LLM-Based Vulnerability Detection in High-Level Scripts and HDL Modules

Andre Nakkab, Mehul Sudrik
New York University
New York, NY 11201
`ajn313@nyu.edu, ms14029@nyu.edu`

Abstract

Detection of security vulnerabilities is a major necessity in the modern age of information. Many previously explored AI-based vulnerability detection methods require costly model training and do not have public implementations that could be used in a practical setting. Vulnerabilities in must be detected rapidly and accurately to avoid serious negative outcomes, so time wasted training a model could be dangerous. Unfortunately, existing vulnerability detection methods struggle with a large volume of False Positives predictions, which can muddy the waters and distract developers from fixing real issues. Additionally, very little investigation has been done into vulnerability detection for hardware description language (HDL), such as Verilog modules. Due to these factors, we posit a methodology by which existing, public-facing large language models (LLMs) can be utilized as competitively effective vulnerability detection tools. This could save dramatically on time and training costs, while also allowing for a greater degree of intra-model customizability to reduce false positive rates. We find that LLMs provide numerous benefits as vulnerability detection tools, but also come with their own limitations.

1 Introduction

Within the field of cybersecurity, there is a persistent metaphorical arms race between the exploitation and repair of security vulnerabilities within code. Cybersecurity engineers must be vigilant about preventing fragile, risky code from being released to the public, lest bad actors take advantage of this fragility to cause harm. [2]

In order to perform this task effectively, these vulnerabilities must first be recognized and detected. Doing this manually often means going over large sections of code with a fine toothed comb in order to see where mistakes may have been made which introduce risks into the code. This is a considerable time investment even for experienced engineers, so much so that it may stymie productivity at an organizational scale.

1.1 Related work

To address these concerns, numerous methodologies have been researched in order to detect vulnerabilities in an automated fashion, often utilizing deep learning tools. However, these methods tend to come with a very high false positive rate (FPR), meaning that they often report that code contains vulnerabilities when it does not. This can introduce a considerable amount of additional workload for development teams aiming to fix security risks, so the overall benefit of these tools is somewhat mitigated by this downside. [1]

Additionally, there have been studies exploring LLMs specifically for vulnerability detection in code. These studies have shown competitive results when compared to purpose-trained models, though they have largely reported similar issues wtih FPR, with most errors being false positives. However, these studies only focus on vulnerabilities in high-level code. [5]

1.2 Our contribution

We seek to address some of the issues with previous studies. Firstly, we seek to reproduce results showing that LLMs can be an effective vulnerability detection mechanism, while avoiding the time sink of creating a purpose-trained model. Secondly, we posit that by utilizing appropriate prompt engineering, we can achieve a degree of customizability in our outputs such that FPR is reduced. Thirdly, we seek to provide a method of detecting vulnerabilities at the hardware level, by testing this method on Verilog HDL modules in parallel with our testing of high-level code.

2 Methods

In both our explorations of high-level code and Verilog modules, we utilized GPT-3.5 and GPT-4 as our primary testbeds. Other LLMs like CodeLlama can still be effective at this task, but given our limited time, we decided to focus on the most powerful LLMs available to us, and the two most commonly used by the general public.

Due to the manual nature of our current techniques, we selected a subset of the datasets that we used for each category of code. For high-level code, a total of 100 samples of code were used, and for Verilog, a total of 233 samples were used.

We measure the standard binary classification metrics of true positive, where code which contains vulnerabilities is correctly reported by the model as vulnerable, true negative, where the model accurately reports that there are no vulnerabilities, false positive, where the model reports a vulnerability where there is none, and false negative, where the model misses a vulnerability and reports that fragile code is clean.

2.1 Datasets

For dataset, we delved into the Security Patches dataset, focusing on high-level code blocks, these blocks were classified into two distinct categories: fragile (vulnerable) and non-fragile (secure). Our primary data sources for this dataset were the National Vulnerability Database (NVD) and Open Source Vulnerabilities (OSV). The NVD and OSV repositories provided a comprehensive collection of vulnerable code segments spanning two decades, from 2002 to 2022. These repositories are notable for their extensive coverage of software vulnerabilities and are widely recognized for their reliability and depth. Each entry in these databases included a detailed record of the vulnerability, along with the corresponding patches. These patches, crucial for our analysis, were readily accessible through GitHub links. Our dataset was encapsulated in a CSV file, which served as a structured compilation of this information. This file included direct links to the relevant commits on GitHub, a description of each code segment, an indication of whether the code was vulnerable, and the specific type of vulnerability it possessed. This comprehensive dataset formed the backbone of our analysis, enabling us to conduct a thorough examination of the effectiveness of security patches in high-level code. [3]

Verilog code was sourced from the Verilog GitHub dataset, which does not internally distinguish between vulnerable and non-vulnerable code. Therefore, each module utilized in testing was manually examined for vulnerabilities before being sent to the LLM. Additionally, if the LLM noticed a vulnerability that the human rater missed, this was still counted as a true positive. [4]

2.2 High-Level Code

In this case study, we explored the effectiveness of "High-Level-Code" vulnerability detection using different linguistic prompts. Our methodology involved presenting three distinct prompts to assess if variations in phrasing influenced the detection capabilities. These prompts were:

1. Prompt-A: "Can you check if there is any code vulnerability in the following block of code?"
2. Prompt-B: "Can you check if the below code block has any security risks?"
3. Prompt-C: "Do you see any security risk in the following block of code?"

To conduct this analysis, we allocated 60 prompts for GPT-4 and 40 for GPT-3.5 adhering to the usage limitation of 40 prompts every three hours. Within these sets, for GPT-3.5, we used 23 vulnerable and 17 patched code blocks. Similarly, for GPT-4, we utilized 37 vulnerable and 23 patched code blocks.

Our approach to evaluating the results involved constructing a matrix to categorize responses as true positives, true negatives, false positives, and false negatives, separately for GPT-3 and GPT-4. When the AI was prompted, it provided detailed analyses of potential vulnerabilities in the given code blocks. If GPT's description accurately identified an existing vulnerability in a code block, we classified this as a True Positive. Conversely, if it failed to recognize an actual vulnerability, we marked it as a False Negative.

For the non-vulnerable, or patched, code blocks, our evaluation criteria were slightly different. If a code block had been patched for a specific vulnerability (say 'X'), and GPT still flagged 'X' as a potential issue, we considered this a False Positive. If GPT correctly identified that the code was secure from 'X', we recorded it as a True Negative. This systematic approach allowed us to assess the precision and reliability of GPT-3 and GPT-4 in identifying code vulnerabilities.

2.3 Hardware Description Language

When approaching vulnerability detection for Verliog modules, we focused on a single prompt which gave relatively neutral results in initial testing. Similarly to with the high-level code analysis, the master prompt would be fed to the LLM with an HDL module appended to the end of it. The prompts attempted were:

1. "Please check the following Verilog module for security risks:"
2. "Please report any definite, specific security vulnerabilities in the following Verilog module:"
3. "Does the following Verilog module contain any security vulnerabilities?:"

We settled on the first prompt as our master prompt, as it seemed to be the best at balancing between false positives and false negatives.

We then parse the output of the LLM to check whether it calls out a specific security vulnerability, which we would consider a positive report, or if it either gives general security advice with no specificity or specifically states there does not seem to be a vulnerability, which we would consider a negative report.

3 Results

For both high-level code and HDL, a relatively competitive degree of accuracy was achieved using our methods. As anticipated, we were able to customize the distribution of our outputs utilizing prompts that were more tuned toward providing false positives or false negatives. We can further analyze our results by code category.

3.1 High-Level

Our investigation into the effectiveness of GPT-3.5 and GPT-4 in identifying vulnerabilities in high-level code yielded insightful results. The analysis was structured around three distinct prompts, each designed to evaluate the models' ability to detect code vulnerabilities. The performance of each model was assessed based on four key metrics: True Positive, True Negative, False Positive, and False Negative rates, along with calculated values for Accuracy, Precision, and Recall.

For GPT-3.5, the results were as follows:

- Prompt-A: Showed a balanced performance with 17 True Positives and 12 True Negatives. However, it had 5 False Positives and 6 False Negatives.
- Prompt-B: Exhibited the best performance among the three prompts for GPT-3.5, with higher True Positives (18) and True Negatives (14), and lower False Positives (3) and False Negatives (5).
- Prompt-C: Mirrored the results of Prompt-A in True Positives and False Positives but had a slightly lower performance in True Negatives and False Negatives.

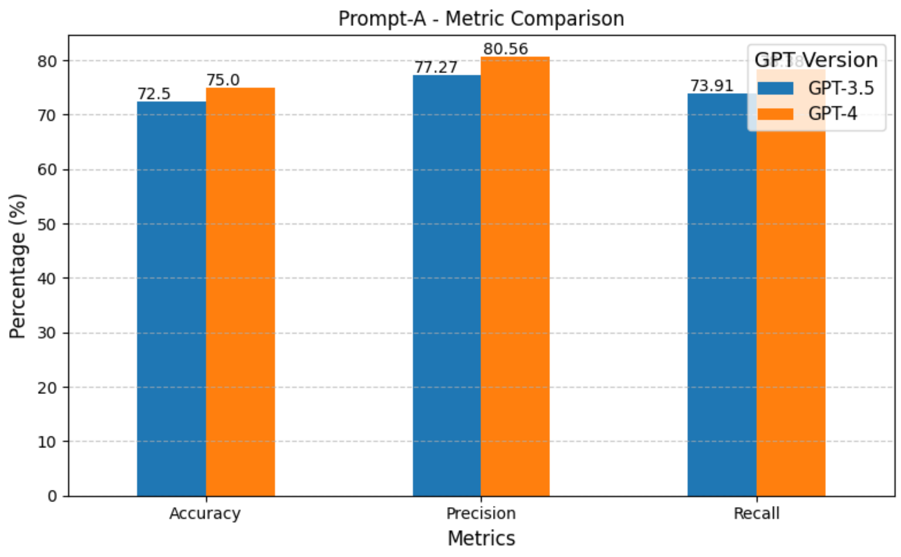


Figure 1: Prompt A metrics

The overall accuracy ranged from 72.5% to 80%, with Prompt-B leading. Precision and Recall metrics were also the highest for Prompt-B, indicating its effectiveness in correctly identifying vulnerabilities and minimizing false identifications.

For GPT-4, the results were as follows:

- Prompt-A: Demonstrated a strong ability to identify True Positives (29) and True Negatives (16) but also had a notable number of False Positives (7) and False Negatives (8).
- Prompt-B: This prompt showed the highest accuracy (81.67%) and precision (86.11%) across all tests. It had the highest True Positives (31) and True Negatives (18), with relatively fewer False Positives (5) and False Negatives (6).
- Prompt-C: While it had a high number of True Positives (28), it also showed a significant number of False Positives (9) and False Negatives (9), leading to the lowest accuracy (70%) among the GPT-4 prompts.

Comparative Analysis:

When comparing GPT-3.5 and GPT-4, it is evident that GPT-4 generally outperforms GPT-3.5 in terms of Accuracy, Precision, and Recall across all prompts. This suggests that the advancements in GPT-4 have led to improved capabilities in identifying code vulnerabilities. However, the presence of False Positives and False Negatives in both models highlights the ongoing challenge in achieving perfect accuracy in automated vulnerability detection.

The variation in results across different prompts also underscores the importance of prompt design in eliciting accurate responses from AI models. Prompt-B consistently showed better results in both GPT versions, suggesting that its phrasing was more effective in guiding the AI to analyze code vulnerabilities accurately.

In conclusion, while both GPT-3.5 and GPT-4 demonstrate significant potential in identifying vulnerabilities in high-level code, the choice of prompts and the inherent limitations of AI in understanding context and nuance in code analysis are critical factors that influence their performance. These findings open avenues for further research into optimizing prompt design and improving AI models for more accurate vulnerability detection in software development.

We include comparison graphs for the metrics on each of our prompts.

Table 1: GPT-3.5 and GPT-4 Performance Metrics

	Prompt-A	**Prompt-B**	**Prompt-C**
GPT - 3.5			
True Positive	17	18	18
True Negative	12	14	12
False Positive	5	3	5
False Negative	6	5	5
Total (23 Vulnerable, 17 Patched)	40	40	40
Accuracy	72.5	80	75
Precision	77.27	85.71	78.26
Recall	73.91	78.26	78.26
GPT - 4			
True Positive	29	31	28
True Negative	16	18	14
False Positive	7	5	9
False Negative	8	6	9
Total (37 Vulnerable, 23 Patched)	60	60	60
Accuracy	75	81.67	70
Precision	80.56	86.11	75.67
Recall	78.38	83.78	75.67

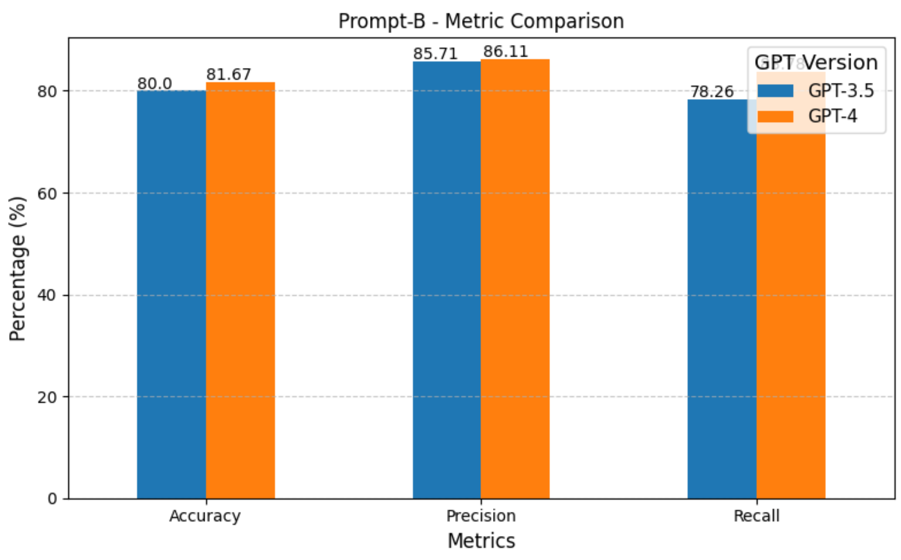


Figure 2: Prompt B metrics

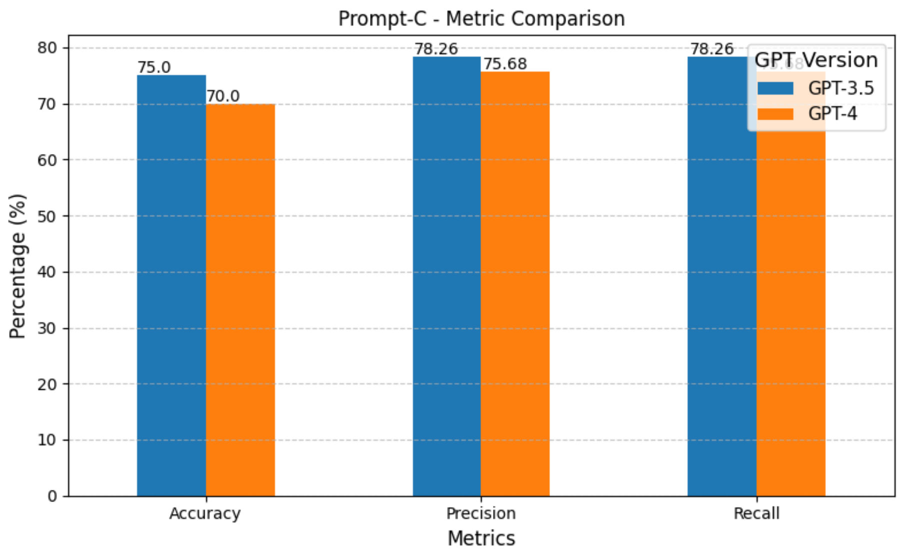


Figure 3: Prompt C metrics

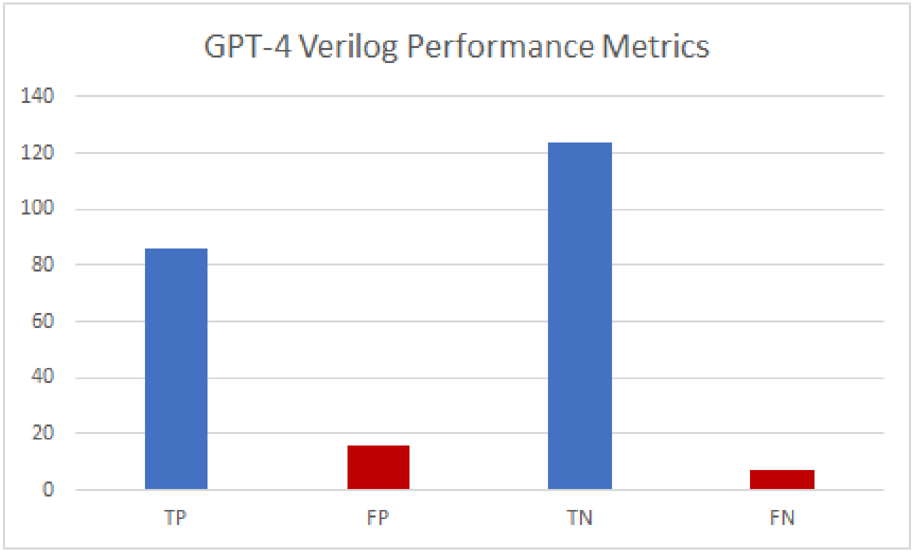


Figure 4: True/False metrics for Verilog

3.2 HDL

When testing this method on HDL code, we found rather compellingly that we were able to achieve around 90.13% accuracy, with a precision of 84.31 and a recall of 92.47. We see a reporting breakdown as follows:

1. True Positives: 86
2. False Positives: 16
3. True Negatives: 124
4. False Negatives: 7

We additionally see that GPT-4 reports various potential pathways to addressing security risks. While this may be obfuscatory to an inexperienced engineer, it can act as a force multiplier for experienced engineers. Generally, it is relatively easy to differentiate between what is considered a positive report

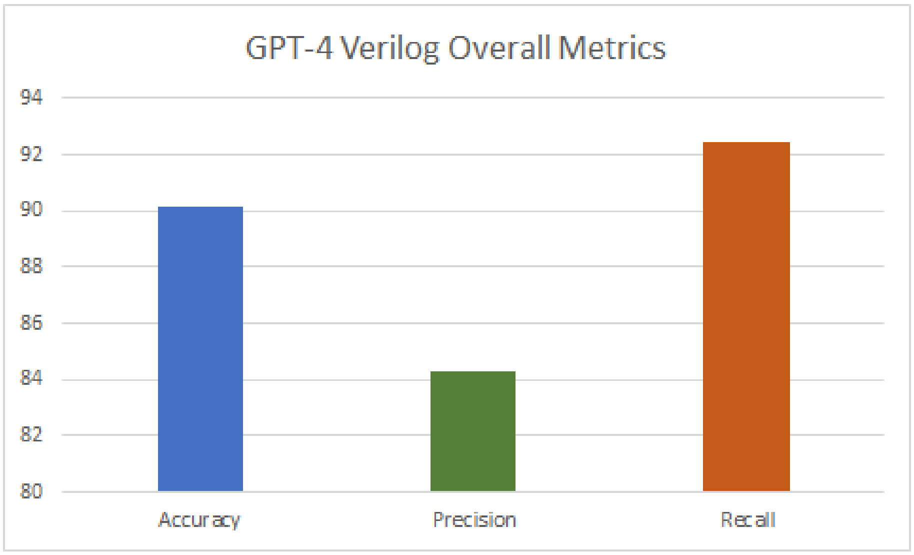
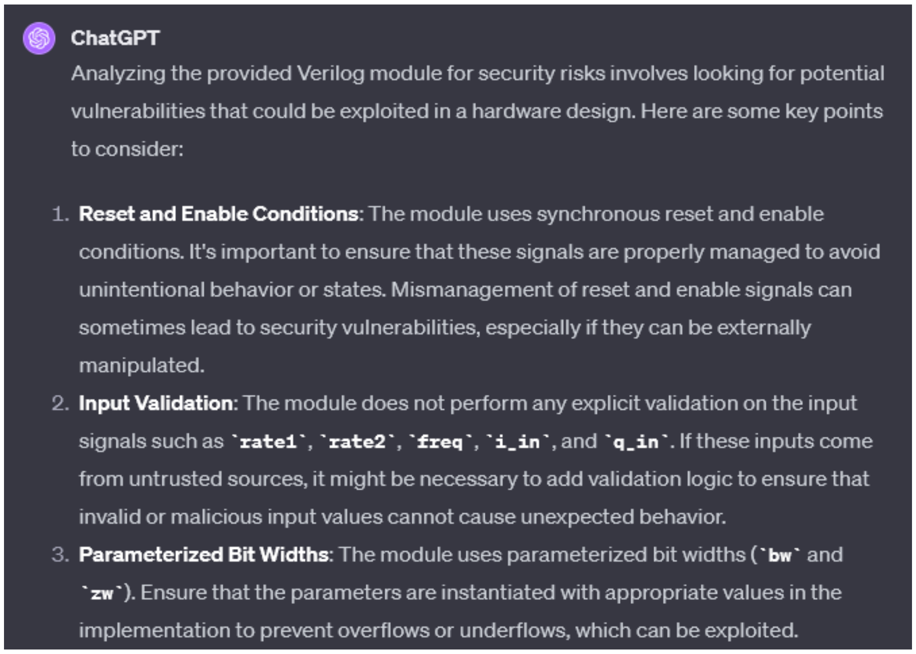


Figure 5: Overall performance metrics for Verilog



ChatGPT

Analyzing the provided Verilog module for security risks involves looking for potential vulnerabilities that could be exploited in a hardware design. Here are some key points to consider:

1. **Reset and Enable Conditions**: The module uses synchronous reset and enable conditions. It's important to ensure that these signals are properly managed to avoid unintentional behavior or states. Mismanagement of reset and enable signals can sometimes lead to security vulnerabilities, especially if they can be externally manipulated.
2. **Input Validation**: The module does not perform any explicit validation on the input signals such as `rate1`, `rate2`, `freq`, `i_in`, and `q_in`. If these inputs come from untrusted sources, it might be necessary to add validation logic to ensure that invalid or malicious input values cannot cause unexpected behavior.
3. **Parameterized Bit Widths**: The module uses parameterized bit widths (`bw` and `zw`). Ensure that the parameters are instantiated with appropriate values in the implementation to prevent overflows or underflows, which can be exploited.

Figure 6: Sample of GPT-4 output

or a negative report. Beyond that, it may provide avenues of approach to detect and address security concerns that would not have been thought of otherwise.

4 Conclusion

We posit that LLMs represent an important force multiplier for cybersecurity engineers. When we consider the advantages of LLM usage over traditional deep learning models, it becomes clear that the expediency and efficacy on display is difficult to ignore.

If these tools were utilized in this fashion on a large scale, it would likely lead to a large scale reduction in the number of code vulnerabilities being exploited by bad actors across the Internet. The financial savings of mass adoption would also be considerable, as it would prevent costly security breaches & the associated loss of income, as well as save cybersecurity engineers a considerable amount of time which could be spent on tasks which require more human feedback.

4.1 Future Work

Going forward, we would like to explore the efficacy of other LLMs such as CodeLlama and Bard, and do a much more in depth comparative statistical analysis on cross-model performance.

Additionally, we would like to see if we could create a parser which refines the LLM's feedback into something more easily and directly usable by an engineer.

Finally, we would like to train or fine-tune our own LLM with the intent of more accurately diagnosing vulnerabilities, as well as generatively providing fixes on the fly.

5 Supplementary Material

We provide a GitHub repository which contains all the necessary information for our LLM-based vulnerability detection study. Included are: results spreadsheets, links to datasets, share links to sample LLM conversations, and the report itself.

The repository can be found at the following URL:

https://github.com/ajn313/llm_vulnerability_detection

References

[1] A. G. Ayar, A. Sahruri, S. Aygun, M. S. Moghadam, M. H. Najafi and M. Margala, "Detecting Vulnerability in Hardware Description Languages: Opcode Language Processing," in IEEE Embedded Systems Letters, doi: 10.1109/LES.2023.3334728.

[2] G. Bhandari, A. Naseer, and L. Moonen, "CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software," in Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '21),doi: /10.1145/3475960.3475985.

[3] S. Reis and R. Abreu, "A ground-truth dataset of real security patches," arXiv preprint arXiv:2110.09635, 2021. Available: https://arxiv.org/abs/2110.09635

[4] S. Thakur, B. Ahmad, Z. Fan, H. Pearce, B. Tan, R. Karri, B. Dolan-Gavitt, and S. Garg, "Benchmarking Large Language Models for Automated Verilog RTL Code Generation," arXiv preprint arXiv:2212.11140, 2022. Available: https://arxiv.org/abs/2212.11140

[5] S. Lingarkar, "Detecting Code Vulnerabilities in Large Language Models (LLMs)", LinkedIn, [Online]. Available: https://www.linkedin.com/pulse/detecting-code-vulnerabilities-large-language-models-llms-lingarkar/. [Accessed: 12/20/2023].