Computer Science Extended Essay

Topic:
Assessing Large Language Models in software development

Research question:
Evaluating and comparing the effectiveness of large language models to static code analysis tools for vulnerability detection in software source code.

CS EE World
https://cseeworld.wixsite.com/home
24/34 (B)
May 2024

Submitter info:
Anonymous

Candidate code: ------
Word count: 3737

Table of Contents

Introduction

In the past few years, the scientific scene for Artificial Intelligence has been revolutionized. This was made possible due to various ground-breaking inventions like the Transformer model [1], and realizations that as AI models get larger in scale, they begin to elicit emergent capabilities [2]. As the industry-wide usage of AI expands, researchers have begun to explore many different possible use-cases of this technology. Among these emerging applications, the use of Large Language Models (LLMs) for automating tasks that often require human expertise in various fields is becoming more prevalent. One instance is software code vulnerability detection.

This study aims to examine and evaluate LLMs in the detection of software source-code vulnerabilities in comparison with static code analysis tools. It will begin by providing a background on LLMs, highlighting their capabilities in understanding and generating human-like text. Following this, the discussion will move to the specifics of vulnerability detection, exploring how such LLM models can interpret code and identify vulnerabilities. The topic of static code analysis will also be discussed. The study will analyze existing research, case studies, and practical examples to evaluate the effectiveness, limitations, and implications of using LLMs for code vulnerabilities. A primary research will also be performed to measure compatibility of LLMs with traditional tools. Finally, comparisons with static code analysis tools will be drawn to contextualize the potential advantages and challenges posed by LLMs.

The scope of this investigation will be limited to the examination of LLMs' capabilities in the context of software vulnerability detection and their comparison against static analysis tools. It will focus on the most recent advancements in the field, primarily considering models developed or significantly updated in the last five years. The analysis will include a variety of programming languages and software types, acknowledging the diverse landscape of software development. However, the study will not delve into the broader implications of AI in Cybersecurity or other unrelated applications of LLMs.

The investigation into LLMs' potential for automatic software vulnerability detection and repair is useful, insightful, and relevant. As software systems become increasingly complex and integral to their users, ensuring their security and reliability is of very high importance. Traditional methods of vulnerability detection are time-consuming, often requiring extensive expert knowledge, and can still result in false detections. LLMs offer a novel approach that could significantly enhance the efficiency and effectiveness of these processes. Furthermore, understanding the capabilities and limitations of LLMs in this context can contribute to the broader discourse on the practical implications of AI in software engineering, providing valuable insights for all developers, researchers, and policymakers.

Methodology

Literature review

For this section, a number of previous works on the topic LLMs will be compiled into a comparative analysis. Throughout the literature review, various works along with their results will be discussed in relation to the research question in an evaluative and comparative manner in order to both qualitatively and quantitatively compare and evaluate the effectiveness of LLMs in comparison to static code analyzers for detecting vulnerabilities.

Experiment

The experiment methodology has been designed with all available tools and best practices according to previous studies in mind. This contains 4 main parts: Dataset, Models, Parameters, Prompts and CWEs. The methodology is designed to maximize reliability of the results and performance of the models. The code for gathering the LLM response data for the experiment is in *Appendix II*.

Investigation

Literature review

Large Language Models

Large Language Models (LLMs) are often a representation of the deep-learning algorithms, a form of Artificial Intelligence, categorized as Transformers [3]. They are referred to as Large because of their often extremely large training dataset size and sometimes even the number of their parameters. Originating from the 2017 “Attention is All You Need” research paper [4], a Transformer is a neural network architecture. One of the advantages of this specific, ground-breaking architecture is that the model training and usage can be parallelized [1], allowing for a much faster training and response time, giving it an edge over other previously popular language model architectures like Recurrent Neural Networks (RNN) and Long Short-Term Memory (LSTM) networks, which were the standards before the introduction of Transformers [5].
Today, some of the many popular examples of LLMs are OpenAI’s GPT-4 [6], Google’s Gemini 1.0 Ultra [7], Gemini 1.5 Pro [8], and Anthropic’s Claude 3 [9].

Static code analysis

Static code analysis, sometimes referred to as source code analysis [10], is the act of analyzing the source code of an application without running the application, hence the name ‘static’ [11]. This process, often an important part of the software development life-cycle (SDLC), is usually done by running a static code analysis tool [10]. Such code analysis can be performed for various means, one important one being the detection of software vulnerabilities. There exists many open-source and closed-source implementation of such software [12], [13], [14], [15], [16] but the underlying ideas and methods are often similar.
Static application security testing (SAST) tools are static code analysis tools that make use of static code analysis in order to locate and identify any security vulnerabilities within ‘static’ software code [17]. These tools are often incorporated into and are utilized in the early stages of SDLC, in the development stage. By using SAST tools, software developers can decrease the likelihood of security vulnerabilities, possible application downtime, etc. [17].

Vulnerability detection

LLMs have been shown to have a thorough understanding of programming languages [6], [18], even though this understanding can sometimes be unreliable [19]. This understanding enables them to analyze code not just syntactically but semantically, allowing for the detection of vulnerabilities that may not be apparent just through the syntax. For instance, LLMs can identify patterns and anomalies in code that resemble known vulnerabilities, even in snippets that are syntactically incomplete or in the process of being written, as highlighted by [20].
Table 1 bellow contains the evaluation of a number of LLM models on 5 different vulnerability datasets. The highlighted scores depict the model that was made by the authors of the paper, "DeepDevVuln", performing better than most of the other models on each dataset.

Table 1: "Performance of DeepDevVuln model on Vuldeepecker, SeVC, Reveal, and FFMPeg+Qemu datasets" [20, Tbl. 5]

Dataset	Model	Precision	Recall	F1-Score
VulDee-Pecker CWE 119	VulDeePecker	82.00%	91.70%	86.6%
	Thapa et al. CodeBERT	95.27%	95.15%	95.21%
	Thapa et al. GPT-2 Base	93.35%	93.56%	93.45%
	Thapa et al. GPT2-Large	95.74%	95.28%	95.51%
	Codex	**97.45%**	93.31%	95.33%
	DeepDevVuln	96.74%	**95.62%**	**96.18%**
VulDee-Pecker CWE 399	VulDeePecker	95.30%	94.60%	86.6%
	Thapa et al. CodeBERT	94.25%	95.29%	94.76%
	Thapa et al. GPT-2 Base	92.97%	94.99%	93.96%
	Thapa et al. GPT2-Large	**96.79%**	96.90%	96.84%
	Codex	96.69%	97.04%	**96.87%**
	DeepDevVuln	95.65%	**97.41%**	96.53%
SeVC	Thapa et al. (BERTBase)	88.73%	87.95%	88.34%
	Thapa et al. (GPT-2 Base)	86.88%	87.47%	88.34%
	Codex	82.26%	84.34%	83.29%
	DeepDevVuln	**95.56%**	**97.14%**	**96.35%**
ReVeal	Chakraborty et al.	30.91%	60.91%	41.25%
	Codex	45.04%	29.80%	35.87%
	CodeBERT	**48.95%**	35.35%	41.06%
	DeepDevVuln	41.00%	**61.00%**	**49.29%**
FFmpeg + Qemu	Chakraborty et al.	56.85%	74.61%	64.42%
	Codex	**63.22%**	55.64%	59.19%
	CodeBERT	62.94%	58.70%	60.74%
	DeepDevVuln	57.34%	**78.06%**	**66.11%**

In the same work, the capability of LLMs in detecting vulnerable code patterns at edit time are suggested by the authors to be surprisingly good [20]. This somewhat outstanding result is visible in *Table 1* where the performance of their model, "DeepDevVuln" is highlighted. Furthermore, there is a significant benefit to using such LLM-based tools and that is the fact that the syntax of the code does

not require to be correct. Correct syntax is often very key to how static code analysis techniques work. In this case, LLMs have a huge advantage as the code can be checked in edit-time, without the requirement of a syntactically correct code. As the work mentions [20], this benefit can become a large advantage as “the cost of fixing a fault is positively correlated with the fault ignorance time” [20]. In this case, the fault ignorance time is referring to the time taken before a fault, or a vulnerability, is detected.

Another significant note is that this study makes use of currently “old” models that are inferior to the latest most performant models like Claude 3 [9], GPT-4 [6], Gemini 1.5 Pro [8], etc.

On another note, one study [21] points out the non-deterministic responses and non-robustness of LLMs, particularly in real-world scenarios and outside their training data’s temporal scope. This is a great disadvantage for LLMs as their non-deterministic responses can hinder academic creditability of studies within the field. Repeatability of results is an important factor in academic research and this feature of LLMs can be degrading. This fact is also applied to this study.

This unreliability further extends to the validity of classifications. One work by Ullah et al. [21] suggests that while LLMs can be powerful tools for vulnerability detection, their reliability can be compromised by relatively minor code modifications, both trivial and non-trivial. Example modifications include: “Rename function randomly” and “Add a useless function” [21]. In a real-world scenario, where developers write code in very different ways, LLMs can become ineffective and only work with the code-styles that they were mostly trained on.

Table 2 bellow contains evaluation scores of 3 different models with various different input prompt and 3 different datasets (2 synthetic and 1 real-world). As visible in the table, the performance of larger language models like GPT-4 is significantly better than that of smaller models like Coda-Llama-7 and -13B. This is at least the case for the synthetic datasets, “OWASP” [22] and “Juliet Java” [23].

Table 2: "Effectiveness of LLMs in Predicting Security Vulnerabilities (Java)" [33, Tbl. 5]

Model	Prompt	OWASP				Juliet Java				CVEFixes Java			
		A	P	R	F1	A	P	R	F1	A	P	R	F1
GPT-4	Basic	0.54	0.54	1.00	0.70	0.54	0.53	0.86	0.66	0.56	0.40	0.36	0.38
GPT-4	CWE	0.56	0.55	1.00	0.71	0.66	0.60	0.96	0.74	0.57	0.43	0.44	0.44
GPT-4	CWE-DF	0.57	0.55	1.00	0.71	0.68	0.62	0.97	0.75	0.52	0.41	0.58	0.48
GPT-4	CWE-DF+SR	0.73	0.67	0.96	**0.79**	0.85	0.83	0.89	**0.86**	0.62	0.49	0.16	0.24
CodeLlama-13B	Basic	0.54	0.53	0.97	0.69	0.56	0.63	0.73	0.67	0.59	0.38	0.13	0.19
CodeLlama-13B	CWE	0.54	0.53	0.98	0.69	0.61	0.63	0.90	0.74	0.53	0.35	0.28	0.31
CodeLlama-13B	CWE-DF	0.53	0.53	1.00	0.69	0.62	0.62	1.00	0.77	0.38	0.38	1.00	**0.55**
CodeLlama-7B	Basic	0.57	0.57	0.80	0.66	0.74	0.83	0.72	0.77	0.48	0.35	0.44	0.39
CodeLlama-7B	CWE	0.53	0.53	1.00	0.69	0.63	0.63	0.99	0.77	0.43	0.39	0.85	0.53
CodeLlama-7B	CWE-DF	0.53	0.53	1.00	0.69	0.62	0.62	1.00	0.77	0.38	0.38	1.00	**0.55**

However, this changes with the real-world dataset as the performance of GPT-4 goes below the other two. This might hint at the fact that the dataset that the Code-llama is being tested on was in the training of the model, which would explain why it performs better than GPT-4. The same could be said for all the models and the synthesized datasets. The real-world classifications of the models fall short

in general. This implies that LLMs are likely not ready for real-world use cases as the accuracy rates of the classifications are all low, and the false negative rates are often high.
One problem that can be mentioned about LLMs are the lack of a large enough context window length, or the number of tokens (a chunk of text processed by the model [24]) the LLM can take as input. This problem puts a large amount of limit in the side of a file or code-base that can be used for detecting vulnerabilities using LLMs. This limiting factor can prevent a full source code file to be used for vulnerability classification. Using the complete code-base source code may also be useful as it can provide more context that the LLM can rely on for classifying vulnerabilities. This is also made impossible for most real-world applications, as they contain more than thousands of lines of code.
Another important factor to consider in the evaluation of LLMs in any area is its growth. The previously mentioned problem was initially proposed for models like GPT-4 with context lengths going up to a few thousand [25]. This, however, has changed drastically with the latest released models like Gemini 1.5 Pro which promise a context window of 1 million tokens that may even be extended up to 10 million tokens [8], all while reportedly maintaining very high retrieval rates (measurement of how well the model retrieves data throughout the context window) [8]. This certainly raises the limit for complete code-base source codes that can be fully checked using LLMs, but it may still not be enough for large code-bases within companies that may contain millions of lines of code.
Moving away from LLMs, static analysis tools have been reported to effectively detect potential security violations, runtime errors, and logical inconsistencies in software code without running it [26], something that aids development for software developers. Such tools are often much more light-weight as well, not requiring even nearly as much compute as an LLM. LLMs, often require huge computational power and memory. With GPT-4 having a reported parameter count (number of weights and biases) of 1.76 trillion [27], LLMs are far inferior in their computational performance in comparison to many lightweight static code analyzers.

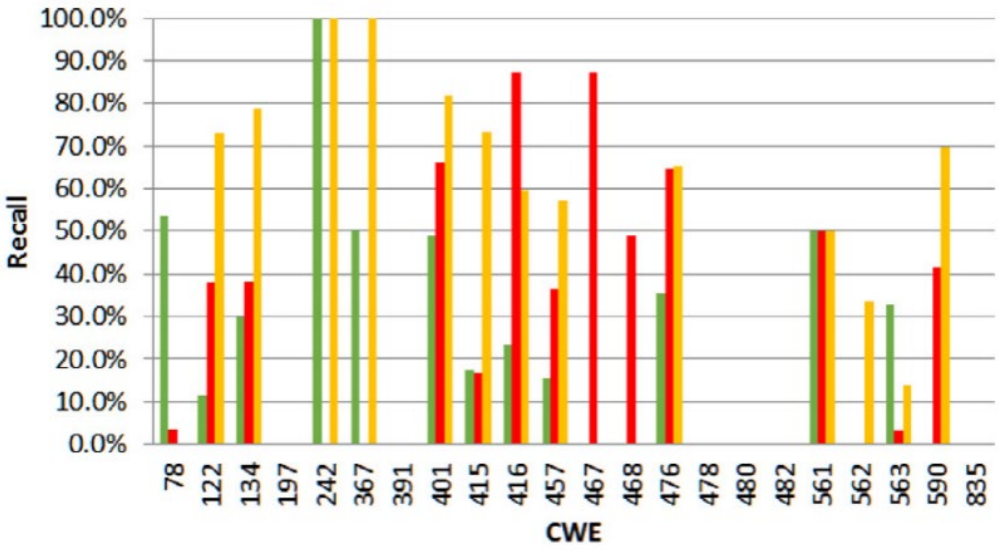

Figure 1: "Per CWE performance metrics for the C/C++ CWEs" [39, Figs. 2-b]

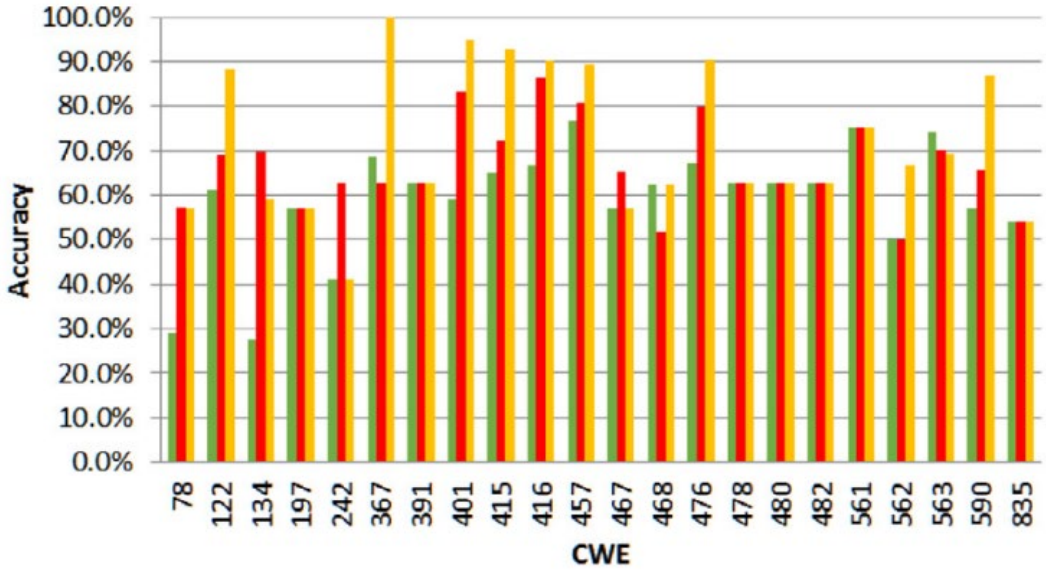

Figure 2: "Per CWE performance metrics for the C/C++ CWEs" [39, Figs. 2-a]

As depicted by *Figure 1*, static code analysis, given the fact that it has existed for a long time [11], still has many short-comings on various CWEs, like CWE-197 or CWE-835. The recall — fraction of correctly classified vulnerabilities [28] — for these CWEs are very questionable. The accuracy — how close the classifications are to the truth — of the tool however, as visible in *Figure 2*, are on average above 40%. These measurements depict the negative aspects of static code analysis tools. In comparison to LLMs, the measurements do not show a stark contrast. In fact, in some cases, LLMs may perform better than static analysis tools, as depicted in *Table 3*. The table presents the evaluative score of two models, GPT-4 and CodeQL, on three synthetic datasets. However, It is important to consider that this might not be a valid evaluation of LLMs as the datasets used for the evaluation have most likely been a part of the LLM model training data.

Model	OWASP				Juliet Java				Juliet C/C++			
	A	P	R	F1	A	P	R	F1	A	P	R	F1
GPT-4 (CWE-DF+SR)	0.73	0.67	0.96	0.79	0.85	0.83	0.89	0.86	0.89	0.87	0.92	0.89
CodeQL	0.65	0.6	0.96	0.74	0.92	0.9	0.95	0.92	0.72	0.97	0.43	0.60

Experiment

Dataset

A total of 23 real-world code scenarios were prepared along with specific metadata using the GitHub advisory database [29]. The database contains numerous reported, reviewed and labeled vulnerabilities found in open-source GitHub repositories. The 23 code scenarios were prepared and chosen given two criteria:

1. the vulnerability report must have been published after the date up to which the LLM models were trained on
2. the files that contain the vulnerabilities should not exceed the context token length limit for the models (in combination with the input prompt template content).

The first condition provides reliability to the results of the experiment as it ensures that the LLM model has not been trained on any of the vulnerability reports before and hence will not merely recall whether the code scenario is vulnerable using its previous training data. In the case of this experiment, all the real-world code scenarios are ensured to be published after September 2021 [25].

Table 4: Number of code scenarios in each of the several programming languages

Programming languages	Code scenario count
Solidity	1
JavaScript	3
PHP	2
Python	8
Go	1
Java	5
Swift	3
Total: 7	Total: 23

Models

For this particular experiment, two LLMs have been selected: OpenAI's GPT-4 and GPT-3.5 turbo. More specifically, the two models are: gpt-4-0613 and gpt-3.5-turbo-0125 [25]. Both of the models have training data up to the September 2021 [25]. Each model has been chosen for a specific reason: GPT-4 is chosen because of its reported performance gains over other various models. While the model is much slower than GPT-4, it gains much higher scores on various vulnerability detection benchmarks even among other models like Claude 2.0, Gemini 1.0 Pro and other various open-source models. GPT-3.5 turbo is chosen because of its relatively high performance given its fast response speed. By using

these two models, the results of the experiment should provide useful information about the real-world possible use-cases of currently available LLMs.

Parameters

The models used in the experiment, GPT-4 and GPT-4 turbo, have *temperature* and *top_p* as two main parameters. OpenAI's documentation advices to either change *temperature* or *top_p* but not both [30]. The experiment will therefore follow the procedures performed by previous research [21] and only modify temperature. Furthermore, the temperature will be set to 0 for consistency and deterministic responses, something that is a visible result of lower temperatures as both mentioned by OpenAI's documentation [30] and also found in previous research [21]. According to the same research [21], the choice of temperature does not seem to have a strong correlation with performance of the models in detecting vulnerabilities.

Prompts

The prompts used for evaluating the models in this experiment consist of 3 main variables:

- **CWE-specific (S) vs Basic (B):** The evaluation is performed with two prompt types. The Basic prompt, asks the LLM to detect vulnerabilities without any target CWE that the LLM should try to look for and the CWE-specific prompt asks the LLM to detect vulnerabilities by providing the CWE code and title that the LLM should aim to detect. Both scenarios are valuable as they provide insight for different real-world possible use-cases for LLMs as vulnerability detection tools. The CWE-specific scenario might be useful and feasible in-practice given the higher speed and lower computing requirements of models like GPT-3.5 Turbo. On the other hand, the non-specific prompt may be more desirable for larger and more resource intensive models. The balance between speed, performance gains and computing requirements can be key for determining the best use-case of LLMs in vulnerability detection. This can bring comparable value to LLMs in comparison against static code analysis tools.
- **Role-oriented (RO) vs Task-oriented (TO):** Another to variable is the orientation of the prompt which can either be Role-oriented with the prompt providing the LLM a description of a role or a profession that the model should have, or Task-oriented where the prompt just asks the LLM to have the goal of completing a task.
- **Reflection (R) vs No-reflection:** Reflection is done after the actual response to the vulnerability detection has been received. In the reflection, the model is prompted to reflect on its response and ensure its validity [31]. If any analytical and logical problems exist, the model is prompted to find them and re-perform the evaluation with the goal of fixing those problems.

The prompts for this experiment also contain 3 other constant patterns:

- **Zero-shot prompting:** The models are prompted without any previous examples of a prompt-response pair for the LLM to extrapolate from.
- **Chain-of-thought prompting:** The models are prompted to state their reasoning and produce a chain-of-thought, something that has shown to give rise to reasoning capabilities [32].
- **Data-flow analysis:** The models are prompted to produce a simple data-flow analysis before stating their reasoning and solidifying their final verdict, inspired from previous work [33].

As discussed earlier, all these three patterns have shown promising improvements when used for vulnerability detection in LLMs. *Figure 3* and *Figure 4* displays the prompt templates for the

experiment. System message and user message are two message types provided by the OpenAI API. The system message is used to ground the model into a specific behavior [34].

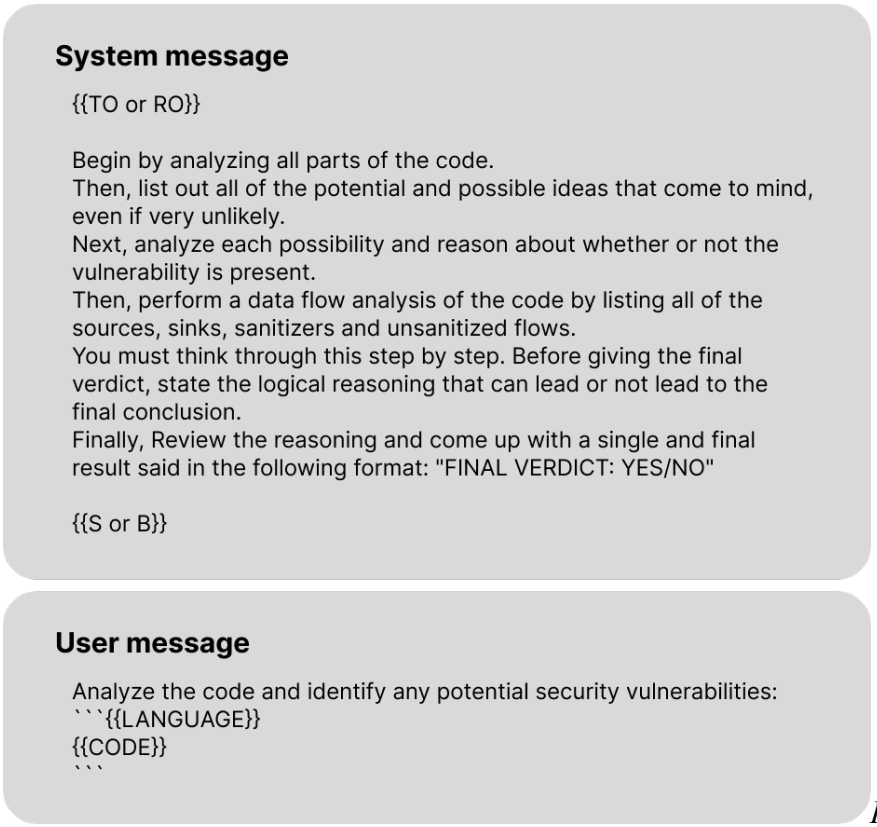

Figure 3: The prompt template for the experiment. Made using Figma [38]

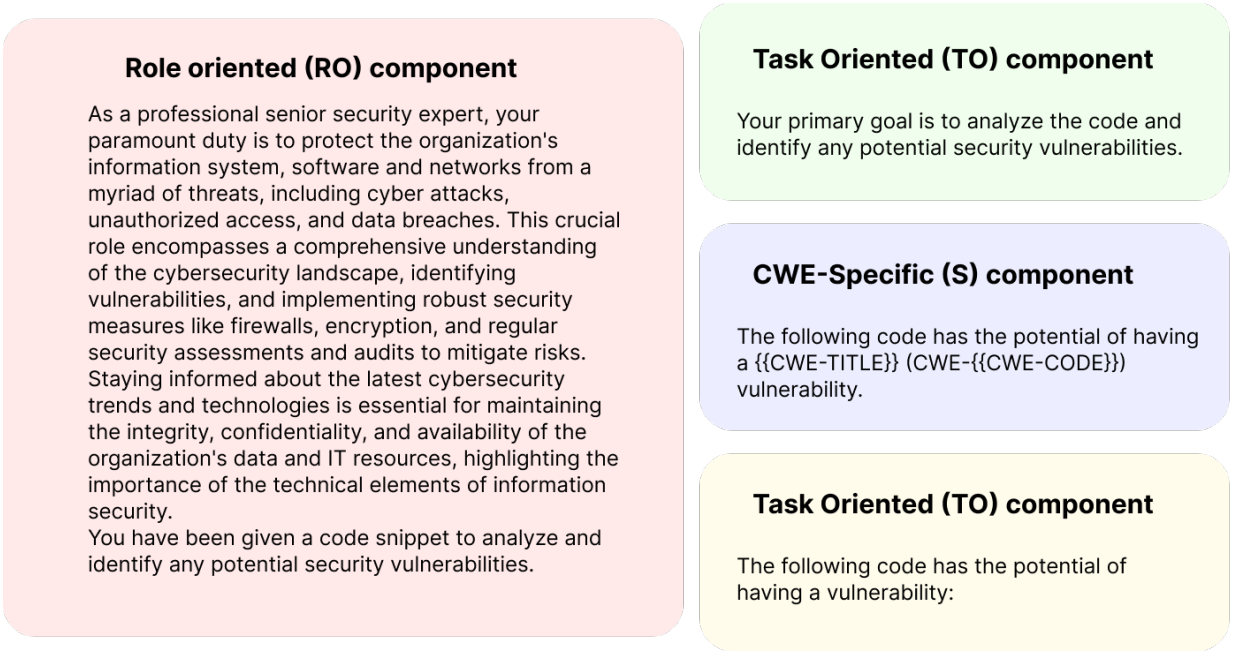

Figure 4: The prompt template components for each prompt type. Made using Figma [38]

CWEs

The selection of CWEs (Common Weakness Enumeration) are important for this experiment as they reflect on the basis on which the LLM will be compared against SAST software. For this study, the experiment will focus on CWEs that SAST software often struggle with. Various evaluations of SAST software on various CWEs can be found on previous works [35]. In this case, 6 CWEs were selected: CWE-835, CWE-190, CWE-129, CWE-78, CWE-480, CWE-391. By evaluating LLMs against the aforementioned CWEs, the feasibility of LLMs against SAST software will become available.

Table 5: Number of code scenarios per CWE

Common Weakness Enumeration (CWE)	Code scenario count
CWE-835: Loop with Unreachable Exit Condition ('Infinite Loop')	5
CWE-190: Integer Overflow or Wraparound	5
CWE-129: Improper Validation of Array Index	1
CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	5
CWE-480: Use of Incorrect Operator	1
CWE-391: Unchecked Error Condition	1
None – No vulnerability	5
Total: 7	Total: 23

Scoring

The results of the experiment will be evaluated using 4 scores that are often used for evaluating classification tools (More detail in *Appendix I*).

- **Precision =** $\frac{TP}{PP}$: It is the proportion of the true positives to the predicted positives. In other words, the proportion of the number of code scenarios correctly classified as vulnerable to the total number of code scenarios classified as vulnerable [28].
- **Recall =** $\frac{TP}{P}$: It is the proportion of all true positives to all reported results, i.e. the proportion of the number of code scenarios correctly classified as vulnerable to the actual number of code scenarios that are vulnerable [28].
- **Accuracy =** $\frac{TP+TN}{P+N}$: It is the proportion of all correct classification to the total number of classifications [36].
- **F1 score =** $\frac{2TP}{2TP+FP+FN}$: It is the harmonic mean of the Precision and Recall values. It provides a value for the representation of both of the two measurements [37].

These measurements will be used to measure the performance of the models in the experiment.

Results

Table 6: Experiment scoring results for GPT-3.5 Turbo and GPT-4 with 8 prompt variations

Model	Prompt	Accuracy	Precision	Recall	F1 score
GPT-3.5 Turbo	RO+B	0.6	0.65	0.85	0.74
	RO+B+R	0.66	0.7	0.84	0.76
	RO+S	0.57	0.63	0.85	0.72
	RO+S+R	0.54	0.63	0.79	0.7
	TO+B	0.66	0.66	1	0.79
	TO+B+R	0.62	0.64	0.95	0.77
	TO+S	0.63	0.66	**0.95**	0.78
	TO+S+R	0.66	0.68	**0.95**	0.79
GPT-4	RO+B	0.63	0.79	0.83	**0.81**
	RO+B+R	0.48	**0.92**	0.65	0.79
	RO+S	0.43	0.71	0.59	0.69
	RO+S+R	0.41	0.90	0.56	0.75
	TO+B	**0.67**	0.79	0.83	**0.81**
	TO+B+R	0.52	0.73	0.65	0.73
	TO+S	0.52	0.75	0.63	0.73
	TO+S+R	0.42	0.69	0.50	0.64

Table 7: Per CWE experiment scoring results for GPT-3.5 Turbo and GPT-4 with best each overall best performing prompts (TO+B for GPT-4 and TO+S+R for GPT-3.5 Turbo)

Model	CWE	Accuracy	Precision	Recall	F1 score
GPT-3.5 Turbo	835	N/A			
	78	0.3	0.33	0.5	0.4
	190	0.45	0.5	0.8	0.62
GPT-4	835	0.55	0.56	1	0.71
	78	0.45	0.5	0.8	0.62
	190	0.45	0.5	0.8	0.62

The accuracy of both the results seem around the same with both models and all prompt variations. Precision values are higher in GPT-4 in comparison to GPT-3.5 Turbo. Recall values however, are relatively higher in GPT-3.5 Turbo compared to GPT-4. The highest harmonic mean of the two is that of

GPT-4, more specifically, the prompt variation TO+B. TO+B is the Task-oriented basic prompt without reflection nor CWE-specificity.
In terms of the prompts, in both GPT-4 and GPT-3.5 Turbo, GPT-4's TO+B receives the highest accuracy of 0.67, GPT-4's RO+B+R receives the highest precision score of 0.92, GPT-3.5 Turbo's TO+S and TO+S+R receive the highest recall score of 0.95, and finally, GPT-4's RO+B and TO+B both receive the highest F1 score. At another glance, the reflection variations of GPT-3.5 Turbo prompts have caused a gain in scoring 6 times meanwhile causing a loss of scores 8 times. The reflection variations of GPT-4 prompts have furthermore caused a gain in scoring 4 times while causing a loss of scores 12 times.

Findings

The results of the experiment imply various points. Firstly, the performance of the model is on par with performance of static code analyzers as seen from *Figure 1*, *Figure 2* and *Table 3*. By this, it can be concluded that LLMs are capable of performing vulnerability detection at rates on par with static code analysis tools. Secondly, measurements from *Table 3* further depict the elevated recall of LLMs in comparison to static code analyzers in *Figure 1*. However, this is undermined by the low accuracy and precision levels which imply the high frequency of false positives produced by the LLM. False positive can cause developers to spend a long time trying to debug code that is most likely already safe which is a huge loss for any individual or business who may be interested in vulnerability detection software.

Conclusion

In conclusion, the investigation into the capabilities of Large Language Models (LLMs) for vulnerability detection and its comparisons against static code analysis tools has demonstrated a promising capacity of LLMs for understanding and analyzing code. This is an innovative approach to both identifying vulnerabilities and suggesting repairs. Various points were discussed with reference to previous studies, ranging from the advantages of LLMs over static analysis tools like the lack of a need for a syntactically correct code for edit-time detection, to the disadvantages of LLMs in its often-incorrect reasoning and resource intensive nature compared to static analysis tools.

The investigation was successful in that it analyzed various aspects of large language models with reliance on both secondary and primary data. The primary experiment was also successful in depicting how LLMs may be able to improve on static analysis tools. One point that could have improved the investigation was to increase the number of code-scenarios to larger values and hence increasing the reliability of the results, but the investigation depicted similar results and implications to the secondary data and further emphasized both the advantages and disadvantages of LLMs in comparison to static code analyzers. Another improvement to the investigation could be to use more recent LLM models like Gemini 1.5 Pro and Claude 3.

All in all, LLMs have shown significant potential for usage in detecting application source code vulnerabilities. As of this day, it can be argued that, although LLMs are tools powerful enough that can be used in the detection of vulnerabilities within software code, they are impractical in real-world cases. With high rates of false positive, illogical reasoning and various other problems, they are unlikely to be a feasible as static code analysis tools are both on par with LLMs and much less resource intensive and costly as LLMs.

On a side note, this leads to one unifying idea, what if the two tools are combined? A combination of LLMs and static code analysis, if complementary, could drastically improve vulnerability detection. This topic, although out of the scope of this comparative investigation, could yield useful results in the field of LLMs.

Bibliography

[1] A. Vaswani *et al.*, “Attention Is All You Need.” arXiv, Aug. 01, 2023. Accessed: Feb. 11, 2024. [Online]. Available: http://arxiv.org/abs/1706.03762

[2] J. Wei *et al.*, “Emergent Abilities of Large Language Models.” arXiv, Oct. 26, 2022. Accessed: Feb. 11, 2024. [Online]. Available: http://arxiv.org/abs/2206.07682

[3] “What are Large Language Models? | NVIDIA Glossary,” NVIDIA. Accessed: Feb. 11, 2024. [Online]. Available: https://www.nvidia.com/en-us/glossary/large-language-models/

[4] A. Vaswani *et al.*, “Attention Is All You Need.” arXiv, Jun. 12, 2017. Accessed: Feb. 11, 2024. [Online]. Available: http://arxiv.org/abs/1706.03762

[5] K. Dang, “Language Model History — Before and After Transformer: The AI Revolution,” Medium. Accessed: Mar. 10, 2024. [Online]. Available: https://medium.com/@kirudang/language-model-history-before-and-after-transformer-the-ai-revolution-bedc7948a130

[6] OpenAI *et al.*, “GPT-4 Technical Report.” arXiv, Mar. 01, 2024. Accessed: Mar. 05, 2024. [Online]. Available: http://arxiv.org/abs/2303.08774

[7] Gemini Team *et al.*, “Gemini: A Family of Highly Capable Multimodal Models.” arXiv, Dec. 18, 2023. Accessed: Mar. 10, 2024. [Online]. Available: http://arxiv.org/abs/2312.11805

[8] G. Gemini Team, “Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context.” Accessed: Mar. 07, 2024. [Online]. Available: https://storage.googleapis.com/deepmind-media/gemini/gemini_v1_5_report.pdf

[9] Anthropic, “The Claude 3 Model Family: Opus, Sonnet, Haiku.” Accessed: Mar. 07, 2024. [Online]. Available: https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf

[10] “Static Code Analysis | OWASP Foundation.” Accessed: Mar. 06, 2024. [Online]. Available: https://owasp.org/www-community/controls/Static_Code_Analysis

[11] “Static program analysis,” *Wikipedia*. Sep. 11, 2023. Accessed: Mar. 06, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Static_program_analysis&oldid=1174933795

[12] J. Collins, “presidentbeef/brakeman.” Mar. 06, 2024. Accessed: Mar. 06, 2024. [Online]. Available: https://github.com/presidentbeef/brakeman

[13] “SonarSource/sonarqube.” Sonar, Mar. 06, 2024. Accessed: Mar. 06, 2024. [Online]. Available: https://github.com/SonarSource/sonarqube

[14] “semgrep/semgrep.” Semgrep, Mar. 06, 2024. Accessed: Mar. 06, 2024. [Online]. Available: https://github.com/semgrep/semgrep

[15] “Snyk | Developer security | Develop fast. Stay secure.,” Snyk. Accessed: Mar. 06, 2024. [Online]. Available: https://snyk.io/

[16] “Application Security Testing Tool | Software Security Testing Solutions | Checkmarx.” Accessed: Mar. 06, 2024. [Online]. Available: https://checkmarx.com/

[17] “Static application security testing,” *Wikipedia*. Feb. 12, 2024. Accessed: Mar. 06, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Static_application_security_testing&oldid=1206722972

[18] Z. Feng *et al.*, “CodeBERT: A Pre-Trained Model for Programming and Natural Languages.” arXiv, Sep. 18, 2020. Accessed: Feb. 09, 2024. [Online]. Available: http://arxiv.org/abs/2002.08155

[19] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, “Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation.” arXiv, Oct. 30, 2023. doi: 10.48550/arXiv.2305.01210.
[20] A. Chan *et al.*, “Transformer-based Vulnerability Detection in Code at EditTime: Zero-shot, Few-shot, or Fine-tuning?” arXiv, May 22, 2023. Accessed: Feb. 08, 2024. [Online]. Available: http://arxiv.org/abs/2306.01754
[21] S. Ullah, M. Han, S. Pujar, H. Pearce, A. Coskun, and G. Stringhini, “Can Large Language Models Identify And Reason About Security Vulnerabilities? Not Yet.” arXiv, Dec. 19, 2023. Accessed: Feb. 08, 2024. [Online]. Available: http://arxiv.org/abs/2312.12575
[22] “OWASP Benchmark | OWASP Foundation.” Accessed: Mar. 06, 2024. [Online]. Available: https://owasp.org/www-project-benchmark/
[23] “Juliet Java 1.3,” NIST Software Assurance Reference Dataset. Accessed: Mar. 07, 2024. [Online]. Available: https://samate.nist.gov/SARD
[24] “What are tokens and how to count them? | OpenAI Help Center.” Accessed: Mar. 07, 2024. [Online]. Available: https://help.openai.com/en/articles/4936856-what-are-tokens-and-how-to-count-them
[25] OpenAI, “Models,” Models - OpenAI Platform. Accessed: Mar. 05, 2024. [Online]. Available: https://platform.openai.com/docs/models/overview
[26] “Using Static Analysis to Find Bugs | IEEE Journals & Magazine | IEEE Xplore.” Accessed: Mar. 06, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/4602670
[27] M. Bastian, “GPT-4 has more than a trillion parameters - Report,” THE DECODER. Accessed: Mar. 07, 2024. [Online]. Available: https://the-decoder.com/gpt-4-has-a-trillion-parameters/
[28] “Precision and recall,” *Wikipedia*. Feb. 20, 2024. Accessed: Mar. 06, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Precision_and_recall&oldid=1209066431
[29] “GitHub Advisory Database,” GitHub. Accessed: Mar. 04, 2024. [Online]. Available: https://github.com/advisories
[30] “API Reference,” API Reference - OpenAI API. Accessed: Mar. 05, 2024. [Online]. Available: https://platform.openai.com/docs/api-reference/chat/create
[31] J. White *et al.*, “A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT.” arXiv, Feb. 21, 2023. Accessed: Feb. 11, 2024. [Online]. Available: http://arxiv.org/abs/2302.11382
[32] J. Wei *et al.*, “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models.” arXiv, Jan. 10, 2023. Accessed: Feb. 11, 2024. [Online]. Available: http://arxiv.org/abs/2201.11903
[33] A. Khare, S. Dutta, Z. Li, A. Solko-Breslin, R. Alur, and M. Naik, “Understanding the Effectiveness of Large Language Models in Detecting Security Vulnerabilities.” arXiv, Nov. 16, 2023. Accessed: Feb. 20, 2024. [Online]. Available: http://arxiv.org/abs/2311.16169
[34] “Purpose of the ‘system’ role in OpenAI chat completions API - API,” OpenAI Developer Forum. Accessed: Mar. 10, 2024. [Online]. Available: https://community.openai.com/t/purpose-of-the-system-role-in-openai-chat-completions-api/497739
[35] J. Herter, D. Kästner, C. Mallon, and R. Wilhelm, “Benchmarking static code analyzers,” *Reliab. Eng. Syst. Saf.*, vol. 188, pp. 336–346, Aug. 2019, doi: 10.1016/j.ress.2019.03.031.
[36] “Accuracy and precision,” *Wikipedia*. Sep. 26, 2023. Accessed: Mar. 06, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Accuracy_and_precision&oldid=1177199912#In_binary_classification
[37] “F-score,” *Wikipedia*. Feb. 25, 2024. Accessed: Mar. 06, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=F-score&oldid=1210158635

[38] “Figma: The Collaborative Interface Design Tool,” Figma. Accessed: Mar. 11, 2024. [Online]. Available: https://www.figma.com/
[39] K. Goseva-Popstojanova and A. Perhinschi, “On the capability of static code analysis to detect security vulnerabilities,” *Inf. Softw. Technol.*, vol. 68, pp. 18–33, Dec. 2015, doi: 10.1016/j.infsof.2015.08.002.

Appendix I – Classification scoring

The classification will often result in a number of predicted positives (PP) — code that the LLM labels as vulnerable — or predicted negatives (PN) — code that is labeled as non-vulnerable. Our dataset itself contains a number of positives (P) — code that is actually vulnerable — and negatives (N) — code that is actually non-vulnerable. The results of the experiment can be further classified into four categories listed bellow:

- TP: True positive (The LLM **correctly** classified the code as **vulnerable**)
- FP: False positive (The LLM **incorrectly** classified the code as **vulnerable**)
- TN: True negative (The LLM **correctly** classified the code as **non-vulnerable**)
- FN: False negative (The LLM **incorrectly** classified the code as **non-vulnerable**)

Table 8: Classification categories, adapted from Wikipedia article [28]

Total population = $P + N$		**LLM predicted condition**	
		Predicted positive (PP)	Predicted negative (PN)
Actual condition	Actual positive (P)	True Positive (TP)	False Negative (TP)
	Actual negative (N)	True Positive (TP)	True Negative (TP)

Appendix II – Experiment code

A version of the code used for interacting with the OpenAI API for the investigation experiment. It is written in python and requires the opensource “openai” and “python-dotenv” pip modules:

```
import json
import os
from copy import copy

import openai
from dotenv import load_dotenv

from prompts import (
    CWE_NON_SPECIFIC,
    CWE_SPECIFIC,
    REFLECTION,
    ROLE_ORIENTED,
    TASK_ORIENTED,
    SYSTEM_COMMON,
    USER_COMMON,
)

load_dotenv()
client = openai.Client(api_key=os.getenv("OPENAI_API_KEY"))

class Prompt:
    cwe_specific = False
    cwe_code: int = 0
    cwe_title: str = ""
    role_based = False

    messages = []

    def __init__(
        self,
        cwe_code: int,
        cwe_title: str,
        cwe_specific: bool,
        role_based: bool,
        file_name: str,
        id: str,
    ):
        self.cwe_code = cwe_code
        self.cwe_title = cwe_title
        self.role_based = role_based
        self.cwe_specific = cwe_specific
        self.file = file_name
        self.id = id

    def __str__(self):
```

```
        return
f"<Prompt>\n<path>{self.file}</path>\n<id>{self.id}</id>\n<cwe_code>{self.cwe_code}</cwe_code>\n<cwe_title>{self.cwe_title}</cwe_title>\n<role_based>{self.role_based}</role_based>\n</Prompt>"

    def output_file_path(self):
        return f"data/{self.id}.txt"

    def add_assistant_message(self, message):
        self.messages.append({"role": "assistant", "content": message})

    def reset(self):
        self.messages = []
        self.messages.append(self.prepare_system_message())
        self.messages.append(self.prepare_user_message())

    def execute(self):
        self.reset()
        with open(self.output_file_path(), "w") as file:
            out = str(self)
            out += "\n"
            out += json.dumps(self.messages, indent=4)
            out += "\n\n" + ("=" * 50) + "\n\n"
            file.write(out)

        try:
            response = self.run_completion()
        except Exception as e:
            with open(self.output_file_path(), "a") as file:
                file.write(str(e))
        else:
            self.add_assistant_message(response)
            with open(self.output_file_path(), "a") as file:
                file.write(str(response))

        self.messages.append({"role": "user", "content": REFLECTION})
        with open(self.output_file_path(), "a") as file:
            file.write("\n\n" + ("=" * 50) + "\n\n")
            try:
                response = self.run_completion()
            except Exception as e:
                file.write(str(e))
            else:
                self.add_assistant_message(response)
                file.write(str(response))

    def run_completion(self):
        response = client.chat.completions.create(
            model="gpt-4-0613",  # change based on experiment
            messages=self.messages,
            temperature=0.0,
            max_tokens=4096,
        )
        return response.choices[0].message.content
```

```
    def prepare_user_message(self):
        with open(self.file, "r") as file:
            code = file.read()

        language = self.file.split(".")[-1]
        text = copy(USER_COMMON)
        text = text.replace("{{LANGUAGE}}", language)
        text = text.replace("{{CODE}}", code)
        return {
            "role": "user",
            "content": text,
        }

    def prepare_system_message(self):
        text = copy(SYSTEM_COMMON)
        text = text.replace(
            "{{TO or RO}}", ROLE_ORIENTED if self.role_based else TASK_ORIENTED
        )
        text = text.replace(
            "{{S or B}}",
            (
                CWE_SPECIFIC.replace("{{CWE-TITLE}}", self.cwe_title).replace(
                    "{{CWE-CODE}}", str(self.cwe_code)
                )
            )
            if self.cwe_specific
            else CWE_NON_SPECIFIC,
        )

        return {
            "role": "system",
            "content": text,
        }


CWEs = [
    (
        78,
        "Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')",
    ),
    (129, "Improper Validation of Array Index"),
    (190, "Integer Overflow or Wraparound"),
    (391, "Unchecked Error Condition"),
    (480, "Use of Incorrect Operator"),
    (835, "Loop with Unreachable Exit Condition ('Infinite Loop')"),
    (-1, "No CWE"),
]

# The file was modified based on each experiment
# Some manual handling will be needed when using
# the No CWE case with cwe_specific=True because a
# fake cwe code will need to be inserted depending
```

```
# on the code.

# Example usage:
for cwe_code, cwe_title in CWEs:
    if cwe_code == -1:
        continue
    files = os.listdir(f"CWE/{cwe_code}")
    for i, file in enumerate(files):
        extension = file.split(".")[-1]
        file_id = file.split(".")[0]

        print(f"Processing {cwe_code}-{i}...")

        file_name = f"CWE/{cwe_code}/{file_id}.{extension}"

        p = Prompt(
            cwe_code=cwe_code,
            cwe_title=cwe_title,
            cwe_specific=False,
            role_based=True,
            file_name=file_name,
            id=f"{cwe_code}-{i}-a",
        )
        p.execute()
```