New Tricks to Old Codes: Can AI Chatbots Replace Static Code Analysis Tools?

Omer Said Ozturk
Sabanci University
İstanbul, Turkiye

Emre Ekmekcioglu
Sabanci University
İstanbul, Turkiye

Orcun Cetin
Sabanci University
İstanbul, Turkiye

Budi Arief
University of Kent
Canterbury, England

Julio Hernandez-Castro
University of Kent
Canterbury, England

ABSTRACT

The prevalence and significance of web services in our daily lives make it imperative to ensure that they are – as much as possible – free from vulnerabilities. However, developing a complex piece of software free from any security vulnerabilities is hard, if not impossible. One way to progress towards achieving this holy grail is by using static code analysis tools to root out any common or known vulnerabilities that may accidentally be introduced during the development process. Static code analysis tools have significantly contributed to addressing the problem above, but are imperfect. It is conceivable that static code analysis can be improved by using AI-powered tools, which have recently increased in popularity. However, there is still very little work in analysing both types of tools' effectiveness, and this is a research gap that our paper aims to fill. We carried out a study involving 11 static code analysers, and one AI-powered chatbot named ChatGPT, to assess their effectiveness in detecting 92 vulnerabilities representing the top 10 known vulnerability categories in web applications, as classified by OWASP. We particularly focused on PHP vulnerabilities since it is one of the most widely used languages in web applications. However, it has few security mechanisms to help its software developers. We found that the success rate of ChatGPT in terms of finding security vulnerabilities in PHP is around 62-68%. At the same time, the best traditional static code analyser tested has a success rate of 32%. Even combining several traditional static code analysers (with the best features on certain aspects of detection) would only achieve a rate of 53%, which is still significantly lower than ChatGPT's success rate. Nonetheless, ChatGPT has a very high false positive rate of 91%. In comparison, the worst false positive rate of any traditional static code analyser is 82%. These findings highlight the promising potential of ChatGPT for improving the static code analysis process but reveal certain caveats (especially regarding accuracy) in its current state. Our findings suggest that one interesting possibility to explore in future works would be to pick the best of both worlds by combining traditional static code analysers with ChatGPT to find security vulnerabilities more effectively.

EICC 2023, June 14–15, 2023, Stavanger, Norway

ACM ISBN 978-1-4503-9829-9/23/06.
https://doi.org/10.1145/3590777.3590780

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

ChatGPT · AI · Static code analysis · PHP vulnerabilities · Tools evaluation · Vulnerability detection · AI in cyber security

ACM Reference Format:
Omer Said Ozturk, Emre Ekmekcioglu, Orcun Cetin, Budi Arief, and Julio Hernandez-Castro. 2023. New Tricks to Old Codes: Can AI Chatbots Replace Static Code Analysis Tools?. In *European Interdisciplinary Cybersecurity Conference (EICC 2023), June 14–15, 2023, Stavanger, Norway*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3590777.3590780

1 INTRODUCTION

A vulnerability, in the context of software engineering, is a defect in the implementation or design that opens a pathway for an attacker with the right set of skills to exploit the weaknesses of the targeted software systems [6]. Vulnerability remediation would cost more after software is being deployed [6]. Even worse, sometimes fixing one vulnerability could cause other vulnerabilities to be introduced [3].

For the reasons mentioned above, vulnerabilities must be identified and addressed in the early stages of software development life cycle, which typically includes 5 important stages: (i) requirements; (ii) design; (iii) implementation; (iv) testing; and (v) deployment. Each stage yields its own security recommendations. Typically, most of the vulnerabilities are found in the implementation stage of the software development life-cycle. In the implementation stage, developers frequently use static code analysers to evaluate their codes' security posture and identify security vulnerabilities.

Vulnerabilities exist in all types of software. As the world becomes more interconnected through the Internet, there is a growing concern caused by vulnerabilities on websites and web-based applications. This is due to the increased reliance that our society has on web services, which means any disruption or security compromise on these can cause significant negative consequences. Additionally, the "online" nature of these web services make them vulnerable to remote attacks at any time, which opens up the attack surface considerably. Many of these web services are generated by using the PHP programming language [10].

A developer may implement and launch a perfectly working PHP application like WordPress, but it is very likely that they would not anticipate all the ways that attackers on the internet might try to compromise their code.

To alleviate the problem of vulnerable software, there are tools – in particular, static analysis tools – that developers can use. There are various static code analysers aimed to detect vulnerabilities before deployment. However, remarkably little research has been undertaken into the effectiveness of these static code analysers.

AI-powered tools like ChatGPT [1] also have the potential to play a significant role in static code analysis for security. By using natural language processing, these tools are able to understand and interpret code written in programming languages like PHP and identify potential vulnerabilities. This can allow developers to identify and fix vulnerabilities in their code. Furthermore, by asking these tools to provide the corresponding Common Weakness Enumeration [2] (CWE) number for each vulnerability found, developers can understand the type of vulnerability and the potential impact on the application. Additionally, by asking for the patch of the identified vulnerabilities, developers can apply the fixes without having to manually search for a solution. Accordingly to the mentioned reasons, we also wanted to evaluate the efficiency of the well-known AI-powered tool ChatGPT for identifying and fixing vulnerabilities in PHP code.

In this paper, we present an extensive study comparing the effectiveness (in terms of the ability to detect software vulnerability correctly) of 11 different static code analysers and one AI-powered tool ChatGPT. For this study, 92 OWASP Top 10 Web Application Security Risks [3] were used to assess the vulnerability detection capabilities of the static code analysers chosen for this study.

Contributions. The key contributions of our paper are:

- The creation of a PHP code dataset with known OWASP Top 10 vulnerabilities, that we will share openly with the academic community[4] and could play the role of the ground truth when assessing the effectiveness of different static code analysers.
- A detailed study of the effectiveness of 11 of the most popular free open-source static code analysers.
- A detailed study of the effectiveness of AI-powered tool ChatGPT for static code analysis.
- The results of the scanning process both for free open-source static code analysers and ChatGPT will share openly with the academic community.
- A comparison between the capabilities of these analysers and ChatGPT at finding different categories of vulnerabilities, and a study of their combined coverage.

The rest of this paper is organised as follows. Section 2 outlines our approach, including the background information about the selection of the static code analysers, the process of scanning code using ChatGPT, information about the procedure used, and the technique used in evaluating the outputs. Section 3 provides the key findings of our research, along with the statistical data leading to the key insights obtained. Section 4 discusses the implications of our findings. Section 5 presents related work and discusses how our research complements them. Finally, Section 6 concludes our paper and provides several ideas for future work.

[1]https://openai.com/blog/chatgpt/
[2]https://cwe.mitre.org/index.html
[3]https://owasp.org/www-project-top-ten/
[4]https://github.com/New-Tricks-to-Old-Codes/Replace-Static-Analysis-Tools

2 METHODOLOGY

2.1 Vulnerability selection

We wrote the vulnerable code samples with reference to the OWASP Top 10 2021, including all 10 categories and prioritizing the more common ones in web applications for greater measurement efficiency. In total, they included 92 vulnerable codes by ensuring at least one code from each category and arranging the code numbers in their respective categories. To ensure accurate classification, we matched each code in the database with its corresponding CWE number from the MITRE CWE list. We verified the classification of vulnerable codes by matching them with CWE numbers and mapping them to their associated OWASP Top 10 categories. The distribution of the number of shown vulnerabilities in the code database is aligned with the most found vulnerabilities in web applications on the internet [9].

2.2 ChatGPT

ChatGPT is an AI-powered chatbot created by the OpenAI organization. It was first opened to the public in 2022 and Jan 9 Version is the latest release of the tool when this paper is written. As its popularity is increasing day by day, the community started using the tool to write code and bug fixing. It should be also noted that the official page of the tool gives a bug-fixing example as its first dialogue sample. In this study, we evaluated the vulnerability detection performance of the ChatGPT to see its advantages and disadvantages over static code analysis tools. A detailed explanation of the methodology that was followed during the scanning of the codes via ChatGPT is explained in the Study procedure section.

2.3 PHP Static Code analysers

In this study, we focused on the vulnerability detection performance of PHP-supporting open-source and free static code analysers. Our references for the selection process were the lists "Source Code Analysis Tools" by OWASP [5] and "Source Code Security Analysers" by NIST [6]. All free and open-source static code analysers supporting PHP have been chosen from the mentioned lists. Another criteria for the selection is the working environment, we chose the static code analysers which can be used locally, and we excluded the ones which need to upload the source code to a cloud environment or a repository-sharing platform such as Github. In total, we have evaluated 11 tools. The complete list of tools with their version number are given in Table 1.

2.4 Study procedure

This section includes a detailed explanation of the study procedure that we followed. The preparation of the PHP code database consists of developing vulnerable and secure codes, and manual testing for exploitation. The development and test phases were carried out in a Linux environment running Ubuntu (20.04) with PHP version 7.4, Apache2 (2.4.25), and MySQL (10.3.37-MariaDB). PHP 7.4 has been chosen because according to W3Techs, it is the most frequently used version of PHP [11]. Before running and testing processes, the

[5]https://owasp.org/www-community/Source_Code_Analysis_Tools
[6]https://www.nist.gov/itl/ssd/software-quality-group/source-code-security-analyzers

Table 1: Static code analysis tools considered in the study

Tool Name	Version
Betterscan CE	scanmycode3-ce
OWASP WAP	v2.1
phpcs-security-audit (PHPcs)	v3.7.1 (stable)
Pixy	No version information. Compiled from the source.
Progpilot	v1.0.0
Psalm	dev-master
RATS	No version information. Compiled from the source.
RIPS	v0.55
SonarQube CE	9.8.0-community (docker container)
Visual Code Grepper (VCG)	v2.2.0
Horusec	v2.8.0

manual testing phase was carried out. Every piece of code was tested manually. Each one of these vulnerable codes was exploited. On the other hand, compliant codes were tested by the same individuals and given to a penetration testing company [7] for validation. None of these compliant codes was exploited during this procedure.

During the study, for AI-based scanning, we used ChatGPT Jan 9 Version and the process consisted of two steps:

(1) **Finding security flaws:** We asked ChatGPT to identify any security flaws in the given PHP code and to provide the corresponding Common Weakness Enumeration (CWE) number for each flaw. The question pattern was "Think yourself as a cybersecurity specialist. Are there any security flaws in the following PHP code? If there are vulnerabilities, can you give the corresponding CWE number for each of them." The reason for asking ChatGPT to "Think yourself as a cybersecurity specialist" is that this question intends to make the AI have comprehensive knowledge of the domain and to be able to identify potential vulnerabilities in the PHP code.
(2) **Providing patches:** We asked ChatGPT to fix the identified vulnerabilities by providing a patch for the given code. The question pattern was "Can you fix these vulnerabilities?". We asked this question because some static code analysers have the ability to fix vulnerable code. Even though the vulnerability-fixing capabilities of the analysers were not considered while making comparisons, this feature is worth mentioning with its own advantages.

It should be mentioned that some vulnerabilities required multiple files, which were provided together with the question, as they were necessary to be given together to understand the vulnerability and fix it. We conducted scans for each vulnerability regularly and used the same version of the tool throughout the study. The questions and answers were logged on a file along with the scan date and time. Another thing to mention is the static code analysers are deterministic in nature, producing consistent results with each run. Conversely, the results generated by ChatGPT are non-deterministic and can vary with each execution. To address this issue, the scanning process is repeated and every single code is submitted to ChatGPT twice. This approach is taken in an effort to present a more comprehensive and robust set of findings.

In the process of scanning the codes via static code analysers, we used the mentioned environment. For each of the static code analysers, the provided options were reviewed and the appropriate option for vulnerability analysis was used in scanning. To avoid any wrong configuration of the aforementioned tools, all of the researchers configured them separately before the process.

2.5 Output evaluation

In the initial output evaluation, the codes were divided into two categories: vulnerable and secure. The vulnerable codes were further divided into three sub-categories: Found, Partially Found, and Not Found. For the secure codes, there were two sub-categories: Correct and False Positive.

The criteria for labelling outputs of vulnerable codes according to the three subcategories were agreed upon after a thorough review of all outputs and they are as follows: If the tool has the ability of categorisation, in order for its output to be considered as "Found", it must correctly categorise and identify the vulnerability in the code, then point to the relevant part of the code (e.g. the relevant function, variable, or the entire relevant line in the code). Otherwise, if the tool does not have the categorisation ability, it must point to the function, variable, or vulnerable line, in order for its output to be considered as "Found".

The output that does not contain any correct categorisation or contains half-right categorisation (for example, having a relationship "ParentOf", "ChildOf", "CanPrecede" with the correct CWE category) and indicating the function, variable or corresponding line from which the vulnerability originated are labelled "Partially found". In addition, since some of the vulnerabilities originate from the same function they can be miscategorised by the tools, in this case, we look for the function that makes the code vulnerable and we expect the tool for highlighting the corresponding function. An example of the mentioned case can be seen in SSRF and Local file inclusion vulnerabilities. Both of the vulnerabilities can be originated from the "file_get_contents( )" function in PHP, it is not an unexpected situation to categorise them wrong, thus we considered these cases as "Partially Found". The output that does not comply with the mentioned criteria, that does not correctly categorise, and/or does not show the function, variable, or line, where the vulnerability originates, is labelled as "Could not find".

The evaluation of the output of the secure code was done by respecting the mentioned two sub-categories which are "Correct" and "False positive". For being labelled as "False positive" the tool needs to detect a vulnerability that doesn't actually exist in the code. Otherwise, if the tool does not mention any vulnerability its output was evaluated as "Correct".

3 RESULTS

In this section, we present the vulnerability detection results of each of these tools. First, we looked at their performance when they were used on their own. Second, we investigated the performance of combining two or more of static code analysers, to see if an improved performance could be achieved by having these tools complement each other, and then we compared the results of the collaborations with the results of ChatGPT.

[7] https://vanderlog.com/

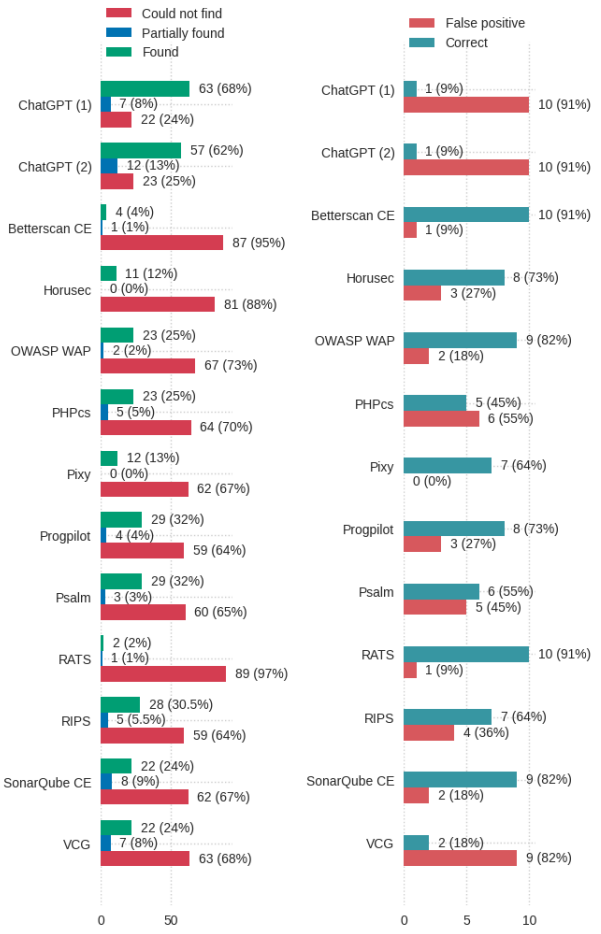

Figure 1: The tools' vulnerability detection performance (left) and compliant code performance (right)

3.1 Vulnerability detection efficacy of ChatGPT

As it is mentioned in section 2.4, unlike static code analysers, the output of ChatGPT is not deterministic. Hence each run may have different output and evaluation performance. For that reason, in this part we will give the results from two distinct ChatGPT scans on the same version. Even though the results are not exactly the same, they are quite similar.

At the first scan, the AI-powered tool detected 63 of the 92 vulnerable codes correctly (a 68% success rate). There are 7 vulnerabilities that the tool partially detected, and the remaining 22 vulnerabilities could not be caught by the tool. By considering these numbers, the tool can say that there is an issue in the relevant part of the code in terms of security with a rate of 8%, but it fails at a rate of 24% in detecting the vulnerabilities. The tool gave 10 "False positive" out of 11 secure codes. The statistics have shown that the tool reported 91% of the given codes are vulnerable, even if they are secure.

At the second scan, the AI-powered tool detected 57 of the 92 vulnerabilities correctly (a 62% success rate). There are 12 vulnerabilities that the tool partially detects, and the remaining 23 vulnerabilities could not be caught by the tool. By considering the obtained numbers, the tool can say that there is an issue in the relevant part of the code in terms of security with a rate of 13%, but it fails at a rate of 25% in detecting the vulnerabilities. Similarly to the first scan, the tool gave 10 "False positive" out of 11 secure codes.

Statistics show that ChatGPT has a high vulnerability detection rate, but also gives a high rate of "False positive" responses. The statistics of the first and the second scans are kind of proof of the non-deterministic nature mentioned above but it should also be considered that there is no huge deviation between the scan results. The advantages and drawbacks of the findings will be included in the Discussion section.

3.2 Vulnerability detection efficacy of Static Code Analysers

Due to the rule-based deterministic structure of static code analysers, the output of the tool does not change between run iterations and is strongly related to their rule set. Belong-side their rule set, there may be other factors that affect the performance of the static code analyser. For example, in this study, we realised that in some of the codes, Pixy static code analyser was giving syntax errors for our code and was not giving any output other than the syntax error. As mentioned before, we manually tested each vulnerability and they were running without any problems. For that reason, the results of Figure 1 are not adding up to 100% for Pixy.

3.2.1 Vulnerability detection efficacy of a single tool. First, we would like to determine the vulnerability detection efficacy of each static code analysis tool when being used individually. Figure 1 shows a general overview of the efficacy of the tools. As shown in Figure 1 (left), no single tool managed to detect all or the majority of the vulnerabilities.

There is no single static code analysis tool that could detect all the vulnerabilities. In fact, the most successful ones only managed to detect 32% of the vulnerable codes. Meanwhile, the least successful tool could only detect 2% of them. While the overall performance of Progpilot and Psalm is the best, other tools may perform better in individual categories. This can be seen in Table 2. For example, the overall efficacy of SonarQube CE is 24%, but it is the best-performing tool for the category of cryptographic failures.

3.2.2 Vulnerability detection efficacy of a combination of multiple tools. As mentioned before, different static code analysers may have different categories of which they are the best. Hence, using them in combination may increase the overall found vulnerability coverage. Each two-combination of all 11 static code analysers has been analysed. The best performing five pairs are given in Table 3. According to that table, instead of using a single tool, if we use two tools together, we can find more than 50% of the vulnerabilities.

3.3 Efficacy of AI vs Static code analyser

If we compare the individual performance of each static code analyser and ChatGPT, we see a big difference between the found vulnerability percentages. As mentioned before, the best-performing static code analysis tools are able to find 32% of the vulnerabilities while both ChatGPT iterations roughly double this identification. Even if two static code analysers are used in combination, they are outperformed by ChatGPT as can be seen in Table 3. The only statistical advantage of static analysis tools emerges when we consider

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

	Broken Access Control	Cryptographic Failures	Injection	Insecure Design	Security Misconfiguration	Vulnerable and Outdated Components	Identification and Authentication Failures	Software and Data Integrity Failures	Security Logging and Monitoring Failures	SSRF
Total Count	12	22	31	8	2	1	6	2	3	5
ChatGPT (1)	9 (75%)	16 (73%)	23 (74%)	6 (75%)	2 (100%)	0 (0%)	2 (33%)	1 (50%)	3 (100%)	1 (20%)
ChatGPT (2)	7 (58%)	12 (55%)	21 (68%)	6 (75%)	2 (100%)	1 (100%)	4 (67%)	2 (100%)	1 (33%)	1 (20%)
Betterscan CE	1 (8%)	0 (0%)	3 (10%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Horusec	0 (0%)	1 (5%)	9 (29%)	1 (12%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
OWASP WAP	3 (25%)	0 (0%)	17 (55%)	3 (38%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
PHPcs	3 (25%)	1 (5%)	16 (52%)	3 (38%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Pixy	0 (0%)	0 (0%)	12 (39%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Progpilot	6 (50%)	0 (0%)	21 (68%)	0 (0%)	0 (0%)	0 (0%)	2 (33%)	0 (0%)	0 (0%)	0 (0%)
Psalm	6 (50%)	0 (0%)	21 (68%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (50%)	0 (0%)	1 (20%)
RATS	0 (0%)	0 (0%)	2 (6%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
RIPS	6 (50%)	0 (0%)	18 (58%)	3 (38%)	0 (0%)	0 (0%)	0 (0%)	1 (50%)	0 (0%)	0 (0%)
SonarQube CE	0 (0%)	16 (73%)	2 (6%)	0 (0%)	1 (50%)	1 (100%)	2 (33%)	0 (0%)	0 (0%)	0 (0%)
VCG	2 (17%)	0 (0%)	20 (65%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)

Table 3: Combination of tools

	Found	Partially found	Could not find	False positive
		n=92		n=11
ChatGPT (1)	63 (68%)	7 (8%)	22 (24%)	10 (91%)
ChatGPT (2)	57 (62%)	12 (13%)	23 (25%)	10 (91%)
Combination				
Psalm and SonarQube CE	49 (53%)	11 (12%)	32 (35%)	6 (55%)
RIPS and SonarQube CE	48 (52%)	13 (14%)	31 (34%)	5 (45%)
Progpilot and SonarQube CE	47 (51%)	12 (13%)	33 (36%)	5 (45%)
OWASP WAP and SonarQube CE	43 (47%)	10 (11%)	39 (42%)	4 (36%)
VCG and SonarQube CE	42 (46%)	12 (13%)	38 (41%)	10 (91%)

compliant codes. For example, if we consider the ChatGPT scan 2 and compare it with RIPS and SonarQube CE combination, we see 36% less "False positive" with a trade-off of 10% less vulnerable code identification. In addition to overall performance, if a category-wise comparison is made between the performance of ChatGPT and static code analysis tools, we see none of the static code analysers is performing better than ChatGPT in each category. Furthermore, we observed that using ChatGPT with a static code analyser like VCG could enhance vulnerability detection performance at the expense of more false positives. Despite achieving high overall vulnerability coverage (75%) of vulnerable code, the approach also identifies all the compliant code as vulnerable.

4 DISCUSSION

It is obvious that ChatGPT has a much better rate of vulnerability detection than static code analysers when each of the tools is used alone. Even if ChatGPT's false positive rate is much higher than those of the traditional static code analysers, ChatGPT's high true positive rate shows that there is a great potential for using it for static code analysis. In fact, as it is mentioned in Section 3.3, even the ones with the best success rate among the 2 combinations of static code analysers could not reach the success rate of ChatGPT in vulnerability detection.

However, in real life, the usage of ChatGPT for static code analysis has some drawbacks that developers should pay attention to:

(1) *Privacy of sensitive code:* We have concerns about the privacy of the submitted code to the tool, we do not know what ChatGPT does with the submitted data thus, sensitive codes should not be given. What we suggest to developers when they need to scan their sensitive code is dividing the code into code snippets and only giving the necessary part to the tool, but it should be also noted that this approach may lead to some missing detection if the given code snippets were not selected carefully.
(2) *Non-deterministic nature of ChatGPT:* Another issue is the non-deterministic nature of ChatGPT, as it is mentioned in the previous sections the answers of the tool can vary in different executions. We suggest developers scan their code multiple times to get more accurate answers. It is also important to say that it is a good practice to test the patched codes manually.

We believe that ChatGPT has the potential to be breaking new ground in code analysis but according to our findings using a code analyser as a complementary measure would be the best practice.

5 RELATED WORK

Even though there are studies about using NLP models for detecting vulnerabilities and the effectiveness of static code analysers individually, we could not manage to find any related work covering both artificial intelligence (AI) and static code analysers.

5.1 Related work for AI to find vulnerabilities in code

There is no study in the literature in which ChatGPT was used in static code analysis. Relatedly, there are studies in which static code analysis is done using NLP models. Ziems et al., used natural language processing models BERT and LSTM in their study. Similar to our study they created a database containing software vulnerabilities written in C/C++, and they designed deep learning models to detect and classify them. According to their results, their best model had 93% accuracy in vulnerability detection [13].

5.2 Related work for static code vulnerability analysis tools

Several researchers investigated how the effectiveness of static code analysers in finding vulnerabilities in PHP code.

Da Fonseca et al. analysed 35 WordPress plugins for security vulnerabilities by using RIPS and phpSAFE. Although they were able to detect vulnerabilities in the plugins, the performance still needs to be improved in terms of vulnerability coverage and false positive removal [4].

Baset et al., after testing IDE security plugins for various languages, concluded that these tools lack information about the checks they perform on the codes [2]. Schuckert et al. mentioned a number of SQLi code patterns that are difficult for the code analysers to observe and would go undetected in the analysis, resulting in false negatives or false positives in the report [8]. Amankwah et al. concluded that no static analysis tool is perfect, some are able to detect a few vulnerabilities but they should all complement each other in order to create better results [1].

Various other studies have also looked into the effectiveness of static code analysers for other languages. Most notably, Zhioua et al. evaluated several static code analysis techniques and tools for C language and concluded that the tools were unable to identify most of the security vulnerabilities. Furthermore, even if they could detect a vulnerability, they would be unable to identify the security properties that might be affected by the vulnerability [12].

In another study, Perhinschi focused on vulnerability assessment in Java and C/C++ code samples using three different static code analysers. His results were quite similar to our findings. His study showed that the performance of static code analysers is highly correlated with the vulnerability type, and all three static code analysers missed many CWEs. He concluded that the static code analysers could not give adequate results despite their claims [7].

Lebanidze compared the generations of static code analysers and pointed out the reasons behind the low number of vulnerability detection, independent of programming language. Lebanidze drew a picture of how next-generation static code analysers should be constructed in order to detect problems that current-generation static code analysers are unable to find [5].

6 CONCLUSION

This paper presents a study in measuring the effectiveness of PHP-based static code analysers and the AI-powered tool ChatGPT. In this study, we carefully created a dataset of vulnerable PHP code with 92 PHP vulnerabilities based on OWASP Top 10 web application vulnerabilities. We used this dataset to assess the effectiveness of 11 popular free open-source static code analysers for PHP, as well as the effectiveness of ChatGPT in finding these vulnerabilities.

As AI is becoming more intertwined with our daily lives, AI-powered tools such as ChatGPT have increased in popularity. Instead of just searching for information or using ChatGPT as a simple chatbot, people started to use it to do software development work, such as writing code and fixing bugs. Nevertheless, there are currently not much information and very little knowledge about such usage. This led us to include ChatGPT in this study, to evaluate ChatGPT's effectiveness in detecting and fixing security bugs in software written in an old language like PHP, and to see if we can use ChatGPT efficiently and correctly in software development.

Our study sheds some light on the potential use of ChatGPT for finding software vulnerabilities. Based on our results, developers can use ChatGPT as a static code analysis tool, but with some precautionary measures mentioned in Section 4.

ACKNOWLEDGMENTS

This work was partly supported by the European Commission under the Horizon 2020 Programme (H2020), as part of the HEROES project (https://heroes-fct.eu/, Grant Agreement no. 101021801).

REFERENCES

[1] Richard Amankwah, Patrick Kwaku Kudjo, and Samuel Yeboah Antwi. 2017. Evaluation of software vulnerability detection methods and tools: a review. *International Journal of Computer Applications* 169, 8 (2017), 22–27.
[2] Aniqua Z Baset and Tamara Denning. 2017. Ide plugins for detecting input-validation vulnerabilities. In *2017 IEEE Security and Privacy Workshops (SPW)*. IEEE, 143–146.
[3] Frederick P Brooks Jr. 1995. *The mythical man-month: essays on software engineering*. Pearson Education.
[4] José Carlos Coelho Martins da Fonseca and Marco Paulo Amorim Vieira. 2014. A practical experience on the impact of plugins in web security. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*. IEEE, 21–30.
[5] Evgeny Lebanidze. 2008. The Need for Fourth Generation Static Analysis Tools for Security–From Bugs to Flaws. In *Application Security Conference*.
[6] Mark S Merkow and Lakshmikanth Raghavan. 2010. *Secure and resilient software development*. CRC Press.
[7] Andrei M Perhinschi. 2015. *Static Code Analysis: On Detection of Security Vulnerabilities and Classification of Warning Messages*. West Virginia University.
[8] Felix Schuckert, Basel Katt, and Hanno Langweg. 2020. Diffcult SQLi Code Patterns for Static Code Analysis Tools. In *Norsk IKT-konferanse for forskning og utdanning*.
[9] Veracode. 2020. Volume 11 the State of Software Security: Flaw Frequency by Language. https://www.veracode.com/sites/default/files/pdf/resources/infosheets/state-of-software-security-volume-11-government-and-education-veracode-infosheet.pdf. Last access on April 8th, 2023.
[10] W3Techs. 2023. Usage statistics of PHP for websites. https://w3techs.com/technologies/details/pl-php. Last access on April 8th, 2023.
[11] W3Techs. 2023. Usage statistics of PHP version 7 for websites. https://w3techs.com/technologies/details/pl-php/7. Last access on April 8th, 2023.
[12] Zeineb Zhioua, Stuart Short, and Yves Roudier. 2014. Static code analysis for software security verification: Problems and approaches. In *IEEE 38th Int'l Computer Software and Applications Conference Workshops*. IEEE, 102–109.
[13] Noah Ziems and Shaoen Wu. 2021. Security vulnerability detection using deep learning natural language processing. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 1–6.