Journal of Universal Computer Science, vol. 30, no. 9 (2024), 1163-1183
submitted: 27/9/2023, accepted: 30/5/2024, appeared: 14/9/2024 CC BY-ND 4.0

An Empirical Evaluation of Large Language Models in Static Code Analysis for PHP Vulnerability Detection

Orçun Çetin
(Sabanci University, Istanbul, Türkiye
https://orcid.org/0000-0001-9670-0295, orcun.cetin@sabanciuniv.edu)

Emre Ekmekcioglu
(Sabanci University, Istanbul, Türkiye
https://orcid.org/0009-0003-3790-4839, eekmekcioglu@sabanciuniv.edu)

Budi Arief
(University of Kent, Canterbury, UK
https://orcid.org/0000-0002-1830-1587, b.arief@kent.ac.uk)

Julio Hernandez-Castro
(Universidad Politécnica de Madrid, Madrid, Spain
https://orcid.org/0000-0002-6432-5328, jc.hernandez.castro@upm.es)

Abstract: Web services play an important role in our daily lives. They are used in a wide range of activities, from online banking and shopping to education, entertainment and social interactions. Therefore, it is essential to ensure that they are kept as secure as possible. However – as is the case with any complex software system – creating a sophisticated software free from any security vulnerabilities is a very challenging task. One method to enhance software security is by employing static code analysis. This technique can be used to identify potential vulnerabilities in the source code before they are exploited by bad actors. This approach has been instrumental in tackling many vulnerabilities, but it is not without limitations. Recent research suggests that static code analysis can benefit from the use of large language models (LLMs). This is a promising line of research, but there are still very few and quite limited studies in the literature on the effectiveness of various LLMs at detecting vulnerabilities in source code. This is the research gap that we aim to address in this work. Our study examined five notable LLM chatbot models: ChatGPT 4, ChatGPT 3.5, Claude, Bard/Gemini[1], and Llama-2, assessing their abilities to identify 104 known vulnerabilities spanning the Top-10 categories defined by the Open Worldwide Application Security Project (OWASP). Moreover, we evaluated issues related to these LLMs' false-positive rates using 97 patched code samples. We specifically focused on PHP vulnerabilities, given its prevalence in web applications. We found that ChatGPT-4 has the highest vulnerability detection rate, with over 61.5% of vulnerabilities found, followed by ChatGPT-3.5 at 50%. Bard has the highest rate of vulnerabilities missed, at 53.8%, and the lowest detection rate, at 13.4%. For all models, there is a significant percentage of vulnerabilities that were classified as partially found, indicating a level of uncertainty or incomplete detection across all tested LLMs. Moreover, we found that ChatGPT-4 and ChatGPT-3.5 are consistently more effective across most categories, compared to other models. Bard and Llama-2 display limited effectiveness in detecting vulnerabilities across the majority of categories listed. Surprisingly, our findings reveal high false positive rates across all LLMs. Even the model demonstrating the best performance (ChatGPT-4) notched a false positive rate of nearly 63%, while several models glaringly under-performed, hitting startlingly bad false

[1] Since 8 February 2024, Bard was renamed to Gemini. As our study was carried out earlier, we will refer to it as Bard in the rest of this work.

positive rates of over 90%. Finally, simultaneously deploying multiple LLMs for static analysis resulted in only a marginal enhancement in the rates of vulnerability detection. We believe these results are generalizable to most other programming languages, and hence far from being limited to PHP only.

Keywords: ChatGPT, Claude, Bard, Gemini, Llama-2, Static code analysis, PHP vulnerabilities, Vulnerability detection, LLM in cybersecurity
Categories: D.2.3, D2.4, I.2.5, I.2.7, K.6.5
DOI: 10.3897/jucs.134739

1 Introduction

In software engineering, a vulnerability[2] refers to a flaw in the design or implementation that provides an opportunity for attackers to abuse the system's weaknesses [Merkow and Raghavan 2010]. Addressing these vulnerabilities becomes more costly once the software has already been deployed. Hence, to reduce development costs and prevent attacks, it is crucial to detect and patch vulnerabilities during the initial phases of the software development life cycle. This cycle usually includes five important stages: (i) requirements; (ii) design; (iii) implementation; (iv) testing; and (v) deployment. Each phase requires unique cybersecurity steps and guidelines. The bulk of vulnerabilities are typically discovered during the implementation stage. During this phase, developers often employ static code analysis tools to assess the security of their code, pinpoint and fix any security vulnerabilities found. As a result of this process, both implementation and, at times, even design vulnerabilities are detected and removed or patched before moving into the later phases of the software development life cycle.

All software is susceptible to vulnerabilities, but the increasing number of vulnerabilities in websites and web-based applications has become a growing concern [Telang and Wattal 2007]. This has become an even more pressing issue, due to the society's increased dependence on web services. Any disruptions or security breaches can lead to serious adverse effects and harms not only to security but also to privacy. Furthermore, the constant online availability of these web services renders them susceptible to remote attacks at any moment, significantly expanding the potential attack surface. A large percentage of these web services are developed using the PHP programming language. Data from W3Techs shows that PHP powers around 78.9% of all websites as a server-side programming language. This means that nearly 8 out of 10 websites utilize PHP in some capacity[3].

To address the issue of software vulnerabilities, software engineering and cybersecurity communities have developed static analysis tools, which can be used to examine source code and highlight components susceptible of containing errors, ideally prior to deployment. Many static code analyzers exist, including RIPS[4] (which can be used to detect PHP vulnerabilities), FindBugs[5] (which can be used to find vulnerabilities in Java code) and CodeSonar[6] (which supports multiple programming languages). Frequently, these tools employ a specific set of predefined rules or patterns to recognize issues in the code. Static code analyzers are very valuable tools, but they can be time consuming and

[2] https://owasp.org/www-community/vulnerabilities/
[3] https://w3techs.com/technologies/details/pl-php
[4] https://github.com/robocoder/rips-scanner
[5] https://code.google.com/archive/p/findbugs/
[6] https://codesecure.com/our-products/codesonar/

labor intensive to run. Therefore, finding ways to automate the whole process can yield many benefits.

In a recent study, Ozturk et al. [Ozturk et al. 2023] have demonstrated that ChatGPT, a Large Language Model (LLM) based chatbot[7], was surprisingly able to achieve a considerably higher rate of vulnerability detection when compared to traditional, targeted, static code analyzers. This study suggests that the dynamic learning and pattern recognition capabilities of LLMs can potentially surpass the efficiency of conventional static code analyzers that rely heavily on predefined rules or patterns. Ozturk et al.'s work opens up new avenues for utilizing LLMs in enhancing software security, allowing for more affordable, easily available, resilient and secure software development practices. However, remarkably little research has been undertaken so far into the effectiveness of LLMs in terms of identifying code vulnerabilities.

In this paper, we present a comprehensive comparison to assess the effectiveness of five popular LLMs in finding vulnerabilities in code. We estimated the effectiveness of these LLMs by using 104 codes that are known to contain vulnerabilities, together with 97 patched codes that should be error-free. We use these dataset as our ground truth, as it has been generated and curated by experts in the field. While vulnerable examples were used to measure different LLMs' ability to detect security issues in the source code, patched codes were used to observe and estimate false-positive rates. To achieve a high accuracy and consistent classification of vulnerable code snippets, we employed a stringent approach, in which we asked each LLM to provide a correct Common Weakness Enumeration (CWE)[8] title, together with the number and an explanation of each vulnerability found.

Contributions. Our research contributions are mainly our new definition of the research problem and question, that identifies a gap in the literature that we aim to address, on top of a thorough literature review on the topic, along with a detailed and well-thought-of new research design and analysis methodology. To this end, we add a clear presentation of our novel findings, in a manner that supports our results and conclusions in a statistically sound way. We also describe the significance of our findings and their likely implications for the field of study, providing new avenues for research. The key findings of our paper are:

- We provided the first study measuring the effectiveness of five popular LLMs on identifying vulnerabilities in real-world code samples.

- We observed a very high detection rate even when a stricter evaluation was applied. The best performing model, which was ChatGPT-4, achieved a detection success rate of 61.5%. This was followed by another OpenAI model called ChatGPT-3.5 with a 50% correct detection.

- We found disappointingly high false positive rates among all the five LLMs. Even the best performing model (ChatGPT-4) recorded close to a 63% in false positives, while other models performed even worse, reaching a staggering false positive rates of up to 97%. *Our study suggests that, indeed, the main challenge when utilizing LLMs as static code analyzers arises from these very high false positive rates.*

- Our results revealed that ChatGPT-4 and ChatGPT-3.5 might yield the most reliable and balanced results in terms of vulnerability detection rates across different categories. For some specific vulnerability types (such as "Security Misconfiguration"

[7] https://openai.com/blog/chatgpt/

[8] https://cwe.mitre.org/

and "Vulnerable & Outdated Components"), ChatGPT-4, Claude and, in some cases, ChatGPT-3.5, have been able to detect 100% of the existing issues. On the other hand, Llama-2 and Bard exhibit significant challenges in detecting the majority of vulnerability types and have numerous instances with a dismal 0% detection rate.

- Lastly, we found that LLM-based vulnerability detection can be slightly improved when various LLMs are combined. The results after combining two LLMs were uninspiring but consistently better than in the solo approach.

2 Related work

Being able to identify – and to some extent, predict – software vulnerabilities, before these vulnerabilities can be exploited by malicious actors, is an important goal of cybersecurity, as well as part of good software engineering practices. Many developers utilize third-party tools as a resource to pinpoint vulnerabilities within their code. The majority of the current tools dedicated to identifying software vulnerabilities primarily rely on static code analysis methods.

Numerous studies have been undertaken to evaluate the efficacy of static analysis tools, aiming to assist security experts in selecting the most suitable ones. Only a few of these studies have so far investigated LLMs' ability to find vulnerabilities in source code. One of them is presented in a recent paper by Ozturk et al. [Ozturk et al. 2023]. They carried out a study in which they constructed a dataset consisting of PHP code vulnerable to 92 PHP vulnerabilities, aligned with the OWASP Top-10 web application vulnerabilities[9]. This was to assess the accuracy of 11 widely-used free open-source static code analyzers for PHP. Additionally, they evaluated the capability of an early version of ChatGPT to identify these vulnerabilities. They discovered that the efficacy rate of ChatGPT, in terms of identifying security vulnerabilities in PHP, ranges approximately between 62-68% while the best traditional static code analyzer tested exhibits a success rate of 32%. Merging the capabilities of several traditional static code analyzers would only result in a detection rate of 53%, a figure that is still significantly lower than the success rate of ChatGPT. Their ChatGPT detection rates were higher than both ChatGPT 3.5 and 4 models observed in the current study. Conversely, we employed a more conservative set of evaluation criteria wherein any minor error in CWE classification was considered to produce only a partial find. This encompasses instances where the correct CWE number was not accurately identified, but the explanation and title of the vulnerability were correctly provided. Their results show that the dynamic learning and pattern discerning capabilities of LLMs have the potential to exceed the efficacy of traditional static code analyzers, which depend on predetermined rules or patterns.

In a similar study, Khare et al. [Khare et al. 2023] investigate the effectiveness of ChatGPT 4 and two local models, CodeLlama-7B and CodeLlama-13B, in detecting security vulnerabilities in the code samples written in C/C++ and Java using different prompts. Their results indicate that while GPT-4 generally shows significant improvements over CodeLlama models, particularly with features like self-reflection, there isn't a noticeable difference between the two versions of CodeLlama itself. Moreover, they compare the effectiveness of ChatGPT 4 with a static analyzer called CodeQL. Their results showed that GPT-4 seems more versatile in detecting a broader range of vulnerabilities. On the contrary, we focused on evaluating the effectiveness of five popular online LLMs that developers and security analysts can easily use to identify vulnerabilities in real-world

[9] https://owasp.org/www-project-top-ten/

PHP code samples. Additional LLMs tested in this study have larger parameter sizes, which allows us to assess the vulnerability detection capabilities of more capable models.

In another LLM-based study, Cheshkovet et al. [Cheshkov et al. 2023] conducted a comparative analysis of various GPT-3 models with the aim to identify security vulnerabilities in Java code. This analysis involved testing the models with both vulnerable and remediated code sourced from GitHub. To gauge the precision and F1 scores of the models, both the vulnerable and patched codes were tested using ChatGPT and other GPT-3 models. The model `text-davinci-003` yielded precision, recall, and F1 score of 0.50, 0.99, and 0.67, respectively, while `gpt-3.5-turbo` resulted in scores of 0.51, 0.80, and 0.62, respectively. They concluded that both models under-performed compared to a basic classifier and faced difficulties in accurately pinpointing vulnerabilities. While these models may excel in various programming tasks, such as resolving challenges, they display considerable deficiencies in identifying security vulnerabilities in code. Consequently, they advocated for more extensive research to enhance the efficacy of these models and have proffered recommendations for ensuing studies. The main difference between our study and theirs centers on the composition of the datasets and the models used.

For instance, Cheshkovet et al. used some models such as `text-davinci-003`, which is currently not used for reasoning but only for Natural Language Processing (NLP) related tasks. Moreover, while they used code pieces from GitHub, we sampled codes that can be compiled and run. Maybe in some cases, this could also help LLMs to better understand the undelying code.

Another study investigating the use of the `gpt-3.5-turbo` model in static application security testing is made by Bakhshandeh et al. [Bakhshandeh et al. 2023]. They used `gpt-3.5-turbo`, Bandit[10] , Semgrep[11] and SonarQube[12] for finding security bugs in Python code. They conducted four different test cases, where the first was to get an answer about whether there was any security vulnerability in the given code and, if so, the line of code from which the vulnerability originated. This case's results were evaluated with a binary classification. While this case is similar to the first part of our question, the evaluation method is slightly different. In our study, when evaluating the responses of LLMs, we considered the descriptive nature of their responses. This can lead to a failure to perform other tasks such as in identifying the vulnerability and describing the existence of the vulnerability (for instance, asking for the CWE of the identified vulnerability, the variable it originated from, or the line of code it contained). Therefore, unlike the evaluation of the aforementioned study, in our study we used three different labels instead of binary classification. Furthermore, we did not count it as a total failure if the LLM correctly identified the vulnerability but did not provide the requested additional findings.

For this reason, the criteria of a binary classification should be clearer. In the second and third test cases, Bakhshandeh et al. gave the model helpful information such as the CWE list and asked which vulnerability was found in the given code. Although this is a good method – to use the model as a complementary assistant for Static Application Security Testing (SAST) tools – the question asked was not neutral and might lead to forcing the model to find a vulnerability from the given list. In addition, giving an overly specific instruction to the model (such as "only answer with JSON") might cause the focus of the model to be dispersed, considering that OpenAI especially has created an endpoint called "function calling" for this functionality. The prompt used in the fourth

[10] https://bandit.readthedocs.io/en/latest/

[11] https://github.com/semgrep/semgrep

[12] https://www.sonarsource.com/products/sonarqube/

case of their study is the most similar to our prompt; they asked the model to determine whether there is a security vulnerability and ask for its corresponding CWE with the vulnerable line. We have also asked LLM to determine if there is a vulnerability in the given source code and if so, to provide the corresponding CWE number. According to their results, while `gpt-3.5-turbo` had a lower F1 score than the Semgrep SAST tool in the first two cases, the F1 scores in the experiments in which the model was used as an assistant (and fed with additional information such as all the labels returned from static code analyzers for the code) were higher than other tools. They pointed out that they used only one model in their study and it would be beneficial to try others. In this respect, we believe that our study fills this gap nicely by covering other models as well.

A study by Li et al. [Li et al. 2023] compared the performance of GPT-4, GPT-3.5, Bard, and Claude 2. Their research predominantly centers around non-security-related concerns and specifically examines a system featuring an LLM component tasked with analyzing the outputs of a specialized tool named UBITect. UBITect [Zhai et al. 2020] is a solution for the Use Before Initialization (UBI) bugs, which are serious vulnerabilities in the Linux kernel, potentially causing information leakage and privilege escalation. Although their investigation did not extend to other security-related aspects, the insights they found in relation to the actual bug detection capabilities of the four popular LLMs showed a considerable degree of congruence with our own results. They made scans using a subset of Bug-50 and showed that GPT-4 is well ahead of the three other models. Although ChatGPT-3.5 is also the second best among the models, Claude and Bard showed very similar results – unlike what we found in our study, in which Claude and Bard performed poorly. This could be caused by the domain difference between studies.

While the study made by Ziems et al. [Ziems and Wu 2021] did not cover any of the models that we evaluated in the current work, they used Bidirectional Encoder Representations from Transformers (BERT) [Devlin et al. 2018] and Long Short-Term Memory (LSTM) [Hochreiter and Schmidhuber 1997] NLP models. For their study, they created a C/C++ code database consisting of software vulnerabilities. The best one of their own deep learning models shows a 93% success.

Various studies have been conducted to examine the efficacy of static analysis tools in the software development process. For example, Mahmood et al. [Mahmood and Mahmoud 2018] emphasized that reliance on static analysis tools may be misplaced, as they don't consistently highlight all the potential issues within the code. They conducted tests utilizing tools such as Yasca[13], PMD[14], LAPSE+[15], and FindBugs[16] and discovered that while these tools are capable of identifying some security vulnerabilities, they are far from foolproof.

Walker et al. [Walker et al. 2020] argued that these tools need to evolve to accommodate the shifting methodologies employed in software development. They observed that a significant number of these tools are deficient in offering a centralized overview and support for multiple repositories – features that would be highly beneficial. Such enhancements would enable developers to obtain comprehensive insights swiftly via a user-friendly dashboard, thereby promoting efficient management and monitoring of software vulnerabilities.

In an another study, Gomes et al. [Gomes et al. 2009] reported that static code analyzers tend to produce a substantial volume of false positive warnings. This could cause true positive warnings to be obscured amidst these false positives. This issue made

[13] https://scovetta.github.io/yasca/

[14] https://pmd.github.io/

[15] https://wiki.owasp.org/index.php/OWASP_LAPSE_Project

[16] https://github.com/findbugsproject/findbugs

the task of identifying genuine bugs in the code more complicated, turning it into a frustrating exercise, and potentially deterring the utilization of such tools. Additionally, they highlighted the importance of improving the readability and informativeness of warning reports to ensure they can be efficiently associated with the pertinent code alterations.

Another study that focused on finding Java and C++ vulnerabilities with static code analysis tools was made by Perhinschi [Perhinschi 2015]. The author stated that the results of this type of tools are highly correlated with the type of vulnerability, and these tools are not good at giving the corresponding CWE. We agree with Perhinschi's findings that static code analysis tools are insufficient and additionally we believe these tools will be replaced with or will be used as a complement to LLMs in the near future.

The study by Lebanidze [Lebanidze 2008] focused on the comparison of static code analyzer's generations. The author mentioned the reasons why static code analyzers have low detection rates, and described the next-generation tools, especially how these next generations tools should be; today we see that LLMs offer the right features to meet most of the improvements that the author had in mind at that time.

Since PHP is one of the most used programming languages for web development, there are other studies that focus on detecting PHP vulnerabilities using static code analysis tools. One of these studies was reported by Da Fonseca et al. [da Fonseca and Vieira 2014], who used two static analysis tools (RIPS[17] and phpSAFE[18]) to scan 35 WordPress plugins in order find any security vulnerabilities. They indicated that they found security issues in the scanned codes; however, we believe their performance could have been improved by the parallel usage of LLMs.

According to the study by Amankwah et al. [Amankwah et al. 2017], a perfect static code analyzer does not exist. This is why the community needs to use a number of them, in the hope that by complementing each other, we will get higher success rates. While the study was conducted well before the emergence of LLMs, their key message (that a perfect static code analyzer does not exist) still holds to this day.

In summary, most of the related work agree that current static code analysis tools are insufficient, and LLM-based static code analysis can potentially improve the state of the art further. As such, the study we present in this paper took this complementary approach, as discussed in the next section.

3 Methodology

In this section, we outline the methods we used to prepare and collect our dataset (including vulnerable and patched code snippets examined in the study), and provide a detailed overview of the LLMs assessed, along with the prompt selection. We focused our evaluation on the accuracy of the tools under investigation, given that we had sound ground truths (that we developed ourselves, as part of previous research) in our data regarding both vulnerabilities and false positives. That made our measurement somewhat easier than in the most common cases in the literature, where the true number of vulnerabilities in a piece of code is unknown and can only be estimated.

3.1 Data Set

We used 104 vulnerable code snippets and 97 patched ones to evaluate the effectiveness of selected LLMs in terms of vulnerability identification. The 104 samples of vulnerable

[17] https://github.com/robocoder/rips-scanner

[18] https://github.com/JoseCarlosFonseca/phpSAFE

code were taken from an earlier study [Ozturk et al. 2023]. These cover 51 different CWEs, which can be mapped onto 10 different OWASP Top-10 2021 categories.

For the patched codes, the 97 we used cover 10 different CWEs which mapped onto 4 different OWASP Top-10 2021 categories. 86 of these were randomly selected from 2 datasets in the Software Assurance Reference Dataset (SARD) Suites by The National Institute of Standards and Technology (NIST)[19,20]. We changed variable names and some of the predefined values in the samples taken from SARD to prevent LLMs from getting any extra valuable information, because at times those had meaningful names related to the patched vulnerability. The remaining 11 patched code samples were gathered from a similar research paper [Ozturk et al. 2023]. We verified that these code samples did not leak any information about the patch or the original vulnerability. The collection of the mentioned datasets used in the research[21] is shared openly with the academic community.

3.2 Large Language Models (LLMs)

In this study, we selected five popular LLM-based chatbots: ChatGPT-4, ChatGPT-3.5, Claude, Llama-2 and Bard[22]. These LLMs can be accessed conveniently through their web interface. Our goal is to evaluate their efficacy in identifying vulnerabilities within the source code. In the following section, we delve deeper into these tools.

We had previous experience on prompt design, and used it to give to the LLM precise context and questions. We, however, do not claim that our prompt selection was optimal in any way and there is likely room to find better prompts in future works. In addition, we decided to use the same prompt for all the LLMs, which in a way ensures fairness in the analysis but on the other hand possibly leaves ample room for improvement, as different prompts targeting each particular LLM would very likely lead to at least slightly better results. We found, however, that this was a good balance in terms of obtaining a fair comparison and simplicity/reproducibility, so we went ahead with it.

3.2.1 ChatGPT4 & ChatGPT-3.5

ChatGPT, developed by OpenAI, is an AI-driven chatbot that was first made available to the public in 2022. As its acclaim grows steadily, the software engineering community has begun utilizing the tool for coding and troubleshooting bugs. In March 2023, OpenAI introduced ChatGPT-4, an enhanced version of ChatGPT-3.5. To use GPT-4, users must sign up for ChatGPT Plus, a $20 monthly subscription that offers premium access to the site. During the code scanning process, we used both the ChatGPT-4 and ChatGPT-3.5 August 2023 version. All the scanning was accomplished through their web-interface.

Though not explicitly designed for cybersecurity, ChatGPT possesses the ability to answer complex queries and furnish details on cybersecurity and associated domains. This feature positions ChatGPT as a beneficial tool for individuals keen on understanding cybersecurity, as well as for researchers and professionals aiming to automate tasks in the realm of cybersecurity.

With its advanced NLP capabilities, ChatGPT can produce coherent and at times even enlightening content across a broad spectrum of cybersecurity subjects, ranging

[19] https://samate.nist.gov/SARD/test-suites/103

[20] https://samate.nist.gov/SARD/test-suites/114

[21] https://github.com/StaticCodeAnalysiswithLLMs

[22] Since 8 February 2024, Bard is known as Gemini – see https://en.wikipedia.org/wiki/Gemini_(chatbot). For consistency, we continue to refer to this LLM as Bard throughout the paper.

from fundamental computer security principles to more intricate topics. Moreover, its proficiency in comprehending natural language queries enables it to seamlessly engage in a dialogue, establishing it as an approachable and intuitive tool for those keen on delving into cybersecurity. On a less sunny note, ChatGPT is also known to have been used to write very convincing phishing emails [Sharma et al. 2023].

3.2.2 Claude 2

Claude 2 is an AI chatbot based on Anthropic's investigations into crafting supportive AI mechanisms[23]. It is an updated version of its ancestor Claude and was released on Jul 11, 2023. The chatbot is available through both a web interface and an API. Currently, its services can be accessed only from the US and UK. The company offers a $20 subscription plan for availability privileges such as priority during heavy traffic times, and the ability to ask more questions. Claude 2 demonstrates proficiency in a broad range of conversational and text manipulation activities. One of the features that distinguish its model from other chatbots is that it can accept up to one hundred thousand tokens, which corresponds to approximately seventy-two thousand words as input. Although Claude is not specifically designed for cybersecurity, as is the case with OpenAI's models, the company states that coding is among its many capabilities. Chen et al. also boast that Claude 2 got a score of 71.2% in the Codex HumanEval Python programming test [Chen et al. 2021].

3.2.3 Bard

Bard[24] is an experimental conversational AI chat offering from Google. The basic model of the chatbot is the Pathways Language Model (PaLM 2). Although its functionality mirrors that of other LLMs, its key difference is that it can (without any additional plugin) retrieve information from the web, allowing it to access up-to-date data and incorporate it into its answers. Since the model arranges the data collected from the internet and facilitates access to information for its users, it provides a great opportunity for individuals who want to learn in the field of cybersecurity. On the model's frequently asked questions page, Google states that Bard can help its users with coding, but also mentions that it is still experimental. The model does not have any paid plans, and at the time of writing it is free for anyone to use.

3.2.4 Llama-2

Meta's Llama-2 is offered in a variety of models, named 7B, 13B and 70B, depending on the parameter size of each[25]. Every model has been released with a fine-tuned version for conversational dialogue. In this study, Llama-2 refers to the Llama-2-70B-chat model, which is a 70 billion parameter model. The company stated that the fine tuning was made with publicly available datasets and more than a million human annotations. The model is open source and free, even for commercial use. At the time of starting this work, Llama-2 was the most current and most powerful model released by Meta.

[23] https://www.anthropic.com/index/claude-2

[24] https://bard.google.com/, which will automatically redirect to https://gemini.google.com/

[25] https://ai.meta.com/llama/

3.3 Prompt Selection

Prompting plays a crucial role while interacting with LLMs. A prompt that is not prepared properly (such as a non-neutral or leading prompt) may cause the model to give heavily biased answers. For example, a simplistic prompt like *"What is the vulnerability in this code?"* may lead the model to believe that there is a vulnerability. Asking instead a more neutral question such as *"Is there a vulnerability in this code?"* offers better results. For this reason, the first criteria for our prompt selection was to produce a clear prompt which does not lead to any biases.

Secondly, since the study was conducted on five different LLMs, the prompt should be simple enough for all models to handle it with ease. In line with the above criteria, we came to the following prompt: *"You are a cybersecurity specialist who is analysing the given PHP code. Are there any security vulnerabilities in the given PHP code? If there are any vulnerabilities give the corresponding CWE."*. The prompt first specifies a role for the LLM, and by doing this, we aimed to increase its level domain awareness.

4 Execution of the Experiment

This section provides an overview of the procedure we followed for executing our experiment, as well as the evaluation criteria employed for ensuring the suitability of the output produced from the experiment.

4.1 Study Procedure

This section provides further explanations of the procedure we followed in this study, to aid in any replication work. First of all, we decided which LLMs would be used within the scope of our research. The study was conducted on OpenAI's ChatGPT-3.5 and ChatGPT-4, Google's Bard, Antropic's Claude and Meta's Llama-2. The selection of these models was influenced by the fact that they offer easy access via a web browser, ease of use through a usable interface and all have very strong conversational capabilities. After the selection of LLMs, we searched for datasets in PHP and then selected random samples from the suitable ones. These was followed by the preparation of selected samples as described in Section 3.1. After this stage was completed, the scan process was carried out in August 2023.

Through these scans, we accessed ChatGPT from its official webpage and we used "ChatGPT August 3 Version". We accessed Bard through its official page and we used its "2023.07.13" version. We worked from the official page of Claude, and scans were made using Claude 2. Lastly, we accessed the Llama-2 70B model through the company Replicate[26] with unknown version and update. It is important to point out that Bard sometimes creates different draft answers for a single question. Since all other LLMs produce only one answer for a given question, only the first draft of Bard was taken into account during our study. This approach was taken to keep the consistency regarding the number of responses for every single code among all LLMs.

Finally, it should be noted that answers from LLMs tend to be non-deterministic (i.e. there will be some slight differences in the answer when asking the same question the second time). However, in the preliminary dry run of our experiment, we found that the differences were minimal. As such, it was decided that (in the interest of time and resources), we only executed each question once. In the future, it is possible to run more elaborate experiments whereby we collect answers from LLMs multiple times, and compare these answers in detail, as outlined further in Section 8.

[26] https://www.llama2.ai/

4.2 Output Evaluation

This section provides understanding of our output evaluation procedure. The most distinctive feature of Chatbots built with LLMs is that they have complex conversational capabilities. Due to this conversational structure we first analyze the given outputs in a qualitative manner to came out with different labels. After obtaining labels from qualitative evaluation, we made a quantitative evaluation to compare the effectiveness of the different models.

For the evaluation of the dataset with vulnerable code, after many trials we came to the conclusion that it would be better to work with 3 different labels. These are ’found’, ’partially found’ and ’not found’. In order for an output to be labeled as ’found’, the LLM must correctly point out the vulnerability, describe it correctly, and provide the corresponding CWE title together with its number. That is much stricter than what other authors, and even ourselves, have considered in other studies. In the case that the LLM accurately explains the vulnerability but provides an incorrect CWE number, an erroneous CWE title, an entirely unrelated CWE, or it failed to offer a CWE at all, the output was evaluated as ’partially found’. The ’not found’ label was reserved for cases where the LLM completely fails to find the vulnerability in the given code.

For the evaluation of the patched code dataset, we followed the same methodology as with the vulnerable dataset. We first analyzed the outputs qualitatively, then we labelled them and evaluate quantitatively. As a result of this analysis, patched code outputs were just labeled as correct and not correct. An output is labeled as ’correct’ in the following cases: (1) the model correctly interprets the code and does not point out the patched vulnerability; (2) the model points out that, while the patching method is secure in the examined code, it might be susceptible to vulnerabilities if used in a different context; and (3) the model identifies that the mitigation used is not the best practice and proposes better alternatives. The ’not correct’ labeled is used in the following cases: (1) if the model cannot detect that patching was implemented in the code under assessment and still considers the code to be vulnerable.(2) if the model completely misinterprets the given code and generates a false positive. Examples of this situation include perceiving non-user input variables or hard-coded values as user input, evaluating the purpose of built-in php or used package functions erroneously, and misinterpreting the regex patterns used.(3) if the model evaluates a content that does not exist. There are studies in the literature – such as [Athaluri et al. 2023] – that identify such cases as “hallucination”. In this case, the model somehow evaluates code that does not exist in the given input sample. A typical example occurs when a model claims there is an SQL injection vulnerability, despite the code provided to the model containing no SQL queries or any form of database library usage.

5 Results

In this section, we evaluated the effectiveness of the studied LLMs’ ability to identify code vulnerabilities. We first compared the effectiveness of LLMs in identifying vulnerabilities and then we explored their results when used in pairs. Moreover, we investigated in greater detail the types of vulnerabilities found, by these LLMs and their false-positive rates.

5.1 Vulnerability detection efficacy of LLMs

This section describes the vulnerability detection statistics of the selected LLMs. As mentioned in Section 3.1, the vulnerable code dataset consists of 104 vulnerabilities which cover 51 different CWEs that are mapped to 10 categories of the 2021 OWASP Top-10.

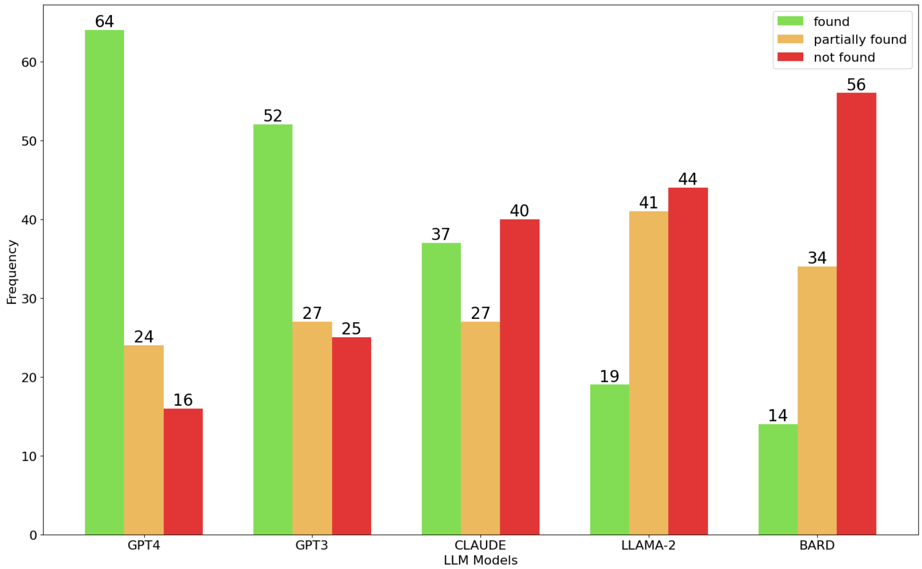

Figure 1: Statistical summary of vulnerability detection per LLM (the absolute number of vulnerabilities found, not found, or partially found is shown on top of each bar)

Figure 1 highlights the vulnerability detection performance of the LLMs used studied in this work. Among these, OpenAI's ChatGPT-4 showed the best performance by some margin. This was followed by ChatGPT-3.5, also developed by OpenAI. After these models, Antropic's Claude took the third spot. In the fourth and fifth places, respectively, we find Meta's Llama-2 70-B model and Google's Bard.

Table 1 shows the vulnerability detection percentages of the studied LLMs. ChatGPT-4, the most successful model, accurately identified 64 out of 104 vulnerabilities, reaching a success rate of 61.53%. Of these, 24 (23.07%) vulnerabilities were only partially recognized. While the model correctly explained the vulnerabilities, it made mistakes in determining the correct CWE number and/or title. The model completely failed to detect 16 vulnerabilities, leading to a failure rate of 15.38%. ChatGPT-3.5, which ranks second in terms of overall vulnerability detection performance, was able to identify 52 of the 104 vulnerabilities in the dataset, showing a performance of 50%. 27 vulnerabilities were partially identified which accounts for 25.96% of the total. ChatGPT-3.5 failed to correctly label 22 code samples, and for 5 samples no labels were assigned. The model failed to find 25 vulnerabilities or 24.03%.The third place in the ranking belongs to Claude, the model is the first LLM that produced a higher failure than correct detection rate. It was able to detect 37 out of 104 vulnerabilities successfully, meanwhile it missed 40 vulnerabilities. The model was followed by Llama-2 and Bard respectively. Llama-2 identified 19 vulnerabilities while it missed 44 of them, identifying 41 vulnerabilities only partially. Bard, which ranked last, had 14 full detection, 34 partial detection and 56 misses.

Model	Found	Partially Found	Not Found
ChatGPT-4	64 (61,53%)	24 (23,07%)	16 (15,38%)
ChatGPT-3.5	52 (50,00%)	27 (25,96%)	25 (24,03%)
Claude	37 (35,57%)	27 (25,26%)	40 (38,46%)
Llama-2	19 (18,26%)	41 (39,42%)	44 (42,30%)
Bard	14 (13,46%)	34 (32,69%)	56 (53,84%)

Table 1: Vulnerability detection percentages of the LLMs under study

5.2 Impact of Vulnerability Type

We decided to study the performance of the models according to the type of vulnerability assessed. It stands to reason that some vulnerabilities, perhaps those that have a more rule-based nature, could be easier to find for the LLMs. Examples of those maybe the cryptographic weakness that occurs when a deprecated or insufficiently strong algorithm is used. Or a SQL injection issue that occurs typically when user input is passed directly to the database interpreter without any sanitization.

On the other hand, it seems plausible that other vulnerabilities types require more and deeper reasoning for their detection and hence constitute a bigger challenge for the LLMs under scrutiny. One such type may be the *insecure design vulnerability* that can be caused by a lack of checks allowing an attacker to change a product's price to a negative value. To gain a clearer insight into how these types of vulnerability affect detection rates, we linked the vulnerabilities to their corresponding CWEs and then aligned them with the 2021 OWASP Top-10 web-based vulnerabilities.

Table 2 displays the number of vulnerabilities detected by each LLM, broken down by OWASP top-10 category. As seen in the table, most of the vulnerabilities in our study were cryptographic failures, followed by injections and broken access control categories. ChatGPT-4 managed to achieved the highest detection rate in 9 out of 10 categories. Surprisingly, for cryptographic failures, ChatGPT-3.5 slightly outperformed ChatGPT-4 by 5%. Usually, ChatGPT-3.5 ranked second highest in vulnerability detection rate. This shows that the GPT based models have both better overall performance and type coverage.

After ChatGPT-3.5, Claude and LLama-2 came next. In every category, Claude consistently achieved a higher vulnerability detection rate than LLama-2. In the categories of security misconfiguration and vulnerable and outdated components, there were only 3 vulnerabilities. Yet, Claude detected all of them, while LLama-2 failed to identify any, resulting in a worrying score of 0%. Similarly, LLama-2 failed to detect any of the vulnerabilities belonging to 4 other categories. Lastly, Bard only detected vulnerabilities related to injection, cryptographic failures and identification & authentication categories. In the end, Bard had the worse overall vulnerability detection rate and coverage.

Each selected LLM was able to identify at least one vulnerability in the cryptographic failures and injection categories. This could be attributed to their widespread presence in web applications. Typically, less well-known and design-related vulnerabilities were harder to identify for LLMs. For example, both GPT-based models reported the lowest detection rates in the security logging and monitoring failures, and on the identification and authentication failures categories. The security logging and monitoring failures category had 3 vulnerabilities. ChatGPT-4 and Claude detected only one, while other LLMs, including ChatGPT-3.5, failed to identify any. This might be because they have very little representation in the source code.

	Tot.	ChatGPT-4	ChatGPT-3.5	Claude	Llama-2	Bard
Broken Access Control	12	8 (66.67%)	6 (50.00%)	4 (33.33%)	3 (25.00%)	0 (0.00%)
Cryptographic Failures	34	16 (47.06%)	18 (52.94%)	6 (17.65%)	4 (11.76%)	1 (2.94%)
Injection	32	25 (78.12%)	21 (65.62%)	16 (50.00%)	11 (34.38%)	11 (34.38%)
Insecure design	8	4 (50.00%)	2 (25.00%)	4 (50.00%)	1 (12.50%)	0 (0.00%)
Security Misconfiguration	2	2 (100.00%)	1 (50.00%)	2 (100.00%)	0 (0.00%)	0 (0.00%)
Vulnerable & Outdated Components	1	1 (100.00%)	0 (0.00%)	1 (100.00%)	0 (0.00%)	0 (0.00%)
Identification & Auth Failures	5	2 (40.00%)	1 (20.00%)	1 (20.00%)	0 (0.00%)	2 (40.00%)
Software & Data Integrity Failures	2	1 (50.00%)	1 (50.00%)	1 (50.00%)	0 (0.00%)	0 (0.00%)
Security Logging & Monitoring Failures	3	1 (33.33%)	0 (0.00%)	1 (33.33%)	0 (0.00%)	0 (0.00%)
SSRF	5	4 (80.00%)	2 (40.00%)	1 (20.00%)	0 (0.00%)	0 (0.00%)

Table 2: Number of vulnerabilities found by OWASP Top 10 category

5.3 Efficacy of Paired Combinations

While the primary focus of our study was to evaluate the effectiveness of individual LLMs in identifying vulnerabilities in code samples, our observations revealed an intriguing potential. The correct vulnerability identification of LLMs on different code samples led us to the idea that these models can complement each other when used together. Therefore, we were interested in exploring the performance of these LLMs in dual combinations and identifying the benefits of their combined use in terms of vulnerability detection.

This section introduces the performance of the dual combinations of LLMs. Figure 2 highlights the vulnerability detection performance of these combinations. The dual combination of ChatGPT-4 and ChatGPT-3.5 has become the most successful one and surpassed the performance of ChatGPT-4 alone by performing 75 correct detections, 17 partial detections and 12 failures. In terms of percentage, the dual combination improves the vulnerability detection rate by showing a 10.57% increase in full detection, a 6.73% decrease in partial detection and a 3.84% decrease in failures compared to the (best) individual performance of ChatGPT-4.

The role of ChatGPT-3.5 in achieving this success is highlighted by its ability to detect 10 codes within the cryptographic failure category that were only partially catched by ChatGPT-4, and its identification of one code within the injection category that was missed by ChatGPT-4. Furthermore, ChatGPT-3.5 managed to partially detect two more codes from the cryptographic failure category and one from the injection category, all of which were overlooked by ChatGPT-4.

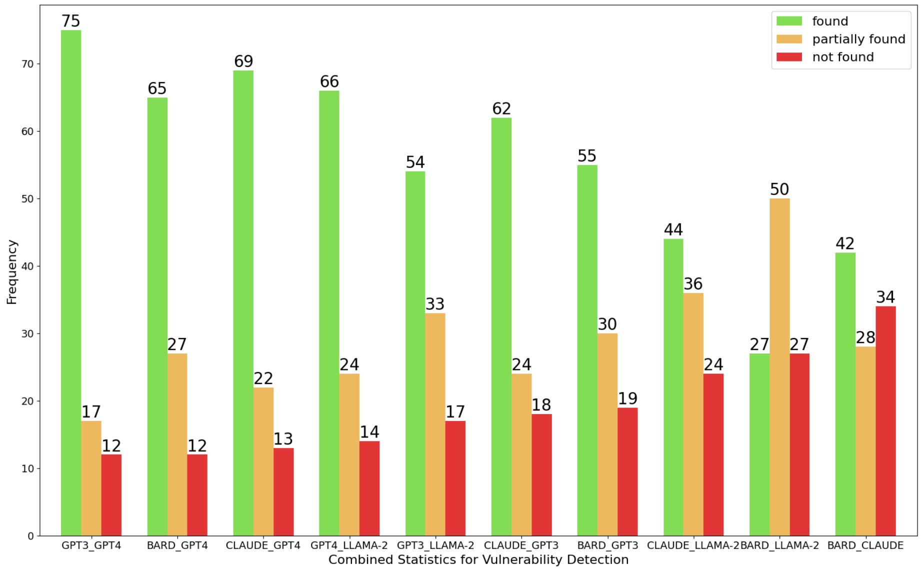

Figure 2: Vulnerability Detection Summary for Paired LLM Combinations

In the dual combination ranking, the second, third and fourth places are for the combinations of Claude, Llama-2 and Bard with ChatGPT-4, respectively. The combination of Claude and ChatGPT-4 showed improved performance compared to the individual performance of ChatGPT-4, with a 4.80% increase in full detection. The combination of Llama-2 and ChatGPT-4 also showed improved performance compared to ChatGPT-4 alone, with a 1.92% increase in full detection and a 1.92% decrease in failure. The last combination involving ChatGPT-4 was with Bard, which came in fourth place. Compared to the individual performance of ChatGPT-4, it showed marginally improved performance with an increase of 0.96% in full detection, a 2.88% in partial detection and a 3.74% reduction in failure.

After the first four duos we have mentioned so far, the Claude and ChatGPT-3.5 duo, which ranks fifth, is the first one that cannot match the individual detection performance of ChatGPT-4 in solo. This duo was followed by Bard's and Claude's duos with ChatGPT-3.5, which took sixth and seventh place, respectively.

In each of the duos so far, first ChatGPT-4 and then ChatGPT-3.5 were crucial, in other words, OpenAI's models were included in the first seven out of nine pairs in terms of performance. The last 3 pairings are Claude - Llama-2, Bard - Claude and Bard - Llama-2, respectively and the ranking is interestingly directly proportional to the individual performances.

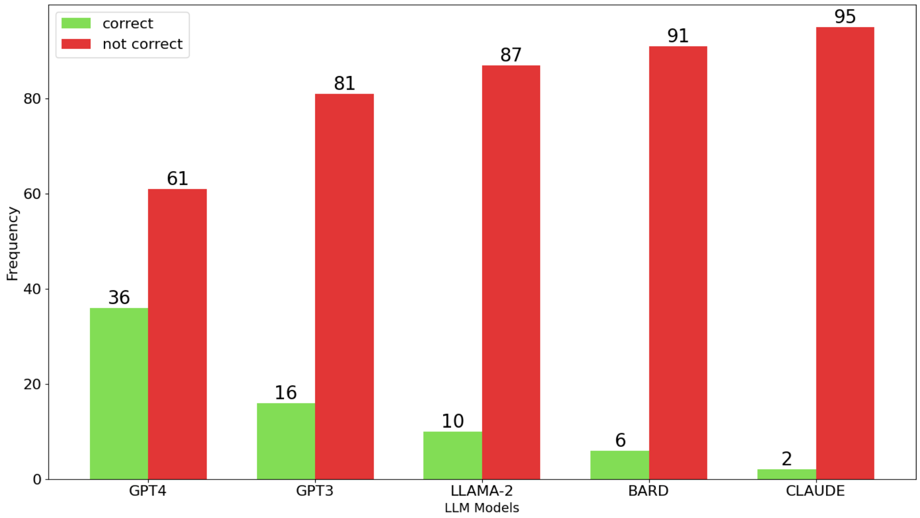

Figure 3: Statistical summary of correct and incorrect answers over patched codes, per LLM

5.4 Individual Performance of LLMs on the Patched Code Dataset

As we mentioned in Section 4.1, we conducted our study over a patched code dataset to assess the false-positive rate of LLMs. In this section, we present the statistics of each model. The results, shown in Figure 3, are presented according to the evaluation metrics described in Section 4.2. All LLMs showed incorrect results on more than half of the patched codes. As with vulnerability detection, the two most successful models in patched code scans were ChatGPT-4 and ChatGPT-3.5, respectively. While ChatGPT-4 classified 36 of 97 scans correctly, ChatGPT-3.5 only identify 16 correctly. Meanwhile Llama-2 took the third place by producing 10 correct identifications.

In the last two places, Bard is fourth with 6 correct and 91 incorrect answers, and Claude is fifth with 2 correct and 95 incorrect answers.

We observed high false positive rates. Even the best performing model had nearly a 63% false positive rate. Conversely, some models significantly under-performed, realizing false positive rates of up to 97%. It appears that high false positive rates constitute a major issue when employing LLMs as static code analyzers. This is interesting but hardly surprising as the same problem of too many false positives is typical also with classical static code analysers.

6 Discussion

In this section, we discuss the key implications of our study, especially regarding the feasibility of using LLMs to replace or complement static analysis tools for finding vulnerabilities in software, as well as the possible unintended consequences of using LLMs when it comes to the privacy of the data involved.

6.1 Feasibility of static analysis with LLMs

The results we obtained from this study show that ChatGPT-4 takes the lead in vulnerability detection from source code analysis. While ChatGPT-3.5 didn't yield results as closely aligned with ChatGPT-4, the disparity between it and the subsequent model, Claude, was more pronounced than the disparity between it and the preceding model. Even though ChatGPT-3.5 has the same number of partially found detection as Claude, there is a difference of approximately 15% in full detection between them, and this is a significant difference. The performances of the Llama-2 and Bard models, which exhibited failure rates of 42.30% and 53.84% respectively, reveal that they are far behind their competitors in finding vulnerabilities.

When considering the analysis of patched code, it is unfortunate that all the LLMs yield incorrect answers for more than 60% of the time. The reality that even the most proficient model only achieves a success rate of 37.11% illustrates that there is substantial room for enhancement in the existing models to reduce the generation of false positives.

The model that surprised us the most in the patch code analysis was Claude. While it ranked third in vulnerability detection, it became the last in this ranking. This situation created the perception that the model was actually always trying to find a vulnerability. Even though it is good to have a high detection ability, it is disappointing to produce that many false positives.

Discussing Bard, the model does present a unique case. Out of 91 responses deemed ”incorrect”, 26 occurred because Bard refused to assist with the desired task. Although this might have stemmed from stringent security measures, the remaining 65 inaccuracies are not to be overlooked or minimized, indicating areas where refinement and improvement are necessary.

Llama-2, on the other hand, ranked third with 6 more ”incorrect” numbers than ChatGPT-3.5. It should also be noted that although ChatGPT-3.5 ranks second in both vulnerable code and patched code scans, its success in the patched code scan is significantly lower. While the difference between ChatGPT-4 and ChatGPT-3.5 in vulnerability detection is 11.53%, this grows to 20.7% over the patched dataset. Considering what we have seen, the model for which static analysis is most feasible seems to be ChatGPT-4.

The practicality of conducting static analysis using LLMs is also affected by their usability and the absence of API support. In our study, code samples were manually inputted into the LLMs, a method that proves impractical for real-world applications. For projects with extensive code bases and multiple vulnerabilities, an automated approach to submitting code, achievable only through API integration, is essential. ChatGPT was the only model that offered API support during our data collection phase. Future studies could concentrate on developing methods to detect vulnerabilities within extensive code samples that contain numerous security vulnerabilities.

6.2 Implications on Privacy

Data is a very important element for the performance of LLMs and data privacy is an issue that needs to be discussed, especially in the cases where proprietary code is input into LLMs for analysis.

The increased popularity of LLM-based chatbots has also led to a rise in attacks against them. An example of this occurred in March 2023[27]. A vulnerable component used in ChatGPT's system ended up in a data leak in which users were able to see other's chats and payment information. For other chatbots, there is no publicly announced hacking

[27] https://www.cshub.com/data/news/openai-confirms-chatgpt-data-breach

news. However, the attacks are not limited only on a company basis, approximately 100 thousand ChatGPT accounts have been compromised because of malicious software such as info stealers and put up for sale on the dark web [Lakshmanan 2023], which shows that cybercriminals have a real interest in this area.

Due to these reasons, sharing sensitive information such as project code is a real concern. Although users can reduce the risks by following the security measures recommended by the companies behind these LLMs, these solutions are generally aimed to protect Personally Identifiable Information (PII) such as addresses, phone numbers, etc. or Sensitive Personally Identifiable Information (SPII) such as payment information.

We wanted to add some suggestions for improving the privacy of codes submitted to the LLMs. Developers can break down their code into smaller, discrete snippets. This makes it challenging for potential adversaries to understand the general functionality of the entire codebase. Secondly, slight obfuscation techniques such as changing the names of code variables to others that do not reveal the purpose in your project, might be of help. Although this may reduce the performance of the model, it can make you safer in the case of any data leak. Lastly you may use different chat sessions and user IDs for every scan, by doing this your code snippets belonging to the same project will not be easily gathered on the same session and it may make it much harder to understand the overall structure of your entire project.

Apart from data privacy concerns related to attacks, individuals or companies may be concerned about their data being used to improve the models. The companies behind the models used within the study have very different policies regarding data privacy.

On 25 April 2023, OpenAI started to offer its users the feature of not saving their conversation history[28]. By default, the company uses customers' chats to further develop its models, but users can now ensure their conversations are not recorded by making the necessary adjustments in the settings. Despite this, it should also be noted that OpenAI states that even if you use the mentioned settings, they will keep your conversations for 30 days to investigate potential abuse cases.

Unlike OpenAI, Antropic's Claude does not use users' conversations by default[29]. Only in two cases, the application employs user prompts and their corresponding responses to further develop its models. One of these situations is when the user gives feedback through the thumbs up/down buttons that come with the answer or by contacting the company itself, the answers will be used by the company. The second case aims to provide a more secure service; if a prompt and conversation are flagged for trust and safety review, the company uses the user's conversation to improve its models, including their Trust and Safety classifiers. In addition to the use of users' data for model development, the conversations are retained for a while and automatically deleted within 90 days, but if a user prompt is flagged for security reasons the conversation can be saved for up to 2 years[30]. Other than that, the company states that any data provided to the application is stored for as long as reasonably necessary for the purposes and criteria specified in the company's Privacy Policy[31].

By default, Google keeps users' chats with Bard, up to 18 months and then automatically deletes them, users can change this period from 3 to 36 months or they can stop it completely. In case a user prefers not to save conversations, Google states that they will keep them, up to 72 hours to "provide the service and process any feedback", as stated

[28] https://openai.com/blog/new-ways-to-manage-your-data-in-chatgpt

[29] https://support.anthropic.com/en/articles/8325621

[30] https://support.anthropic.com/en/articles/7996866-how-long-do-you-store-personal-data

[31] https://console.anthropic.com/legal/privacy

in their Privacy Hub website[32].

Since Llama-2 is an open-source project, users' data can remain with them if they install and use the model on their own computer, without the involvement of any third party such as Replicate, that we used in this work. For this reason, we can say that there are no privacy concerns while using the model and, in certain scenarios, this may be considered crucial and more important than its relatively mediocre performance on the tasks at hand.

7 Conclusion

In this paper, we evaluated five notable LLM chatbot models: ChatGPT 4, ChatGPT 3.5, Claude, Bard, and Llama-2, assessing their capabilities to find vulnerabilities using 104 vulnerable and 97 patched PHP code samples, spanning the Top-10 categories as defined by OWASP.

We found that there is a noticeable variation in the performance of the five LLMs in detecting vulnerabilities, with ChatGPT-4 outperforming the others significantly. ChatGPT-4 has a considerably higher detection rate of 61.53%, implying a higher reliability in identifying vulnerabilities. ChatGPT-3.5 and Claude occupy a middle ground with detection rates of 50% and 35.57% respectively. The low vulnerability detection rates in models like Bard (13.46%) and Llama-2 (18.26%) highlights the limitations and challenges these models still face. The presence of partially found vulnerabilities across all models, regardless of their overall performance in detecting vulnerabilities, indicates a common challenge faced by all LLMs in fully understanding and identifying vulnerabilities.

Furthermore, our analysis revealed that ChatGPT-4 and ChatGPT-3.5 consistently outperformed in identifying vulnerabilities across most categories, compared to their counterparts. Bard and Llama-2 exhibited limited efficacy across a broad range of categories.

Interestingly, high false positive rates were a prevalent finding across the LLMs under study. Despite the high false-positive rate, the potential of LLMs as static code analyzers should not be understated. LLMs, with their dynamic learning and pattern recognition capabilities, can easily exceed the efficiency of conventional static code analyzers. We are of the opinion that, with more fine-tuning, false positives can be reduced. Lastly, we found that employing a combination of two LLMs simultaneously for static analysis only yielded a slight improvement in vulnerability detection rates.

Apart from the experimental results, we believe that the method we used is interesting, particularly regarding having a ground truth dataset with real vulnerabilities manually included and checked, together with an in-depth analysis of the frequently forgotten false positives, thanks to another carefully crafted dataset of patched vulnerabilities. Other works in the future literature should consider following a similar approach to compare their results fairly. Sharing both datasets is also highly recommended for reproducibility.

8 Future Work

We suggest focusing on four particular areas for further research to expand upon the findings of this study. First, future research can investigate the impact of the non-deterministic nature of different LLMs in terms of detecting vulnerabilities in source code. LLMs

[32] https://support.google.com/gemini/answer/13594961

often yield varying responses to the same query when asked repeatedly. This could result in identifying vulnerabilities that were not found in the first place. On the other hand, varying outcomes may also mislead developers and lead to confusion. Future studies should aim to devise a solution that achieves an optimal balance.

Second, for each LLM, prompt optimization can be done separately. This can be useful for exploring which prompts will give better outputs for LLMs that belong to various model classes and possess distinct architectures. While doing this research, resources such as OpenAI's cookbook[33] can be used, and improvements can be made in finding security vulnerabilities.

Third, future studies can focus on developing strategies for establishing effective vulnerability scanning with LLMs for large code bases with multiple vulnerabilities. This would increase the usage of LLMs for detecting various vulnerabilities in a given code base.

Finally, fine-tuning and preparation of the datasets to be used in the fine-tuning process to address complex vulnerabilities will boost the effectiveness of future LLMs in detecting vulnerabilities. Separate research should be carried out to investigate the kind of dataset to use, as the quality of the used dataset in the fine-tuning process will greatly affect the performance after the process. Furthermore, different fine-tuning approaches could be focused on improving the detection capabilities of the LLMs.

Acknowledgements

We would like to thank Omer Ozturk for his helpful comments. Additionally, we extend our deepest gratitude to Steffen Wendzel for his invaluable assistance and guidance as the editor of this special issue. His expertise and support have been instrumental in the completion of this paper.

References

[Amankwah et al. 2017] Amankwah, R., Kudjo, PK., Antwi, SY.: "Evaluation of software vulnerability detection methods and tools: a review"; International Journal of Computer Applications, 169(8):22–27, 2017.

[Athaluri et al. 2023] Athaluri, S.A., Manthena, S.V., Kesapragada, V.K.M., Yarlagadda, V., Dave, T., Duddumpudi, R.T.S.: "Exploring the boundaries of reality: Investigating the phenomenon of artificial intelligence hallucination in scientific writing through chatgpt references"; Cureus 15(4), 2023.

[Bakhshandeh et al. 2023] Bakhshandeh, A., Keramatfar, A., Norouzi, A., Chekidehkhoun, M.M.: "Using chatgpt as a static application security testing tool"; arXiv preprint arXiv:2308.14434, 2023.

[Baset and Denning 2017] Baset, A.Z., Denning, T.: "Ide plugins for detecting input-validation vulnerabilities"; In 2017 IEEE Security and Privacy Workshops (SPW), pages 143–146. IEEE, 2017.

[Chen et al. 2021] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.D.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A.: "Evaluating large language models trained on code"; arXiv preprint arXiv:2107.03374, 2021.

[Cheshkov et al. 2023] Cheshkov, A., Zadorozhny, P., Levichev, R.: "Evaluation of chatgpt model for vulnerability detection"; arXiv preprint arXiv:2304.07232, 2023.

[33] https://cookbook.openai.com/

[da Fonseca and Vieira 2014] da Fonseca, J.C.C.M., Vieira, M.P.A.: "A practical experience on the impact of plugins in web security"; In 2014 IEEE 33rd International Symposium on Reliable Distributed Systems, pages 21–30. IEEE, 2014.

[Devlin et al. 2018] Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: "Bert: Pre-training of deep bidirectional transformers for language understanding"; arXiv preprint arXiv:1810.04805, 2018.

[Gomes et al. 2009] Gomes, I., Morgado, P., Gomes, T., Moreira, R.: "An overview on the static code analysis approach in software development"; Faculdade de Engenharia da Universidade do Porto, Portugal, 2009.

[Hochreiter and Schmidhuber 1997] Hochreiter, S., Schmidhuber, J.: "Long short-term memory"; Neural computation, 9(8), pages 1735–1780, 1997.

[Khare et al. 2023] Khare, A., Dutta, S., Li, Z., Solko-Breslin, A., Alur, R., Naik, M.: "Understanding the effectiveness of large language models in detecting security vulnerabilities"; arXiv preprint arXiv:2311.16169, 2023.

[Koch 2023] Koch, C.: "I used gpt-3 to find 213 security vulnerabilities in a single codebase"; Better Programming, https://betterprogramming.pub/i-used-gpt-3-to-find-213-security-vulnerabilities-in-a-single-codebase-cc3870ba9411, 2023.

[Lakshmanan 2023] Lakshmanan, R.: "Over 100,000 stolen chatgpt account credentials sold on dark web marketplaces"; The Hacker News, https://thehackernews.com/2023/06/over-100000-stolen-chatgpt-account.html, 2023.

[Lebanidze 2008] Lebanidze, E.: "The need for fourth generation static analysis tools for security–from bugs to flaws"; In Application Security Conference, 2008.

[Li et al. 2023] Li, H., Hao, Y., Zhai, Y., Qian, Z.: "The hitchhiker's guide to program analysis: A journey with large language models"; arXiv preprint arXiv:2308.00245, 2023.

[Mahmood and Mahmoud 2018] Mahmood, R., Mahmoud, Q.H.: "Evaluation of static analysis tools for finding vulnerabilities in java and c/c++ source code"; arXiv preprint arXiv:1805.09040, 2018.

[Merkow and Raghavan 2010] Merkow, M.S., Raghavan, L.: "Secure and resilient software development"; CRC Press, 2010.

[Ozturk et al. 2023] Ozturk, O.S., Ekmekcioglu, E., Cetin, O., Arief, B., Hernandez-Castro, J.: "New tricks to old codes: can ai chatbots replace static code analysis tools?"; In Proceedings of the 2023 European Interdisciplinary Cybersecurity Conference, pages 13–18, 2023.

[Perhinschi 2015] Perhinschi, A.M.: "Static Code Analysis: On Detection of Security Vulnerabilities and Classification of Warning Messages"; West Virginia University, 2015.

[Sharma et al. 2023] Sharma, M., Singh, K.,Aggarwal, P.,Dutt, V.: "How well does GPT phish people? An investigation involving cognitive biases and feedback"; In 2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), pages 451-457. IEEE, 2023.

[Telang and Wattal 2007] Telang, R., Wattal, S.: "An empirical analysis of the impact of software vulnerability announcements on firm stock price"; IEEE Transactions on Software engineering, 33(8):544–557, 2007.

[Walker et al. 2020] Walker, A., Coffey, M., Tisnovsky, P., Cerny, T.: "On limitations of modern static analysis tools"; In Information Science and Applications, pages 577–586. Springer, 2020.

[Zhai et al. 2020] Zhai, Y., Hao, Y., Zhang, H., Wang, D., Song, C., Qian, Z., Lesani, M., Krishnamurthy, S.V., Yu, P.: "UBITect: a precise and scalable method to detect use-before-initialization bugs in Linux kernel"; In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 221–232, 2020.

[Ziems and Wu 2021] Ziems, N., Wu, S.: "Security vulnerability detection using deep learning natural language processing"; In IEEE INFOCOM 2021-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), pages 1–6. IEEE, 2021.