Advanced Research on Information Systems Security

ISSN: 2795-4609 | ISSN: 2795-4560
Print & Online

# Assessing Domain Specific LLMs for CWEs Detections

Mohamed El Atoubi[a]*, Xiao Tan[b]

[a,b]*Georgia Institute of Technology, Atlanta, GA 30332, USA*

[a]*Email: matoubi3@gatech.edu*

[b]*Email: xtan70@gatech.edu*

**Abstract**

In recent years, Large Language Models (LLMs) have witnessed a significant evolution branching to several fields of life. From science & engineering to arts & literature, the realm of applications has become limitless. Their ability to assimilate and comprehend contextual writings is astonishing. This ability also extends to human-machine software written code. Hence, many novel attempts have demonstrated cutting-edge experiments with LLMs for software testing and security. These contributions have set the initial seed for promising future research endeavors to use LLMs to detect weaknesses, vulnerabilities, and malicious pieces of software code in even the largest repositories. However, further explorations remain short, especially with domain-specific LLMs. LLMs specifically trained for software security remain undiscovered and their behavior is still undisclosed in the literature. This paper aims to explore this new area of LLMs for software security through testing and comparing the accuracy of these AI models against general domain trained models and discover their abilities to recognize the exact vulnerability while performing and observational study of their behaviors while responding to the precisely crafted prompts. In our experiments, we considered GPT-3.5 from OpenAI and Gemini Pro from Google. We find that, in terms of recall, Gemini Pro outperformed GPT-3.5 by a large margin with recall of 63.13%, while GPT-3.5 has Recall of 43.56%, showing that Gemini Pro is better at identifying the true CWE vulnerability with less type II error. Meanwhile, Gemini Pro is also better at discovering the correct CWE vulnerability No. among all correct identified vulnerable cases, with the accuracy of 13.13% vs the GPT-3.5's 10.61%. However, GPT-3.5 is superior to Gemini Pro in terms of Precision and Accuracy. The Precision of GPT-3.5 is 88.89%, while Gemini Pro has a precision of 54.35%, showing that Gemini Pro inclines to identify case having vulnerability. The Accuracy for both models is similar; GPT-3.5 has Accuracy of 68.75%, and Gemini Pro has Accuracy of 55.50%.

***Keywords:*** LLM; CWE; Software Security; Vulnerabilities; Cybersecurity; AI

***Citation:*** M. Elatoubi and X. Tan, "Assessing Domain Specific LLMs for CWEs Detections", ARIS2-Journal, vol. 5, no. 1, pp. 72–85, May 2025.

***DOI***: https://doi.org/10.56394/aris2.v5i1.53

---------------------------------------------------------------------

*Corresponding author. Email address: matoubi3@gatech.edu

## 1. Introduction

Software vulnerabilities remain a persistent challenge in modern software systems, leading to critical issues such as system failures and buffer overflows. To address this challenge, numerous automated tools have been developed, including VUDDY [1], which focuses on discovering vulnerabilities in cloned code, and VulPecker [2], which employs code similarity algorithms to detect specific vulnerabilities.

The advent of Large Language Models (LLMs) has revolutionized various domains, including software engineering. Their remarkable capabilities across a wide range of tasks have prompted researchers to explore their potential in addressing complex software security challenges. In light of this, our research aims to investigate the effectiveness of two prominent LLM models—GPT-3.5 from OpenAI and Gemini Pro from Google—in detecting software vulnerabilities.

Our study focuses on two primary research questions:

**RQ1: Can LLMs effectively flag code containing vulnerabilities?**

This question aims to assess the ability of GPT-3.5 and Gemini Pro to identify potentially vulnerable code segments. We will evaluate their performance in distinguishing between secure and vulnerable code across various programming languages and vulnerability types.

**RQ2: How accurately can LLMs identify true Common Weakness Enumeration (CWE) categories?**

This question delves deeper into the LLMs' capability to not only detect vulnerabilities but also correctly classify them according to the standardized CWE framework. We will assess the models' precision in assigning appropriate CWE categories to identified vulnerabilities.

To address these research questions, we will employ a comprehensive methodology that leverages recent advancements in LLM-based vulnerability detection techniques. Our approach will include:

1. **Dataset Preparation**: A carefully curated collection of code samples from the DiverseVul dataset, encompassing both vulnerable and non-vulnerable instances across various programming languages and vulnerability types.
2. **Prompt Engineering**: Advanced prompting strategies optimized for vulnerability detection, including chain-of-thought reasoning.
3. **Performance Evaluation**: Rigorous assessment using standard metrics such as precision, recall, accuracy, and F1-score.
4. **Behavioral Analysis**: Observational study to analyze the unique behaviors exhibited by each model when responding to crafted prompts.

This research contributes to the growing body of knowledge on AI-driven software security by providing empirical evidence of LLMs' capabilities in vulnerability detection and offering insights into their practical application in software security workflows.

## 2. Background & Related Work

In the quest to leverage LLMs' high potential for contextual understanding and especially for software code, many research studies have explored the applications of generative AI in cybersecurity and more specifically for software quality assurance and security. The response speed and software static analysis techniques implemented by these models have encouraged researchers to publish their results and set a new horizon for promising future results. One of the first published studies was completed by the DiverseVul team from Yizheng et al. [3], they built a dataset from repositories containing vulnerable and non-vulnerable code. They formatted the code snippets in correct JSON format for automated data processing operations. They tested the dataset on trending LLMs at the time of study such as GPT-2 and RoBERTa which is an extension of BERT language model. The results were above 90% in accuracy but with low precision across all model architectures. Nevertheless, these results were encouraging. Rasmus et al. [4] in a recent study explored a new methodology where they tested on different datasets and with four research questions and different prompt designs like the "zero-shot" and the "chain of thought". They used up-to-date LLMs like GPT-3.5, Falcon-7b-instruct, Dolly-v2-12b, Text-davinci-003, and Llama-2-13b-chat-hf. The results varied in terms of accuracy ranging from 9.7% to 87%. The study didn't include statistics about precision but calculated the F1 measure. Although these experiments shed light on the matter and confessed the limitations of general LLMs applications in cybersecurity in general and software security in particular, they didn't experiment on domain-specific trained LLMs, especially LLMs trained for cybersecurity with threat intelligence capabilities. Our study aims to add to the current literature by exploring domain-specific LLMs trained specifically for cybersecurity and threat intelligence. The only publicly available LLM at the time of conducting this study is Google's Gemini Pro 1.5 which handles general tasks but has a part specifically trained for security analysis which is the model PaLM-Sec-2 and this model was supposed to be made available in a standalone version in the year of 2023 but was removed and introduced as part of the new Gemini Pro 1.5 in 2024.

## 3. Methodology

### *3.1. Experimentation process*

In this section, we elaborate on our proposed method for leveraging LLMs contextual understanding, static analysis abilities, and threat intelligence knowledge to maximize the accuracy and precision of vulnerability detection in software code. We commence by outlining the following experimentation process designed to accommodate both Generic and Domain-specific LLMs intricacies without any compromises. Our process addresses a couple of shortcomings found in previous research which is behavior analysis of LLMs over large datasets and the absence of empirical results on security trained LLM. An overview of the proposed method is illustrated in Figure 1.

We now explain the experimentation steps in detail:

1. We begin by exploring the dataset containing code functions using IDE software.
2. We extract a random sample of code functions that are statistically conforming, more details can be found in the dataset section.
3. As the process is manual, we design a spreadsheet to note results and calculate performance metrics.
4. We design the prompt template and combine it with the code snippet.
5. We insert the combined final prompt in the LLM input console and wait for the results.
6. We note down results about vulnerability detection and CWE precision if the code is vulnerable.
7. In case of out-of-context responses for both vulnerability detection and CWE precision, we use the Feedback control technique mentioned in section 3.3 until we get in-context results.

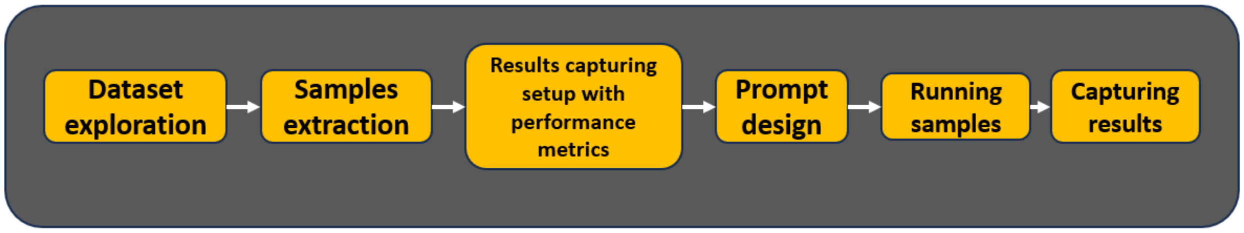


**Figure 1**: Method illustration.

### 3.2. Prompt design

Prompting is the art of knowing how to formulate request queries in order to receive the right output as expected. In recent years after the widespread use of LLMs and studying the responses that sometimes can be wrong, not aligned with context, or sometimes even completely irrelevant. Prompt design and engineering became of ultimate importance and can be seen as the grand steering wheel for directing LLMs artificial reasoning. Many studies have erupted in the field of prompt engineering. The results of these studies introduced different strategies and techniques to approach different problems by reformulating the prompts and empowering certain parts of LLMs artificial processing to maximize the alignment with expected and desired outputs. We will not discuss each prompting strategy and technique in this paper as it is outside of the scope of our study. For our experiment, we opted for the Chain-of-Thought Prompting. This technique enables and empowers the analytical side of LLMs. As the name suggests, the Chain-of-Thought operates by breaking a large problem into smaller parts called thoughts, these thoughts are linked to each other, thus forming a chain. This can be thought of the same way as human logic approaches complex problems. Jason et al. [5] have concluded in their research that the technique of Chain-of-Thought yields better results in LLMs that have sufficiently a large number of parameters. The template for the prompt is shown in Figure 2. Usually, LLMs have a limited number of tokens that can be used in terms of the size of the prompt. In our case this was not an issue as the prompt template was optimized and the code functions were not as large to be of concern. In addition, our main target LLM of study had approximately an unlimited number of tokens. We now explain in detail the steps taken in designing the prompt for software security analysis. We divided the task of the software security analyst into six subtasks as the following:

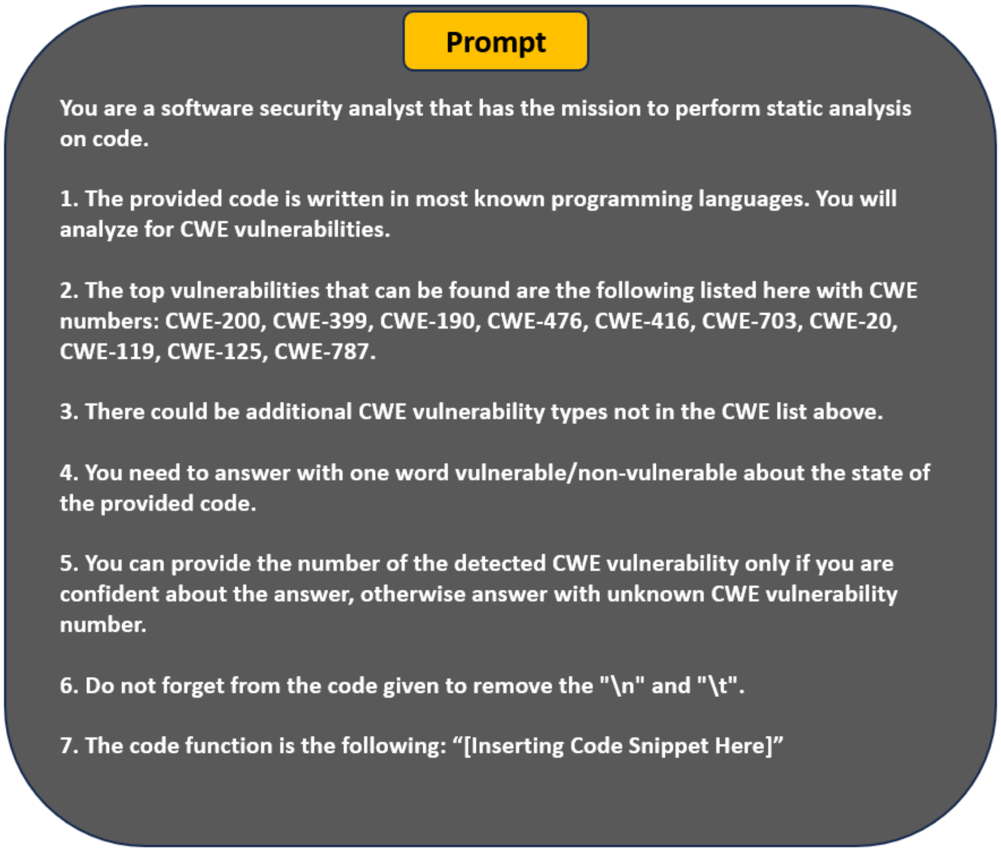
1. The first subtask is to put the LLM in the role of the software security analyst by alerting the presence of vulnerabilities in the provided code functions and stipulating the order to detect any vulnerabilities according to CWE list.

2. The second subtask is to limit the list of potential encountered CWEs after studying the sampled data. In real application scenarios this could be a guess about the top encountered CWEs.

3. To ensure fairness and avoid bias, we added the third subtask to inform the LLM that it might encounter CWE vulnerabilities not mentioned in subtask 2 and should analyze normally.

4. The fourth subtask is limiting the LLM output to choose between two answers whether the provided code is vulnerable or non-vulnerable.
5. The fifth subtask is a conditional one, meaning if the code provided is found to be vulnerable, then, the LLM must provide the CWE number only if it is confident about the type of the CWE.

6. The final subtask is to properly format the code removing space and new line annotations like “\t” and “\n”.



**Prompt**

You are a software security analyst that has the mission to perform static analysis on code.

1. The provided code is written in most known programming languages. You will analyze for CWE vulnerabilities.

2. The top vulnerabilities that can be found are the following listed here with CWE numbers: CWE-200, CWE-399, CWE-190, CWE-476, CWE-416, CWE-703, CWE-20, CWE-119, CWE-125, CWE-787.

3. There could be additional CWE vulnerability types not in the CWE list above.

4. You need to answer with one word vulnerable/non-vulnerable about the state of the provided code.

5. You can provide the number of the detected CWE vulnerability only if you are confident about the answer, otherwise answer with unknown CWE vulnerability number.

6. Do not forget from the code given to remove the "\n" and "\t".

7. The code function is the following: "[Inserting Code Snippet Here]"

**Figure 2**: Prompt used.

### *3.3. Feedback control*

On a few occasions, LLMs' way of responding can change drastically even with a well-designed prompt. This issue is usually faced when using public LLMs due to their openness to learn from prompts input from public users. In the field of AI, this phenomenon is called model drifting. To help correct LLM response behavior and keep it well aligned with the stipulated requirements in the prompt. We thought of using a simple Feedback control method. This would a novel technique to use for LLMs when thought of as closed-loop systems. The Feedback control technique is to wait for the LLM output and check if it provides an output conforming and in-context to the prompt's requirements. If not then manually, input a small prompt stating to the LLM to stick to the prompt and provide an answer. The revised experimentation process in Figure 3 shows how feedback control is implemented. We now give an example of the model drift control prompt if the LLM output is far from what is expected.

Model drift control prompt: "This is an out-of-context response, your answer must only be 'vulnerable' or 'non-vulnerable' code".

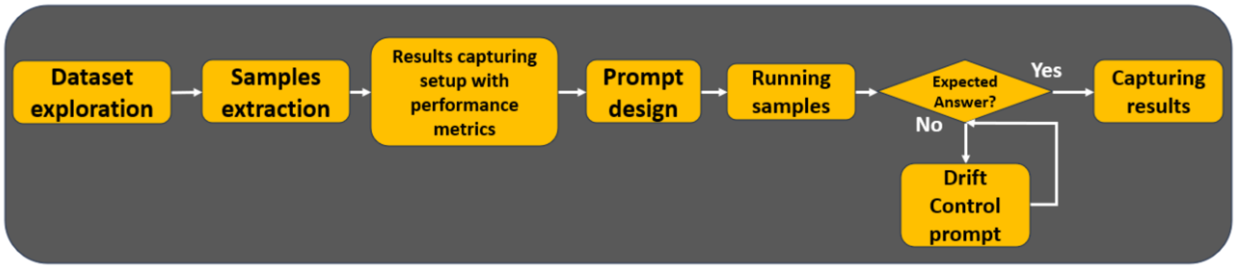


**Figure 3**: Experimentation process with feedback control.

Talk about this technique that we implemented as a novel technique not used before and explain that it was only implemented when the LLM goes out of context.

## 4. Dataset

### *4.1. Data Source*

We will describe the dataset used for this study. We chose to experiment on the most recent and diversified dataset containing vulnerabilities which is DiverseVul. DiverseVul is a collection of public repository code commits with high-quality security issues that are clear and labeled under the Common Weakness Enumeration "CWE" which is a standard category system for hardware and software weaknesses and vulnerabilities [6]. The choice also comes from the fact that the dataset was well organized and labeled correctly in JSON format. The programming language is basically C/C++, which is quite familiar to LLMs, and this was also a great motivation for us. The other criteria were the diversity of the dataset as the code commits were collected from various open-source projects covering several fields such as operating systems, networking protocols, machine learning, security, and Web technology. The dataset contained exactly 7,514 commits from 797 projects, which resulted in 18,945 vulnerable code functions and 330,492 non-vulnerable ones, spanning over 150 CWEs as stated by the DiverseVul team itself [3]. The dataset was made available for the public to use and experiment on by the DiverseVul team. We downloaded the dataset in JSON format file from the Google Drive storage link provided. The file size is around 720 MB, and we used an Intelligent Development Environment software "IDE" to open and explore the dataset.

### *4.2. Sampling*

The dataset itself was easy to explore and well formatted for automated operations such as sending the code commits to the LLMs and receiving feedback via the provided API but due to the objectives of our study explained earlier, we had to run the experiments manually. Due to these constraints, it was impossible for us to run the experiment manually over the whole code commits, hence, the only solution was to sample at random and with an acceptable statistical confidence level. With a confidence level of 95%, a margin of error of 5%, a population proportion of 50% since each data sample can either be vulnerable or non-vulnerable in reality, and a population size of 349,427 code commits, following the sampling equation we need a sample size of 384 samples or more. We decided to fix the sample size at 400 samples to be in a relatively superior precision. We sampled 201 samples for vulnerable and 199 non-vulnerable code functions, each making 400 samples in total.

| Parameters | Confidence level = 95% | Margin of error = 5% | Population proportion = 50% |
|---|---|---|---|
| Population size = 349,427 | Calculated sample size = 384<br>Total samples taken = 400 | Vulnerable samples = 201 | Non vulnerable samples = 199 |

**Table 1:** Sampling results.

## *4.3. Sampling Results*

In this section, we give the results from our sampling. We first begin by listing the distribution of samples across the vulnerable code functions and the found CWEs. The following table lists the results:

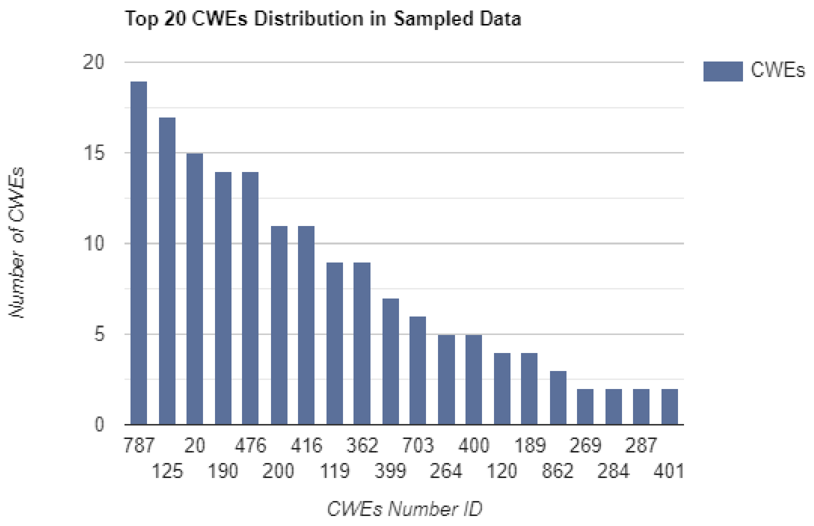


**Figure 4:** Distribution of CWEs across vulnerable code functions.

Listing the CWEs numbers only is not as informative as listing its description, hence, we list the following table containing the CWEs listed in Figure 4 with a description title of the CWE.

| CWE Number | CWE Title |
|---|---|
| 787 | Out-of-bounds Write |
| 125 | Out-of-bounds Read |
| 20 | Improper Input Validation |
| 190 | Integer Overflow or Wraparound |
| 476 | NULL Pointer Dereference |
| 200 | Exposure of Sensitive Information to an Unauthorized Actor |
| 416 | Use After Free |
| 119 | Improper Restriction of Operations within the Bounds of a Memory Buffer |
| 362 | Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition') |
| 399 | Resource Management Errors |
| 703 | Improper Check or Handling of Exceptional Conditions |
| 264 | Permissions, Privileges, and Access Controls |
| 400 | Uncontrolled Resource Consumption |
| 120 | Buffer Copy without Checking Size of Input ('Classic Buffer Overflow') |
| 189 | Numeric Errors |
| 862 | Missing Authorization |
| 269 | Improper Privilege Management |
| 284 | Improper Access Control |
| 287 | Improper Authentication |
| 401 | Missing Release of Memory after Effective Lifetime |

**Table 2:** CWEs found in the sample data and their title.

## 5. Evaluations

To assess the detection accuracy, we conducted an experiment using Google Gemini Pro and Open AI GPT-3.5. We analyzed the responses from both agents for each case in the dataset using the same prompt.

### 5.1 *Summary of Results*

In analyzing the response to compare the accuracy, we defined a response as vulnerable if the response from agents directly told us the case was vulnerable following the prompt requirements or, in rare cases, the agents responded with a detailed explanation from which vulnerability could be inferred. Conversely, a response was labeled as non-vulnerable if the response directly stated the case was vulnerable or non-vulnerability could be inferred from the detailed response with an explanation.

In the same way, the CWE number was recorded if the response stated the case in the dataset was vulnerable and the response contained a CWE number. In some cases, the response which stated the case was vulnerable did not include any CWE number. We defined the case in the dataset had the exact CWE number if the response from models stated the case was vulnerable and output a CWE number which was the same as the true CWE number of this case.

Table 3 and Table 4 present the vulnerability detection results of Google Gemini Pro and Open AI GPT-3.5. Table 5 shows the result of the performance metrics comparison. Precision ($\frac{TP}{TP+FP}$), Recall ($\frac{TP}{TP+FN}$), Accuracy ($\frac{TP+TN}{TP+TN+FP+FN}$), and F-measure ($2 \times \frac{Precision \times Recall}{Precision+Recall}$) were used as performance metrics. Gemini Pro demonstrated superior performance compared to GPT-3.5 in terms of true positives, correctly identifying 125 vulnerable cases compared to GPT-3.5's 88. However, Gemini Pro also produced a significantly higher number of false positives (105) compared to GPT-3.5 (11) and a moderately smaller number of false negatives (73) compared to GPT-3.5 (114), indicating Gemini Pro is much more conservative in diagnosing non-vulnerable cases.

Out of 198 cases with a CWE No. in the selected database, the GPT-3.5 correctly predicted 26 cases with exact CWE No. with an accuracy of 13.13%, while the Gemini Pro correctly predicted 21 cases with an accuracy of 10.61%. Both models performed unsatisfactorily in identifying the CWE No.

| | | **Predicted** | |
|---|---|---|---|
| | | **Vulnerable** | **Non-Vulnerable** |
| **Actual** | **Vulnerable** | TP | FN |
| | | 88 | 114 |
| | **Non-Vulnerable** | FP | TN |
| | | 11 | 187 |

**Table 3:** Confusion Matrix for GPT-3.5

| | | **Predicted** | |
|---|---|---|---|
| | | **Vulnerable** | **Non-Vulnerable** |
| **Actual** | **Vulnerable** | TP | FN |
| | | 125 | 73 |
| | **Non-Vulnerable** | FP | TN |
| | | 105 | 97 |

**Table 4:** Confusion Matrix for Google Gemini Pro

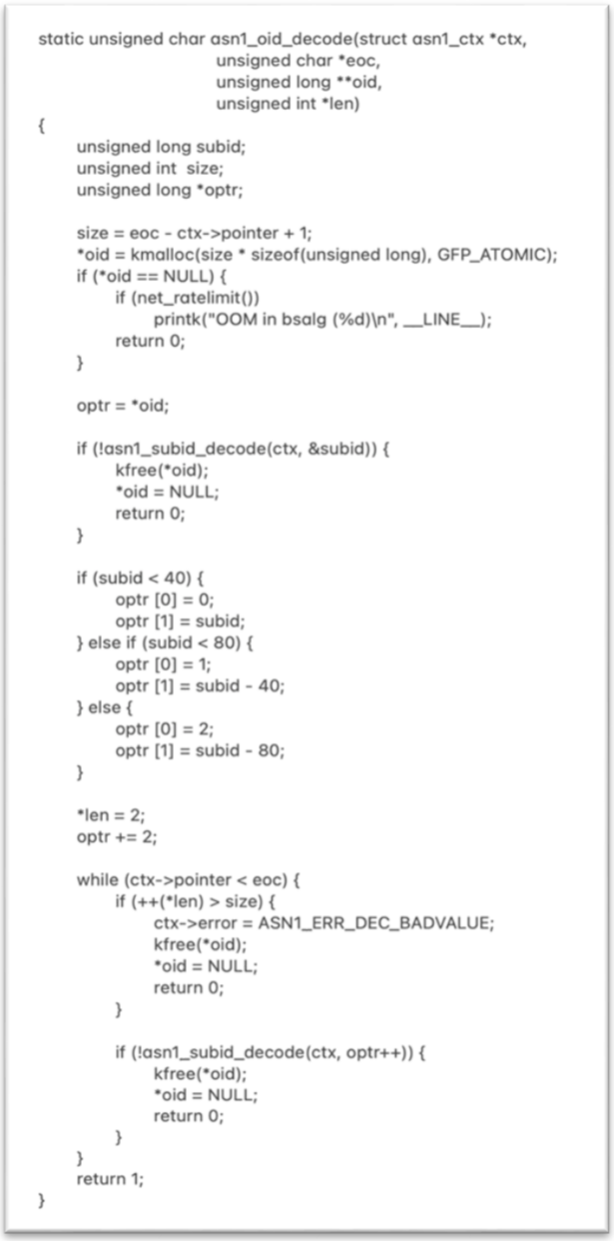
| | **GPT-3.5** | **Google Gemini Pro** |
|---|---|---|
| **Precision** | 88.89% | 54.35% |
| **Recall** | 43.56 % | 63.13% |
| **Accuracy** | 68.75% | 55.50% |
| **F-measure** | 58.47% | 58.41% |

**Table 5:** Performance Metrics Comparison

### 5.2 *Examples of Responses*

In this section, we will discuss specific examples of GPT-3.5 and Gemini Pro's responses when it correctly detected the vulnerabilities and output the required response.

Figure 5 shows a code example with vulnerabilities (CWE-119).



```
static unsigned char asn1_oid_decode(struct asn1_ctx *ctx,
                            unsigned char *eoc,
                            unsigned long **oid,
                            unsigned int *len)
{
      unsigned long subid;
      unsigned int  size;
      unsigned long *optr;

      size = eoc - ctx->pointer + 1;
      *oid = kmalloc(size * sizeof(unsigned long), GFP_ATOMIC);
      if (*oid == NULL) {
            if (net_ratelimit())
                  printk("OOM in bsalg (%d)\n", __LINE__);
            return 0;
      }

      optr = *oid;

      if (!asn1_subid_decode(ctx, &subid)) {
            kfree(*oid);
            *oid = NULL;
            return 0;
      }

      if (subid < 40) {
            optr [0] = 0;
            optr [1] = subid;
      } else if (subid < 80) {
            optr [0] = 1;
            optr [1] = subid - 40;
      } else {
            optr [0] = 2;
            optr [1] = subid - 80;
      }

      *len = 2;
      optr += 2;

      while (ctx->pointer < eoc) {
            if (++(*len) > size) {
                  ctx->error = ASN1_ERR_DEC_BADVALUE;
                  kfree(*oid);
                  *oid = NULL;
                  return 0;
            }

            if (!asn1_subid_decode(ctx, optr++)) {
                  kfree(*oid);
                  *oid = NULL;
                  return 0;
            }
      }
      return 1;
}
```

**Figure 5:** Example Code with Vulnerabilities (CWE-119)

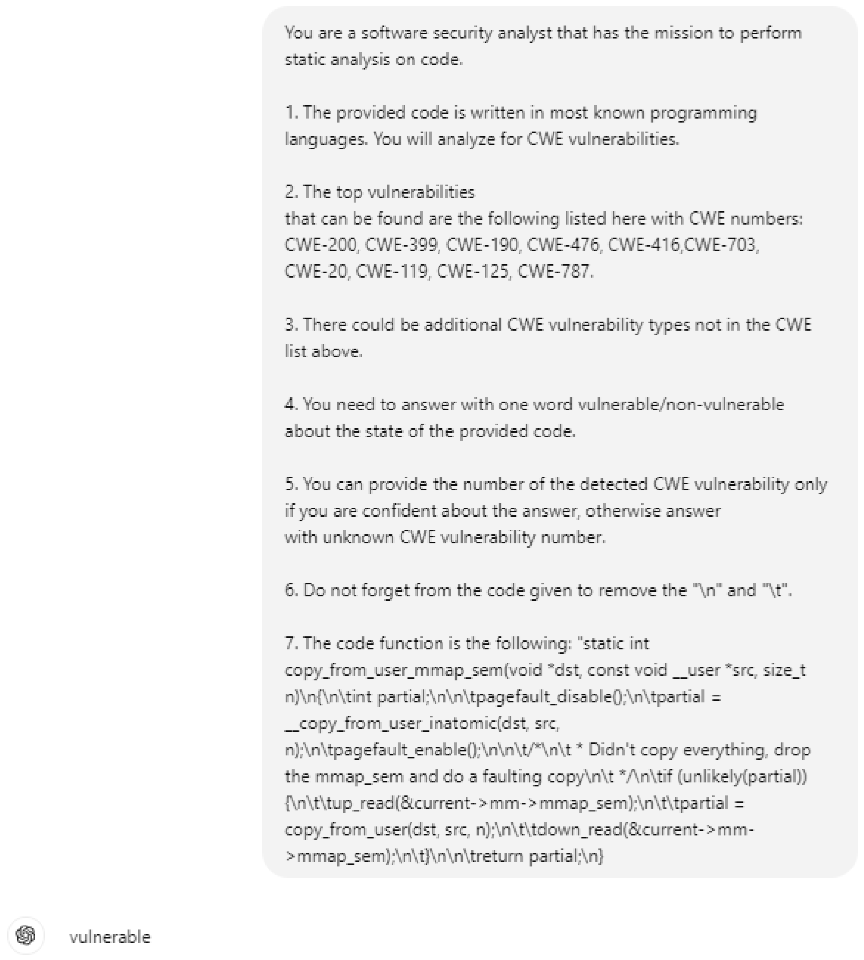
- **Gemini Pro Response:**

The Gemini Pro's response aligns with expected format by outputting whether the vulnerability exists, if existing, then output CWE No. The Gemini Pro correctly identified the code containing vulnerabilities but wrongly identified CWE No. to be CWE-787.

It is worth noting that Gemini Pro sometimes outputs three draft outputs, indicating CWE No., when identifying the code has vulnerability. Sometimes the three-draft output CWEs have different numbers, showing even though Gemini Pro is not sure about the CWE No., it still provides three options for users to investigate it manually, which contradicts GPT-3.5.

- **GPT Response:**

GPT responses were not always aligned with prompt instructions even if the code is vulnerable. The output would only state that code snippet is vulnerable without supplying the type of weakness in the or CWE number. GPT required further prompting to elaborate more on the type of software weakness. As for the determination process, GPT did a recommendable job in judging whether the piece of software was vulnerable or not. In other cases, GPT performed poorly even with simple CWE cases. The following figure represents a response example from GPT.



You are a software security analyst that has the mission to perform static analysis on code.

1. The provided code is written in most known programming languages. You will analyze for CWE vulnerabilities.

2. The top vulnerabilities
that can be found are the following listed here with CWE numbers: CWE-200, CWE-399, CWE-190, CWE-476, CWE-416,CWE-703, CWE-20, CWE-119, CWE-125, CWE-787.

3. There could be additional CWE vulnerability types not in the CWE list above.

4. You need to answer with one word vulnerable/non-vulnerable about the state of the provided code.

5. You can provide the number of the detected CWE vulnerability only if you are confident about the answer, otherwise answer
with unknown CWE vulnerability number.

6. Do not forget from the code given to remove the "\n" and "\t".

7. The code function is the following: "static int copy_from_user_mmap_sem(void *dst, const void __user *src, size_t n)\n{\n\tint partial;\n\n\tpagefault_disable();\n\tpartial = __copy_from_user_inatomic(dst, src, n);\n\tpagefault_enable();\n\n\t/*\n\t * Didn't copy everything, drop the mmap_sem and do a faulting copy\n\t */\n\tif (unlikely(partial)) {\n\t\tup_read(&current->mm->mmap_sem);\n\t\tpartial = copy_from_user(dst, src, n);\n\t\tdown_read(&current->mm->mmap_sem);\n\t}\n\n\treturn partial;\n}

vulnerable

**Figure 6:** Example of GPT response to a vulnerable code snippet.

### *5.3 Performance Metrics Analysis*

From Table 5, the performance metrics between the two models were quite different than initial expectations. The precision metric measures how correctly the model can identify true positives from positive class [8]. The GPT performance metrics results for precision and accuracy were far superior to Gemini Pro especially when it comes to precision as GPT achieves 88.89% against 54.5% for Gemini Pro, but this superiority decreases for accuracy metric where GPT scores 68.75% compared to 55.5% for Gemini Pro. This can be due to the seniority of GPT model and open learning policy from public prompting by users worldwide. For the recall performance metric, Gemini Pro was significantly better than GPT model at 63.13% against 43.56% respectively. The recall performance metric measures the model's ability to detect true positives from all positive samples [8]. This can be explained by the fact that a segment of Gemini Pro is fully trained by Google on threat intelligence data.

## 6. Observations & Limitations

Due to the complex heuristic nature of neural networks, Large Language Models (LLMs) inherit certain features as they evolve which makes their output answers a subject of uncertainty studies. Despite the fact that for the most part of their operations LLMs do produce answers as predictions expect, they can still sometimes throw unexpected output. During our work, we have witnessed some strange behaviors from LLMs under experimentation and especially with GPT model. In this section, we list these observations with assumptive explanations as emphasized previously knowing precisely what is happening inside of an LLM is not possible. For GPT and Gemini Pro we observed the following behaviors and issues:

- Hallucinations: This problem comes from the inherent bias in LLMs where the model would provide output answers that are far from the context provided in input. We witnessed GPT having this problem of hallucinations during experimentation on several occasions. According to J. Song et.al [7], this comes from the False Negatives problem, and this is true for this study as the number of labeled False Negatives makes more than 25% of the whole dataset. Gemini Pro did not exhibit any hallucinations and was far more stable.

- Not responding: This can be considered as a special case of "Hallucinations" behavior. The case here is the LLM responding with a void output content without even trying to build a generic answer stating why it cannot generate an answer. We witnessed GPT behave as such in a couple of cases.

- Responding with a question: In a few cases, the LLM would respond with a question to our first question sometimes the output question seems like a rhetorical question. In other cases, it appears as if the LLM couldn't store the prompt correctly and it had memory issues, asking for the code snippet to be processed although it was supplied in the first prompt with clear delimiters. This behavior was rare in GPT and Gemini models.

- Temporal differences: In this case, the LLM changes answers and judgments between false and true with time. The answer given at time $t_1$ is quite different from the output answer at time $t_2$ (where $t_1 \neq t_2$). GPT had this problem quite often while Gemini Pro did not.

- Fickleness: This behavior is when the LLM is affected by feedback from the user after receiving the first output. The LLM would give a first answer to the LLM, that is true, the user would provide feedback stating that the LLM was wrong. The LLM would then take that new false answer as true and would use it as answer for similar future prompts. GPT is quite sensitive to this issue and can quickly change true original answers to false one after user's feedback. This can be explained by the fact that GPT version used is able to learn from user's feedback. We did not experience such behavior from Gemini Pro.

- Low confidence: The LLM would suggest a list of possible answers instead of deciding on one answer whether it is true or false. GPT didn't face this issue, For Gemini Pro, this was a rare event as well.

- Overexplaining: The LLM provides the correct answer but goes above and beyond by offering detailed explanations to the CWE in code snippet provided. The LLM clearly breaks the instructions in the input prompt.

As a result of the experiments and after thoroughly analyzing the performance and examining the behavior of each model, we concluded that Cybersecurity Domain Specific LLMs are indeed capable of running threat intelligence analysis in the software security area but still not surpassing the General LLMs capabilities. These Domain Specific LLMs are more stable than General LLMs and can be of great utility in the near future. As of now, the general performance results might seem similar to General LLMs but going deeper into the analysis and by assessing behaviors from both models, we are convinced that these Domain Specific LLMs can significantly surpass General LLMs when they reach their phase of maturity.

## 7. Conclusion

Our study on the application of Large Language Models (LLMs) for software vulnerability detection and Common Weakness Enumeration (CWE) identification has yielded several significant insights. We focused on two prominent models: GPT-3.5 from OpenAI and Gemini Pro from Google, evaluating their performance on the DiverseVul dataset.

Key findings include:

1. **Vulnerability Detection Capability**: Both GPT-3.5 and Gemini Pro demonstrated the ability to detect software vulnerabilities in the DiverseVul dataset, albeit with varying degrees of success. This confirms the potential of LLMs in the field of software security analysis.

2. **Performance Metrics**:
   - GPT-3.5 excelled in precision (88.89%) and overall accuracy (68.75%), indicating its strength in correctly identifying vulnerabilities when it flags them.
   - Gemini Pro showed superior recall (63.13% vs. 43.56% for GPT-3.5), suggesting it's more adept at identifying a higher proportion of actual vulnerabilities.

3. **CWE Identification**: Both models struggled with accurate CWE number identification, with GPT-3.5 achieving 13.13% accuracy and Gemini Pro 10.61%. This highlights a significant area for improvement in future iterations.

4. **Model Behaviors**: We observed several interesting behaviors, including:
   - Hallucinations and non-responses, particularly in GPT-3.5
   - Temporal differences in GPT-3.5's responses
   - Gemini Pro's tendency to be more conservative in vulnerability detection

These findings underscore both the potential and current limitations of using LLMs for software vulnerability detection. The models' ability to identify vulnerabilities without specific training in this domain is encouraging, but their performance is not yet at a level suitable for standalone use in critical security applications.

In conclusion, while GPT-3.5 and Gemini Pro show promise in software vulnerability detection, significant improvements are needed, particularly in CWE identification accuracy. As LLMs continue to evolve, their integration into software security workflows presents an exciting frontier for enhancing code quality and security practices. However, current limitations necessitate careful consideration and likely hybrid approaches when applying these models to critical security tasks.

**References**

[1] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in 2017 IEEE Symposium on Security and Privacy (SP), 2017, pp. 595–614.

[2] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "Vulpecker: An automated vulnerability detection system based on code similarity analysis," in Proceedings of the 32nd Annual Conference on Computer Security Applications, ser. ACSAC '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 201–213. [Online]. Available: https://doi.org/10.1145/2991079.2991102

[3] Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. Wagner, "DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection", Symposium on Research in Attacks, Intrusions and Defenses (RAID '23), October 16–18, 2023, Hong Kong, China. ACM, New York, NY, USA, 15 pages. Available: https://doi.org/10.1145/3607199.3607242

[4] R. Jensen, V. Tawosi, S. Alamir, "Software Vulnerability and Functionality Assessment using LLMs", JP Morgan AI Research, London, UK, March 13th, 2024, arXiv:2403.08429.

[5] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. V. Le, D. Zhou, "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models", Google Research, Brain Team, January 10th, 2023, arXiv:2201.11903.

[6] Common Weakness Enumeration List, MITRE, December 20th, 2024, Available: https://cwe.mitre.org/about/index.html.

[7] J. Song, S. Yu, S. Yoon, "Large Language Models are Skeptics: False Negative Problem of Input-conflicting Hallucination", Data Science & AI Laboratory, Seoul National University, Korea, June 20th, 2024, arXiv:2406.13929.

[8] Accuracy vs. precision vs. recall in machine learning: what's the difference?, Classification metrics guide, Evidently AI, Last updated: October 1, 2024.

[9] Mateus-Coelho, Nuno, and Manuela Cruz-Cunha, editors. Exploring Cyber Criminals and Data Privacy Measures. IGI Global, 2023. https://doi.org/10.4018/978-1-6684-8422-7

[10] R. F. Abu Hweidi, M. Jazzar, A. Eleyan and T. Bejaoui, "Forensics Investigation on Social Media Apps and Web Apps Messaging in Android Smartphone," 2023 International Conference on Smart Applications, Communications and Networking (SmartNets), Istanbul, Turkiye, 2023, pp. 1-7, [Online]. Available: 10.1109/SmartNets58706.2023.10216267.

[11] MobilEdit. "MobilEdit." Internet: https://www.mobiledit.com, 2023 [Dec. 10, 2023].

[12] Mateus-Coelho, Nuno Ricardo, et al. "POSMASWEB: Paranoid Operating System Methodology for Anonymous and Secure Web Browsing." Handbook of Research on Cyber Crime and Information Privacy, edited by Maria Manuela Cruz-Cunha and Nuno Mateus-Coelho, IGI Global, 2021, pp. 466-497. https://doi.org/10.4018/978-1-7998-5728-0.ch023

[13] FinalMobile. “Finalmobile” Internet: http://fmf.finaldata.com/Download/fmf4.html, 2023 [Dec. 10, 2023].

[14] L. Rosselina, Y. Suryanto, T. Hermawan and F. Alief, "Framework Design for the Retrieval of Instant Messaging in Social Media as Electronic Evidence," 2020 7th International Conference on Electrical Engineering, Computer Sciences and Informatics (EECSI), Yogyakarta, Indonesia, 2020, pp. 209-215, [Online]. Available: 10.23919/EECSI50503.2020.9251888.

[15] A. Dawabsheh and M. Owda, "In-Vehicles Infotainment System Forensics Case Study," 2023 International Conference on Information Technology (ICIT), Amman, Jordan, 2023, pp. 32-37, [Online]. Available: 10.1109/ICIT58056.2023.10225982.

[16] Mateus-Coelho, N. (2021). A New Methodology for the Development of Secure and Paranoid Operating Systems. Procedia Computer Science, 181, 1207-1215. https://doi.org/10.1016/j.procs.2021.01.318