field of deep learning is moving is staggering. Work on hardware optimizations such as flash attention [2] and selective state space model like mamba [3] is decreasing the memory requirements of large transformers. This makes it possible to use longer context windows and thus capturing dependencies within code that is further apart. An interesting avenue of research would be to pursue repository level context windows in order to detect possible emergent behaviour of the code that is not clear from a perspective of separate functions. For instance, an import of a library that contains vulnerabilities may be harmless depending on how that import is handled.

Regarding the causal modeling, fine-tuning improved the results significantly as the model learned the prompt template and that answers should be provided on a certain format. As discussed the output from the zero-shot model was longer and more nuanced than the output from the fine-tuned models. To make a more fair comparison of the zero-shot results it might be better to perform a sentiment analysis on the output rather than simply comparing the probabilities of the true and false tokens.

One limitation to the use of LLMs is the fact that many models, while available as open source models, are trained on proprietary datasets. Not knowing what training data the model has seen complicates further fine-tuning and makes testing especially troublesome. It is entirely possible that CodeLlama has used the REVEAL dataset as part of it‘s training data and thus rendering our results biased.

References

[1] Saikat Chakraborty et al. *Deep Learning based Vulnerability Detection: Are We There Yet?* Sept. 3, 2020. arXiv: 2009.07235[cs]. URL: http://arxiv.org/abs/2009.07235 (visited on 12/05/2023).

[2] Tri Dao. *FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning*. July 17, 2023. arXiv: 2307.08691[cs]. URL: http://arxiv.org/abs/2307.08691 (visited on 01/05/2024).

[3] Albert Gu and Tri Dao. *Mamba: Linear-Time Sequence Modeling with Selective State Spaces*. Dec. 1, 2023. arXiv: 2312.00752[cs]. URL: http://arxiv.org/abs/2312.00752 (visited on 12/15/2023).

[4] Edward J. Hu et al. *LoRA: Low-Rank Adaptation of Large Language Models*. Oct. 16, 2021. arXiv: 2106.09685[cs]. URL: http://arxiv.org/abs/2106.09685 (visited on 12/05/2023).

[5] Zhen Li et al. "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection". In: *Proceedings 2018 Network and Distributed System Security Symposium.* 2018. DOI: 10.14722/ndss.2018.23158. arXiv: 1801.01681[cs]. URL: http://arxiv.org/abs/1801.01681 (visited on 12/05/2023).

[6] Baptiste Rozière et al. *Code Llama: Open Foundation Models for Code*. Aug. 25, 2023. arXiv: 2308.12950[cs]. URL: http://arxiv.org/abs/2308.12950 (visited on 12/05/2023).

[7] Ashish Vaswani et al. *Attention Is All You Need*. Aug. 1, 2023. arXiv: 1706.03762[cs]. URL: http://arxiv.org/abs/1706.03762 (visited on 12/05/2023).

[8] Xin-Cheng Wen et al. *Vulnerability Detection with Graph Simplification and Enhanced Graph Representation Learning*. Feb. 9, 2023. arXiv: 2302.04675[cs]. URL: http://arxiv.org/abs/2302.04675 (visited on 12/05/2023).

[9] Wikipedia. *2017 Equifax data breach — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=2017%20Equifax%20data%20breach&oldid=1185471165. [Online; accessed 05-December-2023]. 2023.

[10] Yaqin Zhou et al. *Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks*. Sept. 8, 2019. arXiv: 1909.03496[cs,stat]. URL: http://arxiv.org/abs/1909.03496 (visited on 12/05/2023).

VI. Appendix

A. Prompt template

For inference, the statements after [/INST] are removed.

```
<s>
[INST]
<<SYS>>
You are a helpful assitant that
searches for vulnerabilites in code.
<</SYS>>
Is the follwing code vulnerable or not?
Answer with True or False:
static void _UTF7Reset (
  UConverter * cnv ,
  UConverterResetChoice choice ) {

  if (choice<=UCNV_RESET_TO_UNICODE){
    cnv->toUnicodeStatus =0x1000000;
    cnv->toULength = 0 ;
  }
  if (choice!=UCNV_RESET_TO_UNICODE){
    cnv->fromUnicodeStatus=(
    cnv->fromUnicodeStatus&0xf0000000
    )|0x1000000;
  }
}
[/INST]
True
</s>
```