EMPIRICAL STUDY TO EVALUATE CHATGPT FOR STATIC ANALYSIS AGAINST RULE-BASED APPROACH

by
MOHSIN MUNAWAR

Submitted to the Graduate School of Cyber Security
in partial fulfillment of
the requirements for the degree of Master of Science

Sabancı University
July 2023

MOHSIN MUNAWAR 2023 ©

All Rights Reserved

ABSTRACT

EMPIRICAL STUDY TO EVALUATE CHATGPT FOR STATIC ANALYSIS AGAINST RULE-BASED APPROACH

Keywords: Vulnerability detection, OWASP Top 10, Static Analysis, ChatGPT

The cybersecurity industry is undergoing rapid transformation, making it increasingly difficult for security professionals and developers to stay abreast of the latest vulnerabilities. This leads to an urgent need for early detection of these vulnerabilities through static analysis during the software development life cycle [1]. Static analysis tools have the potential to greatly benefit developers by identifying weaknesses during the development phase. However, their limited ability to identify security vulnerabilities leads many developers to abandon these tools. AI-based security tools have demonstrated promising outcomes. The use of AI-powered tools to enhance vulnerability detection is expected to rise in popularity. Yet, there needs to be more research comparing the effectiveness of AI-powered tools to rule-based ones. Our paper seeks to address this research gap by evaluating the effectiveness of both types of tools. In this paper, we assessed the effectiveness of 10 static code analysers and a renowned AI-powered chatbot named ChatGPT in detecting 354 JAVA vulnerabilities that belong to the top 10 widely recognised vulnerability categories in web applications, as classified by the OWASP organisation. Our analysis revealed that even a combination of all the static code analysers detected a disappointingly low number of vulnerabilities. In contrast, ChatGPT performed remarkably well in detecting vulnerabilities in the same code samples. In fact, ChatGPT was able to detect 70% of the vulnerabilities, which is a significant improvement over the 25% detection rate achieved by the static analysers. Furthermore, we assessed the performance of ChatGPT in false positive generation with 354 patched codes of the datasets we used for vulnerability analysis. ChatGPT issue 0% false positives in contrast with the large number of false positive generation by the static analysers. Finally, we found that ChatGPT provided 100% accurate patches for 20% of the vulnerabilities. All in all, ChatGPT seems to be a relatively effective mechanism for detecting and fixing vulnerabilities in the source codes. We reflect on our main findings and potential issues regarding how to use ChatGPT and other static code analysers for vulnerability detection.

ÖZET

CHATGPT'NİN STATİK ANALİZ İÇİN KURAL TABANLI YAKLAŞIMA KARŞI DEĞERLENDİRİLMESİ İÇİN DENEYSEL ÇALIŞMA

Anahtar Kelimeler: Zayıflıkların Tespiti, OWASP Üst 10, Statik Analiz, ChatGPT

Cyber güvenlik endüstrisi hızlı bir dönüşüm geçiriyor, bu da güvenlik uzmanları ve geliştiricilerin en yeni zayıflıklar hakkında bilgi sahibi olmasını giderek zorlaştırıyor [1]. Bu durum, yazılım geliştirme yaşam döngüsü sırasında statik analiz yoluyla bu zayıflıkların erken tespiti için acil bir ihtiyaca yol açıyor. Statik analiz araçları, geliştirme aşamasında zayıflıkları belirleyerek geliştiricilere büyük fayda sağlama potansiyeline sahiptir. Bununla birlikte, güvenlik açıklıklarını tespit etme konusundaki sınırlı yetenekleri, birçok geliştiricinin bu araçları terk etmesine yol açmaktadır. Bu alanda yapay zeka tabanlı güvenlik araçları umut verici sonuçlar göstermiştir. AI destekli araçların zayıflık tespitini artırmak için kullanımının popülerlik kazanması bekleniyor. Ancak, AI destekli araçların etkinliğini kural tabanlı araçlarla karşılaştıran daha fazla araştırmaya ihtiyaç duyulmaktadır. Makalemiz, her iki tür aracın etkinliğini değerlendirerek bu araştırma boşluğunu ele almayı amaçlamaktadır. Bu makalede, OWASP organizasyonu tarafından web uygulamalarında yaygın olarak tanınan en üst 10 zayıflık kategorisine ait 354 JAVA zayıflığını tespit etmede, 10 adet statik kod analiz aracının ve tanınmış bir AI destekli sohbet botu olan ChatGPT'nin etkinliğini değerlendirdik. Analizimiz, tüm statik kod analiz araçlarının bile düşük bir sayıda zayıflık tespit ettiğini ortaya koydu. Buna karşılık, ChatGPT aynı kod örneklerinde zayıflıkları tespit etmede olağanüstü bir performans sergiledi. Aslında, ChatGPT, zayıflıkların %70'ini tespit etmeyi başardı ki bu, statik analiz araçlarının %25'lik tespit oranına göre önemli bir iyileştirme anlamına geliyor. Ayrıca, ChatGPT'nin yanlış pozitif oluşturma performansını, zayıflık analizi için kullandığımız veri setlerindeki 354 yamalı kodla birlikte değerlendirdik. ChatGPT, statik analiz araçlarının ürettiği büyük miktardaki yanlış pozitiflerin aksine, %0 yanlış pozitif üretti. Son olarak, ChatGPT'nin zayıflıkların %20'si için %100 doğru yamalar sağladığını bulduk. Genel olarak, ChatGPT, kaynak kodlardaki zayıflıkları tespit etme ve düzeltme konusunda nispeten etkili bir mekanizma gibi görünmektedir. ChatGPT ve diğer statik kod analiz araçlarını zayıflık tespiti için nasıl kullanacağımıza dair ana bulgularımızı ve potansiyel sorunları değerlendiriyoruz.

ACKNOWLEDGEMENTS

Firstly, I would like to thank my supervisor, Dr. Orçun Çetin, for his continuous support, encouragement, and guidance.

I would like to express my heartfelt gratitude to my loving family, Maaz, Musa, and Daniya, as well as my dear mother and father. Your unwavering support and encouragement have been a constant source of strength throughout my life's journey.

to my family

TABLE OF CONTENTS

LIST OF TABLES .... **ix**

LIST OF FIGURES .... **x**

1. INTRODUCTION **1**
- 1.1. Background and Motivation 1
- 1.2. Research Problem 2
- 1.3. Contributions 3
- 1.4. Thesis Structure 3

2. LITERATURE REVIEW **4**
- 2.1. Rules-Based Static Analysers 4
- 2.2. Models Based Analysis 6
- 2.3. AI-Powered Chatbot Based Analysis 7

3. RESEARCH METHODOLOGY **9**
- 3.1. Research Design 9
- 3.2. Data Collection 9
- 3.3. Tools' Selection 12
- 3.4. Static Analysis Tools 12
- 3.5. ChatGPT 13
- 3.6. Study Procedure 14
- 3.7. Evaluation Metrics 16

4. RESULTS **17**
- 4.1. Overall Detection Rates 17
- 4.2. Vulnerability Categories 21
- 4.3. Efficacy of Vulnerability Patching 25
- 4.4. False Positive Generation Analysis for Patched Codes 26

5. DISCUSSION **27**

5.1. Effectiveness of Vulnerability Detection 27
5.2. Privacy Issues and Concerns 28
5.3. Usability and Other Issues 29
5.4. Threats to Validity 30
5.5. Enhancing ChatGPT for Static Analysis 30

6. CONCLUSION AND FUTURE WORK 32

BIBLIOGRAPHY 33

LIST OF TABLES

Table 4.1. Comparison Based on Datasets 18
Table 4.2. Efficiency of Each Tool 22
Table 4.3. Comparison of Vulnerability Detection Rates between Only Tools and Only ChatGPT 24
Table 4.4. The categories of CWEs that were primarily detected by Static code Analysers 25
Table 4.5. ChatGPT Phase and Fix Results 26

LIST OF FIGURES

Figure 3.1. The Distribution of the Prepared Evaluation Dataset 11

Figure 4.1. Tools' Vulnerability Detection Rates (Recalls) Based on Dataset 19

Figure 4.2. Venn diagram Showing the Number of Vulnerabilities Found .. 20

Figure 4.3. GPT-3.5 vs GPT-4 API Results 20

Figure 4.4. OWASP TOP 10 Vulnerabilities Detection Rates (Recalls) for Each Tool 23

Figure 4.5. OWASP TOP 10 Vulnerabilities Detection Rates (Recalls) for Each Tool 23

1. INTRODUCTION

1.1 Background and Motivation

The cybersecurity and software development communities continually strive to identify vulnerabilities that may become security risks [2]. Proactively identifying vulnerabilities during the development phase allows developers to take necessary measures to reduce potential security breaches or misuse. Early identification of vulnerabilities also reduces the time and resources required for fixes, as fixing the vulnerabilities in the development phase is typically faster and less expensive than fixing them after the software has been released.

For early vulnerability identification, the cybersecurity community developed static code analysers to discover and address potential issues in the source code automatically. Using static code analysis tools, developers can identify software vulnerabilities, inadequate programming practices, and code violations, providing valuable insights into potential security risks and helping developers improve their code's overall security. In other words, these tools are crucial in detecting critical issues that could result in significant security compromises in deployed systems.

While static code analysis tools offer numerous benefits, especially in identifying vulnerabilities that might be overlooked during manual audits, they can also have some limitations that can impact their effectiveness [3]. For instance, many static code analysis tools rely solely on rule-based detection algorithms, which can limit their ability to detect new or evolving security threats. This means that there is a risk that new or uncommon vulnerabilities may not be identified or addressed, potentially leading to security breaches. Last but not least, these tools may have difficulties in identifying complex vulnerabilities that are beyond the scope of their common rule-based algorithms. Complex vulnerabilities can often originate from complex

relationships between different software components or dependencies, making them extremely difficult to detect using rule-based systems. Even regularly maintained static code analysers can detect a very limited number of complex vulnerabilities.

These limitations highlight the importance of using AI-based systems that can be trained with a large amount of both vulnerable and secure code samples to identify common as well as complex and uncommon vulnerabilities. Generative pretrained transformer model–driven artificial intelligence (AI) chatbots, such as ChatGPT, offer significant potential for static code analysis in the field of secure code development. These tools, with the help of natural language processing (NLP), are capable of interpreting and analysing source code in order to identify potential security vulnerabilities. Unlike rule-based algorithms, tools that use NLP techniques can understand and reason about the code, leading to more accurate detection of complex vulnerabilities. In addition, while we did ask the tools to assign corresponding Common Weakness Enumeration (CWE) numbers to each vulnerability detected, it is important to note that our main aim is not solely focused on finding CWEs. After providing multiple prompts to ChatGPT, asking it for CWE detection gave us the most fruitful results. And by obtaining CWE numbers, developers can gain valuable insight into the nature of the vulnerability and its potential impact on the application. Alternatively, by requesting patches for the vulnerabilities, developers can quickly and efficiently fix the issues without having to spend time searching for patches and code fixes manually. Remarkably little research has been undertaken into understanding the efficacy of large language model–driven artificial intelligence tools vulnerability detection and fixing, compared to rule-based systems.

1.2 Research Problem

In this research, we evaluated and compared the effectiveness of the widely known large language model–driven artificial intelligence tool ChatGPT and ten rule-based static code analysers to detect and fix vulnerabilities in JAVA code. For the study, we utilised two datasets of vulnerable JAVA programs. The first dataset was the Juliet Suite, a widely recognised benchmark for testing static analysers. The second, unique, dataset was a custom one we created, inspired by the commonly mentioned vulnerabilities from renowned organisations, such as SANS, MITRE, OWASP, and SonarQube. Moreover, the second dataset also contains one Log4j vulnerability, which got discovered after the training dataset of ChatGPT was finalised.

1.3 Contributions

We make the following contributions in this research:

- We present the first extensive study comparing ChatGPT's vulnerability detection efficacy to 10 static code analysers. Our results showed that ChatGPT outperformed all other static code analysers in terms of vulnerability detection rates. Additionally, the non-deterministic nature of ChatGPT can detect vulnerabilities that were undetected in the previous runs.
- Although ChatGPT substantially outperformed all the static code analysers, we found out that static code analysers can still uncover a small number of critical security vulnerabilities that ChatGPT is unable to detect.
- Our findings indicated that ChatGPT was able to provide 100% accurate patches for 20% of the vulnerabilities detected.

1.4 Thesis Structure

The remainder of this thesis is organised as follows: Chapter 2 gives a comprehensive review of other similar studies in the field of cyber security. Chapter 3 gives our research methodology in full detail, including our experiments, dataset collection, tool selection and evaluation. Chapter 4 presents the evaluation results and findings, comparing ChatGPT with traditional static analysers. Chapter 5 discusses the implications of the results, limitations, and challenges encountered during the research. Chapter 6 gives a summary of the research findings, the contributions, and provides practical recommendations for potential future research directions.

In conclusion, this research aims to contribute to the field of software security by exploring the capabilities of ChatGPT as an AI-powered tool for vulnerability detection in JAVA code. By comparing its performance with traditional static analysers, we aim to shed light on the potential benefits and limitations of AI-based approaches in static analysis. The findings and recommendations derived from this research can guide future developments in AI-based static analysis tools and enhance the security and robustness of software systems.

2. LITERATURE REVIEW

This section states studies that use different approaches for the static analysis and proposed further researches.

2.1 Rules-Based Static Analysers

Software vulnerability prediction is an important task in software development, and many software developers use third-party libraries to help them detect the vulnerabilities in their source codes [4]. Existing software vulnerability tools mostly utilise static and/or dynamic code analysing techniques [5]. Static code analysing techniques analyse the source code of a program without actually running it [6], while dynamic code analysing techniques monitor the program's execution at runtime.

Several studies have been conducted to compare the performance of static and dynamic tools and help security practitioners choose the right tools. Static code analysers are effective in identifying a wide range of vulnerabilities, but are less effective at identifying vulnerabilities that require runtime information. On the other hand, dynamic code analysers are more effective at identifying vulnerabilities that require runtime information, but have a higher false positive rate [7].

In Imtiaz et al. [8] manually analysed the output of 9 industry-leading code analysing tools on web applications, OpenMRS, composed of Maven (Java) and npm (JavaScript) projects. Their analysis shows that the tool's performance is highly correlated to the accuracy, up-to-dateness, and completeness of the vulnerability database, and when it comes to predicting new unique vulnerabilities, their predictions become almost random. They conclude that practitioners should not rely on any single tool at present, as that can result in missing known vulnerabilities.

Various studies have focused on whether static analysis tools are worthwhile for

developers. Mahmood et al. [9] mention that the developers cannot fully depend on the static analysis tools as they do not always point out issues with the code. They tested Yasca, PMD, LAPSE+, and FindBugs, finding that these tools can find some security vulnerabilities in the codes. Nevertheless, they concluded that such tools could assist developers in detecting simple vulnerabilities during the development process. In [10], Walker et al. mention that these tools should adapt to changing methodologies developers are using in software development. Most of the tools lack a centralised overview and multi-repository support, which would be extremely helpful for the developers to view a quick overview from a manageable dashboard.

Gomes et al. [11] claim that the static code analysers return a large number of false positive warnings which results in true positive warnings getting buried within the false positives, making it extremely hard for the developers to find the actual bugs in the code. This becomes a frustrating operation and discourages the use of such tools for the developers. They also mention that the warning reports should be better readable and informative so that they can be matched to the code changes.

Furthermore, as JAVA is widely used in the software industry, some literature studies focused their investigations on evaluating the performance of static code analysers in predicting software vulnerabilities in Java source codes. Rutar et al. [12] conducted a comparison of five Bug Finding Tools, namely Bandera, ESC/Java, FindBugs, JLint, and PMD, by cross-checking their bug reports and warnings on Java programs. Their experimental results showed that no tool strictly subsumes another, and the tools often identify non-overlapping bugs.

Lenarduzzi et al. [13] provided a critical comparison of six popular static analysis tools (SATs) for Java projects: Better Code Hub, CheckStyle, Coverity Scan, FindBugs, PMD, and SonarQube. They compared the detection, agreement, and precision of these tools and found that SonarQube was the most effective tool in detecting quality issues. However, there was little to no agreement among the tools regarding the specific quality issues detected, indicating that different tools identify different forms of quality problems. The precision of the considered tools ranges between 18% and 86%, suggesting that the practical accuracy of some tools is significantly impacted by false positives. It emphasises the fact that these tools often flag code segments as vulnerable or problematic, even though they are actually safe. The prevalence of false positives can significantly impact the usefulness and effectiveness of these tools in real-world scenarios, as developers may need to spend substantial time and effort investigating and addressing non-existent issues.

2.2 Models Based Analysis

The findings of literature studies suggest that many static code analysers tend to generate false positives, leading to actual vulnerabilities or bugs being overlooked [14]. Moreover, most static tools lack the flexibility of custom configurations and updating mechanisms. To overcome these challenges, researchers have proposed and evaluated the use of machine learning techniques for software vulnerability prediction. These studies involve exploring various classification and deep learning (DL) models [15], as well as investigating the impact of different code representation techniques [16], such as software metrics [17], historical metrics [18], text-mining based metrics, and embedding metrics [19]. Results show that machine learning-based techniques can provide more accurate and efficient vulnerability detection while reducing false positives, making them a promising approach for improving code security.

Recent research has also investigated the application of natural language processing (NLP) [20] techniques to anticipate software vulnerabilities. Recently, transformer-based NLP models [21], which are large-scale pre-trained language models, have demonstrated superior performance compared to existing NLP and DL-based models in text-mining tasks.

In a recent study [22], Li et al. presented IVDetect, a transformer-based model for vulnerability detection, which considers vulnerable statements and their surrounding contexts by analysing the data and control dependencies. The authors employed interpretable artificial intelligence (AI) to provide fine-grained interpretations, including the sub-graphs in the Program Dependency Graphs (PDGs) with the essential statements that are related to the detected vulnerability. Their findings demonstrate that IVDetect outperforms existing DL-based approaches by 43%-84% and 105%-255% in top-10 normalised discounted cumulative gain (nDCG) and mean average precision (MAP) ranking scores, respectively. To evaluate the model's ability for statement-level vulnerability detection, two line-based models have been proposed in conjunction with the IVDetect model.

LineVul model, proposed by Fu et al. in their recent study [23], which is an improved version of the IVDetect approach. This study proposed a transformer-based line-level vulnerability prediction approach to address several limitations of the IVDetect approach. The authors reported that LineVul achieved 160%-379% higher F1-measure for function-level predictions and 12%-25% higher top-10 accuracy for line-level predictions compared to baseline approaches. Additionally, their analysis showed that LineVul accurately predicted vulnerable functions affected by

the Top-25 CWEs with a rate of 75%-100%. And, LineVD framework, proposed by Hin et al. in their recent study [24], which formulates statement-level vulnerability detection as a node classification task. LineVD leverages control and data dependencies between statements using graph neural networks and a transformer-based model to encode the raw source code tokens. By addressing the conflicting outputs between function-level and statement-level information, LineVD significantly improves prediction performance without vulnerability status for function code. Their experiments against a collection of C/C++ vulnerabilities obtained from multiple real-world projects demonstrated an increase of 105% in F1-score over current state-of-the-art approaches.

2.3 AI-Powered Chatbot Based Analysis

In this study, we compare the performance of the well-known static analysis tools in detecting vulnerabilities in Java source codes to that of ChatGPT [25], a large-scale pre-trained language model that has been shown to perform well on a variety of NLP tasks.

A comparison of different GPT-3 models was performed by Cheshkovet et al. [26] to find security issues in JAVA codes. This was achieved by testing the models using vulnerable and fixed code from GitHub. They tested both vulnerable and patched functions with ChatGPT and GPT-3 models to ascertain the precision and F1 scores of the models. The text-davinci-003 gave the precision, recall, and F1 score as 0.50, 0.99, and 0.67 respectively, whereas gpt-3.5-turbo gave 0.51, 0.80, and 0.62 respectively. It was concluded that both models performed poorly compared to a basic classifier and struggled to identify vulnerabilities accurately. These models might be good at other programming tasks, like solving challenges, but they still have a lot to learn when it comes to finding security problems in code. They suggest that more research is needed to improve their effectiveness and provide recommendations for future studies. The primary distinction between their study and ours lies in the dataset composition. In their study, ChatGPT was exclusively fed with vulnerable and patched functions, whereas in our study, we supplied working codes to ChatGPT. This difference in dataset composition likely contributed to the better results that we have achieved. Being fed with a working code, ChatGPT is able to gain a deeper understanding of the semantics and logic of the program and its data flow.

A similar study to our research was performed by Ozturk et al. [27]. In this study, they carried out a study to compare the efficiency of ChatGPT against 11 static analysers for 92 PHP vulnerabilities. Their analysis also placed ChatGPT above the static analysers as ChatGPT was able to detect 62-68% of the vulnerabilities compared with 32% success rate of the analysers.

3. RESEARCH METHODOLOGY

This section outlines the research design, data collection process, experimental setup, evaluation metrics, and statistical analysis methods employed in this study.

3.1 Research Design

The research design is structured as a comparative study between ChatGPT and traditional static analysers for vulnerability detection in Java code. The study involves evaluating the performance of ChatGPT and comparing it with the detection capabilities of 10 off-the-shelf traditional static analysis tools. The evaluation is conducted using two distinct datasets: the Juliet Suite v1.3 dataset and a unique dataset inspired by industry standards.

3.2 Data Collection

The datasets used in this research consist of JAVA software vulnerabilities and include runnable code that embodies the most common vulnerabilities. These datasets were sourced from the leading cybersecurity organisations and companies that publish lists of significant vulnerabilities, covering each of OWASP's Top 10 categories for web applications.

One of the datasets utilised in this research was the Juliet Test Suite [28], created by the National Security Agency (NSA) to evaluate the performance of static analysis tools. It includes compilable code for multiple vulnerabilities and helps assess the

accuracy of the results produced by these tools. By comparing the results of different analysers and ChatGPT for the same vulnerabilities, the test suite can highlight flaws that are missed by some or all of the analysers. The suite covers a wide range of security vulnerabilities through examples organised under 112 different Common Weakness Enumerations(CWEs) [28].

This study involved the selection of three randomly chosen vulnerable codes from each of the 112 CWEs in the Juliet Test Suite. When there were fewer than three vulnerable code samples for a CWE category, we made sure to incorporate all the vulnerable code samples that were available into our study. This methodology resulted in a total of 304 distinct vulnerabilities being selected for assessment. These codes were subjected to multiple pre-processing steps, including the removal of comments, renaming of class and function names to randomised 8-letter names, and changing of variables and filenames that indicated potential vulnerabilities. These steps aimed to obscure the true nature of the code and to ensure that the results were not influenced by external factors.

The second "unique" dataset utilised in this research was generated manually using various sources. It consisted of 50 vulnerability-prone codes which were initially evaluated for their potential exploits for each vulnerability. These vulnerabilities are randomly selected from SANS Top 25, Mitre Top 25, and OWASP Top 10 (2021) web application vulnerability examples. In addition, we incorporated the Log4j vulnerability into our study, which was identified after the final update of the training dataset used by ChatGPT.

SANS Top 25 list is a comprehensive and well-researched collection of the most critical software vulnerabilities [29]. It's an important resource for software developers, security researchers, and organisations to understand the types of vulnerabilities that pose the greatest threat to their software systems and applications. The list is updated annually based on a thorough analysis of the latest security data. It also provides detailed information on each vulnerability, including its description, impact, and recommended mitigations.

Mitre Top 25 is a compilation of the most critical and widespread software weaknesses [30] that pose a threat to organisations. It is derived from a vast knowledge base that keeps track of all adverse cybersecurity tactics and techniques used by potential attackers throughout the attack life cycle. The list is constantly updated to reflect the current threat landscape and help organisations stay ahead of potential attacks [31].

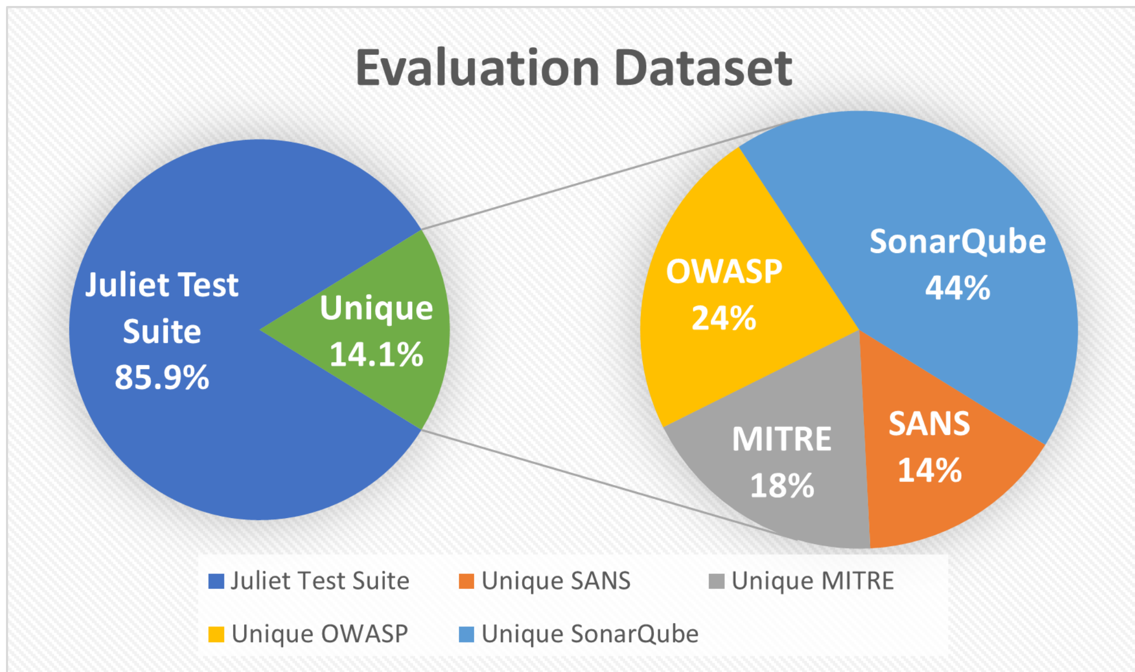

Figure 3.1 The Distribution of the Prepared Evaluation Dataset

The OWASP Top 10 is a widely recognised set of the most critical web application security risks [32]. It is a free and open-source resource that is updated annually and provides a comprehensive list of the top 10 vulnerabilities based on attack statistics from the previous year.

Lastly, SonarQube provides a large database of potential vulnerabilities for all popular programming languages, helping catch tricky bugs and prevent undefined behaviour from impacting end-users. It has easy integration with multiple IDEs and both local and cloud-based versions for testing.

Eventually, after the careful curation of the aforementioned five datasets, a set of 354 code samples is selected for evaluation. We also patched these 354 vulnerable code samples to evaluate the false positive result generation by ChatGPT. As illustrated in Figure 3.1, the evaluation set is further divided into two subsets: the Unique subset (14.1% of the data), representing the most commonly occurring vulnerabilities, and the Juliet subset (85.9% of the data), consisting of various vulnerability types. This division is instrumental in evaluating the tool's performance on popular vulnerabilities against less common ones.

3.3 Tools' Selection

This section describes the process of selecting the traditional static analysers used in the comparative evaluation with ChatGPT for vulnerability detection in JAVA codes.

3.4 Static Analysis Tools

In the realm of cybersecurity, numerous static analysis tools are available online, both commercial and free, which claim to detect security vulnerabilities and bugs in code. For this paper, we utilised OWASP's "Source Code Analysis Tools[1]" and NIST's "Source Code Security Analysers[2]" lists as the basis for our selection process. Subsequently, we picked ten free static code analysers randomly from the aforementioned lists. These analysers can be installed locally, integrated as extensions to JAVA IDEs, or even scan git commits directly. By utilising these tools, we aim to measure the effectiveness of static code analysers' ability to detect vulnerabilities presented in the source code. Next, we provide more information on these tools.

CodeFlow [33] is a powerful tool that integrates fully-configurable static analysis into the software development process, resulting in faster and more efficient coding. Although it is not open-source, CodeFlow integrates seamlessly with GitHub and performs automatic analysis on each code commit. The reporting style of the tool is user-friendly and easily understandable. CodeFlow automatically detects new static analysis issues, code duplications, and changes in cyclomatic complexity with each commit, making it a valuable resource for increasing the quality of JAVA code.

SemGrep [34] is an open-source static analysis tool that promotes faster coding and that enables the detection of bugs and adherence to coding standards in the code. The analysis can be performed on either a local machine or in a build environment. With over 2000 community-driven rules, SemGrep covers a wide range of areas, including security, correctness, and performance bugs, making it a comprehensive resource for increasing the security and quality of software products [35].

---

[1] https://owasp.org/www-community/Source_Code_Analysis_Tools

[2] https://www.nist.gov/itl/ssd/software-quality-group/source-code-security-analysers

CodeBeat [36], **Codiga** [37], and **DeepSource** [38] are configurable tools that integrate with GitHub repositories, tracking new commits and performing code analysis on each request. These tools play a crucial role in keeping bugs and vulnerabilities at bay during the software production process. By continuously monitoring code changes, they help increase the security and quality of the software systems.

PMD [39] is a specialised JAVA source code analyser that functions as an extension for JAVA IDEs. It excels in identifying program flaws, such as redundant code, empty code blocks, and more, helping to maintain the coding standards of software.

FindBugs [40] (also known as SpotBugs) is an open-source tool for analysing JAVA code. It scans bytecodes for bug patterns and defective code, reporting its findings as warnings. It is important to note that not all warnings indicate defects. Despite its lack of maintenance and updated ruleset, FindBugs remains a valuable resource for JAVA and provides integration with common IDEs.

SonarQube [41] is a widely recognised static code analysis tool that provides a vast database of potential vulnerabilities for a range of programming languages. Its goal is to identify and prevent tricky bugs that could negatively impact end-users. SonarQube offers integration with IDEs [41] as well as local and cloud-based testing frameworks, making it a versatile and comprehensive resource.

JLint [42] was a widely used static analysis tool known for its ability to highlight deviations from coding standards and detect bugs. Since its acquisition by Sonar, JLint has been renamed to **SonarLint**. Its ease of installation as an extension for multiple JAVA IDEs makes it a convenient and accessible tool.

3.5 ChatGPT

ChatGPT [25] is a cutting-edge large language model developed by OpenAI that has the ability to generate text based on the input it receives. It uses deep learning algorithms and has been trained on a diverse range of internet text to understand and respond in a variety of languages and topics. It uses a transformer-based architecture (on top of OpenAI's GPT-3 [43] family) and is fine-tuned (with both supervised and reinforcement learning techniques) on specific tasks such as answering questions, generating text, and translation.

While it has not been specifically designed for cybersecurity, it has the capabil-

ity to respond to questions and provide information on the topic of cybersecurity and related fields. This makes ChatGPT a valuable resource for those looking to learn about cybersecurity, as well as for researchers and experts who are seeking to automate cybersecurity-related tasks.

With its advanced language processing capabilities, ChatGPT is able to generate highly readable and informative text on a wide range of cybersecurity topics, from the basics of computer security to more advanced topics. Additionally, its ability to understand natural language inputs allows it to respond to questions in a conversational manner, making it an accessible and user-friendly resource for anyone interested in learning about cybersecurity.

3.6 Study Procedure

In this section, we describe the study procedure and our evaluation criteria. Code assessment phases were carried out on a Windows 10 machine with Java version 17.02, Tomcat version 9.0.73, and MySQL workbench 8.0. Before the assessment, we installed all static code analysers on a single device and established a new ChatGPT account with the Jan 9, 2023, Version. In June, we repeated ChatGPT part of the experiment with the recently released API using GPT-3.5 and GPT-4 models.

During the assessment, we performed a vulnerability assessment with ChatGPT twice for both vulnerable and patched datasets. Our evaluation included the experiments performed on GPT-3.5 (Jan 9 version) via the user interface (UI), and GPT-3.5 and GPT-4 (May 24 version) via the API.

For both experiments, we conducted 2 vulnerability detection runs for vulnerable codes. In the first run, we used all the vulnerable code samples in our datasets. In the second run, our focus was on the vulnerabilities that were not detected during the initial run. This specific approach was adopted to test and evaluate the non-deterministic nature of ChatGPT. By targeting undetected vulnerabilities, we aimed to explore the system's capabilities in uncovering previously missed security issues.

The second run of ChatGPT vulnerability analysis was separated from the first run. A new account and chat windows were used which made sure our previous chats or results did not affect the new results. To categorize our results accurately, we labeled the outcomes as ChatGPT(1) for the first run and ChatGPT(2) for the

second run. For the patched code, only one experiment was conducted to observe false positives. To accomplish this experiment both vulnerable datasets have been visited and necessary patches were made and approved by experts. Juliet dataset offered patched versions of the vulnerable code that they provided. We tested each patched code before adding it to the study. We carefully patched our own sample of vulnerable codes and assessed the patches using known exploits.

The methodology of multiple runs employed for ChatGPT was implemented to test the lower and upper-bound capabilities of ChatGPT. As ChatGPT is an AI tool, each response is generated differently. In the first run, we did not provide information on the vulnerability present in each code and evaluated the responses as such. In the next run, we provided ChatGPT with the possible vulnerability present in the code and compiled the results. These results showed us the capabilities and changes in the behavior of ChatGPT when extra information is provided to it.

As ChatGPT bases its responses on the ongoing conversation, we utilised new chat windows for each vulnerability because ChatGPT does not retain data from previous chats. By using separate chat windows, we can focus on individual vulnerabilities and provide more accurate and relevant information in our responses. In each assessment, we copied and pasted the vulnerable code into a new chat and asked, "Does the code above contain any CWE vulnerability?". This question was selected after several trial runs with ChatGPT to yield the most effective results.

Furthermore, the same experiments were then repeated using the ChatGPT API with the May 24th version. This automated approach allows for streamlined data collection and analysis. The experiments were also extended to include ChatGPT-4, the newer version of ChatGPT, to assess any improvements in detection capabilities. The observations will be discussed in more detail in the "Results" section.

During the static code analysis, we utilised the previously mentioned environment and carefully evaluated the options provided by each analyser, selecting the appropriate option for vulnerability analysis in each case. Each vulnerability is scanned multiple times, and results are saved for further evaluation.

3.7 Evaluation Metrics

The primary evaluation metric used in this study is the detection rate, which measures the percentage of vulnerabilities correctly identified by each tool. The detection rate is calculated by comparing the tool's output with the vulnerabilities present in the datasets.

The evaluation of the effectiveness and accuracy of analysers in producing relevant results was a critical step, and it involved a thorough study of their reports. The results obtained from the analysers were meticulously organised and analysed to provide a clear understanding of the output. The formation of the results included several categories, such as mapping the vulnerabilities into OWASP classifications and maintaining separate records of the results obtained by each tool for both datasets.

Categorising the results in this manner allowed for a comprehensive evaluation of the analysers, providing a clear understanding of their strengths and weaknesses. By identifying cases where the analysers failed to produce accurate results or generated irrelevant warnings, steps can be taken to refine and improve their performance in future iterations.

To further understand the results, separate tables were created for each analysed dataset. This organisation allowed for a more in-depth examination of the results and helped identify areas of improvement or success. The categorisation of results provided valuable insight into the performance of the analysers and enabled the team to make informed decisions regarding their use and potential areas for improvement.

4. RESULTS

In this section, we evaluate and compare the efficacy of the ChatGPT and 10 static code analysers on vulnerability detection and patching by investigating vulnerability detection and patching percentages of each tool and ChatGPT. We also compare vulnerability detection rates of all the static code analysers to rates obtained by using ChatGPT. In addition, we also analyse the relationship between vulnerability detection and vulnerability categories in order to better understand possible reasons behind undetected vulnerabilities. Lastly, we studied ChatGPT's ability to patch vulnerabilities.

4.1 Overall Detection Rates

First, we determine whether ChatGPT had a noticeable impact on the vulnerability detection rates. Table 4.1 provides a comprehensive comparison of ChatGPT and static code analysis tools based on their performance on two datasets: Juliet and Unique. The combined dataset, Juliet and Unique, is also included for a complete overview. ChatGPT is presented twice in the table: once as "ChatGPT (1)" and the other as "ChatGPT (2)". ChatGPT (1) shows detection rates for the initial run of ChatGPT with all the vulnerable codes for both datasets. ChatGPT (2) shows combined detection results for the initial run and an additional run only for undetected vulnerabilities. The first run of ChatGPT achieved a 55.92% detection rate on the Juliet dataset, 64% on the Unique dataset, and an overall detection rate of 57.06% on the combined dataset. However, after undetected vulnerable codes were scanned again and the detection rate significantly increased to 58.7% on the Juliet dataset, 80% on the Unique dataset, and an overall detection rate of 70% on the combined dataset. This indicates the importance of iterative testing with ChatGPT for significantly higher performance of vulnerability detection.

Analysers	Datasets		
	Juliet (304)	Unique(50)	Combined(354)
ChatGPT(1)	170 (55.92%)	32 (64%)	202 (57.06%)
ChatGPT(2)	208 (58.7%)	40 (80%)	248 (70%)
SemGrep	48 (15.7%)	11 (22%)	59 (16.6%)
CodeFactor	3 (0.9%)	0 (0%)	3 (0.8%)
Codiga	4 (1.3%)	1 (2%)	5 (1.4%)
DeepSource	47 (15.4%)	8 (16%)	55 (15.5%)
CodeBeat	1 (0.3%)	1 (2%)	2 (0.5%)
CodeFlow	2 (0.6%)	2 (4%)	4 (1.3%)
SonarLint / JLint	14 (4.6%)	7 (14%)	21 (5.9%)
SonarQube	14 (4.6%)	9 (18%)	23 (6.5%)
FindBugs / SpotBugs	0 (0%)	5 (10%)	5 (1.41%)
PMD	1 (0.3%)	1 (2%)	2 (0.57%)
All Tools Combined	67 (22%)	23 (46%)	90 (25.4%)

Table 4.1 Comparison Based on Datasets

The table also includes other popular static code analysers like SemGrep, CodeFactor, Codiga, DeepSource, CodeBeat, CodeFlow, SonarLint/JLint, SonarQube, FindBugs/SpotBugs, and PMD. These tools have varying detection rates, with the lowest being only 0.5% and the highest being 16.6%. While some analysers have relatively low detection rates, they can still be useful when used in conjunction with other analysers.

Furthermore, we investigated whether combining all the static code analysers can provide better vulnerability detection performance compared to ChatGPT. In comparison, the combined results of all the other tools only achieve a 25.4% detection rate, significantly lower than ChatGPT's initial performance. This suggests that AI-based tools like ChatGPT have the potential to revolutionise the field of static code analysis by providing more accurate and reliable results. While the other tools may still have their strengths and weaknesses, ChatGPT's performance in this table highlights the potential benefits of incorporating AI into detecting code vulnerabilities. In Figure 4.1, the performance of the tools on each dataset is highlighted. All the tools exhibited superior performance on the unique dataset, suggesting their efficacy in addressing common vulnerabilities.

Figure 4.2 presents the percentage of vulnerabilities detected by two groups, ChatGPT and other static analysers combined, and their overlap. ChatGPT detected 187 vulnerabilities, representing 52.8% of the total detected vulnerabilities. On the other hand, the combined result of all static code analysers detected 29 vulnerabilities,

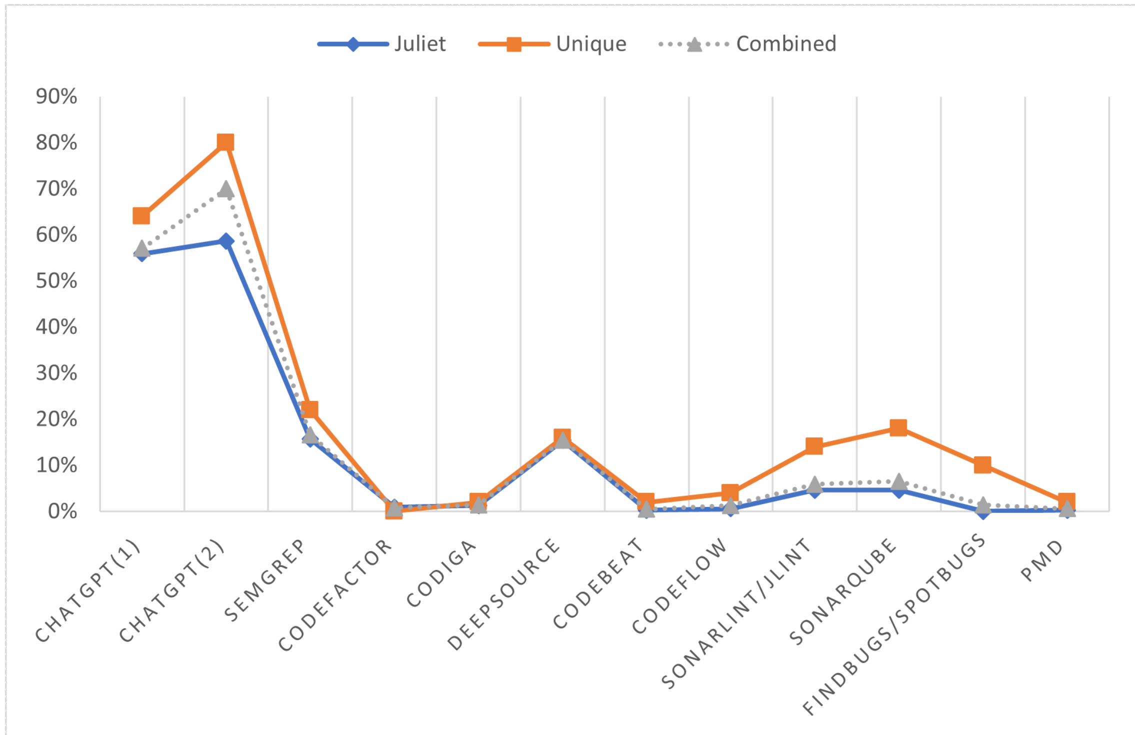

Figure 4.1 Tools' Vulnerability Detection Rates (Recalls) Based on Dataset

accounting for 8.2% of the total vulnerabilities detected. The overlap between the two circles represents the vulnerabilities detected by both groups, which is 61 and accounts for 17.2% of the total vulnerabilities detected. The total number of vulnerabilities undetected is 77, representing 21.8% of the total. These results suggest that combined static analysers can slightly contribute to identifying and mitigating some vulnerabilities.

In June 2023, we repeated experiments using API with GPT-3.5 and GPT-4. Figure 3 shows the experiments repeated using the API of GPT-3.5 and GPT-4. The May 24 version of ChatGPT was used for these experiments. We repeated our previous experiments with the new version of ChatGPT, which gave us comparable results to the previous version. The API also provided us with access to GPT-4 which gave us even better results, proving our hypothesis and conclusion from the previous experiments. Both runs merged together for GPT-4 gave us a detection rate of 90.96%.

We took the initiative to address the issue of false positives by actively patching all 354 of our vulnerable code samples. We opted for the same methodology for the evaluation of ChatGPT against patched code samples. We conducted tests using both GPT-3.5 and GPT-4 (May 24 version) using the API. The results were highly encouraging, as both models yielded negligible false positive rates. These findings demonstrate that ChatGPT, alongside its predecessor GPT-3.5, excels in minimizing

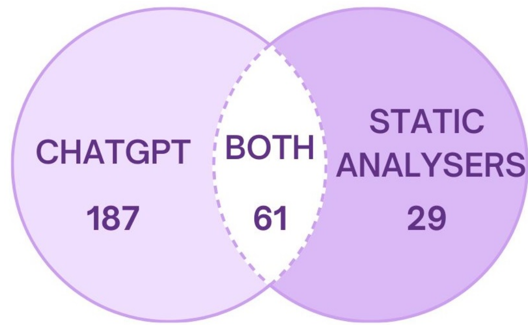

Figure 4.2 Venn diagram Showing the Number of Vulnerabilities Found

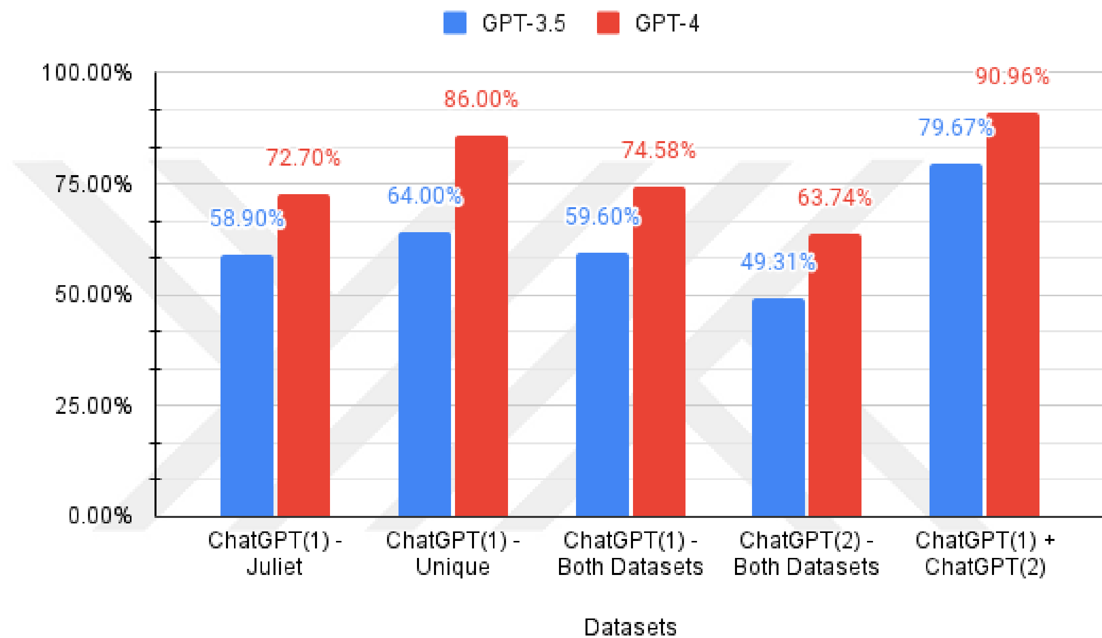

Figure 4.3 GPT-3.5 vs GPT-4 API Results

false positive generation when compared to traditional static analysis tools.

Lastly, our findings revealed that both ChatGPT and the entire range of static code analysers failed to detect the Log4j vulnerability. This vulnerability was identified after ChatGPT's training data cut-off. This situation highlights the challenge of keeping AI-based and rule-based security tools up-to-date with the latest security issues.

Figure 4.3 shows the experiments repeated using the API of ChatGPT. The May 24 version of ChatGPT was used for these experiments. We repeated our previous experiments with the new version of ChatGPT, which gave us comparable results to the previous version. The API also provided us with access to GPT-4 which gave us even better, proving our hypothesis and conclusion from the previous experiments. Both runs merged together for GPT-4 gave us a detection rate of 90.96%. We took

the initiative to address the issue of false positives by actively patching all 354 of our vulnerable code samples. To further evaluate the effectiveness of vulnerability detection, we conducted tests using both GPT-3.5 and GPT-4. The results were highly encouraging, as both models yielded negligible false positive rates. These findings demonstrate that ChatGPT, alongside its predecessor GPT-3.5, excels in minimizing false positive generation when compared to traditional static analysis tools.

4.2 Vulnerability Categories

We saw that nearly 22% of the vulnerabilities remained undetected even with ChatGPT. Some of these vulnerabilities might be harder to detect than others because of their type or categories. To understand the influence of the vulnerability type on the detection rates, we find the CWE number of each vulnerable code and then mapped to OWASP's TOP 10 Web application categories. Table 4.2 reports the number of vulnerabilities found by ChatGPT and each static code analyser per each OWASP top-10 category. Most vulnerabilities in our study were in the injection category (A3), which covers around 24% of all the vulnerabilities in our dataset. This was followed by cryptographic failures on category (A2) with 19% coverage of the combined database.

As shown in Table 4.2 and Figure 4.4, the ChatGPT detection rate was superior to other tools combined in each OWASP top 10 category. For example, ChatGPT managed to detect nearly 72% of all broken access control category, compared to 12.5% for those that belonged to SemGrep and DeepSource. The rest of the static code analysers could not find any vulnerable code belonging to the broken access control category. Similarly, in the category Security Logging and Monitoring Failures (A09), ChatGPT detected 82.3% of the vulnerabilities, while the other tools combined detected only 29.4%. Security misconfigurations (A6) and SSRF (A10) categories contain several vulnerabilities. However, ChatGPT managed to detect all the vulnerabilities. Meanwhile, SemGrep found only one vulnerability in the SSRF category among all the static code analysers.

Looking at the ChatGTP's detection rates for each category, ChatGTP had the lowest detection rate (59%) for the vulnerable and outdated components category. Surprisingly, all other tools combined had a detection rate of 45.5% for the same cat-

OWASP	#	ChatGPT(2)	SemGrep	CodeFactor	Codiga
A01	32	23 (71.9%)	4 (12.5%)	0 (0%)	0 (0%)
A02	68	49 (72.0%)	7 (10.2%)	1 (1.4%)	3 (4.4%)
A03	84	60 (71.4%)	14 (16.6%)	1 (1.1%)	0 (0%)
A04	41	28 (68.3%)	4 (9.7%)	0 (0%)	0 (0%)
A05	22	13 (59.1%)	10 (45.4%)	0 (0%)	0 (0%)
A06	1	1 (100%)	0 (0%)	0 (0%)	0 (0%)
A07	25	18 (72%)	7 (28%)	0 (0%)	0 (0%)
A08	62	40 (64.5%)	7 (11.2%)	1 (1.6%)	2 (3.2%)
A09	17	14 (82.3%)	5 (29.4%)	0 (0%)	0 (0%)
A10	2	2 (100%)	1 (50%)	0 (0%)	0 (0%)

OWASP	#	DeepSource	CodeBeat	CodeFlow	SonarLint / JLint
A01	32	4 (12.5%)	0 (0%)	0 (0%)	0 (0%)
A02	68	9 (13.2%)	0 (0%)	2 (2.9%)	9 (13.2%)
A03	84	14 (16.6%)	0 (0%)	1 (1.1%)	2 (2.3%)
A04	41	5 (12.1%)	0 (0%)	0 (0%)	4 (9.7%)
A05	22	10 (45.4%)	0 (0%)	0 (0%)	0 (0%)
A06	1	0 (0%)	0 (0%)	0 (0%)	0 (0%)
A07	25	5 (20%)	1 (4%)	0 (0%)	0 (0%)
A08	62	6 (9.6%)	1 (1.6%)	1 (1.6%)	6 (9.6%)
A09	17	2 (11.7%)	0 (0%)	0 (0%)	0 (0%)
A10	2	0 (0%)	0 (0%)	0 (0%)	0 (0%)

OWASP	#	SonarQube	FindBugs / SpotBugs	PMD	Other Tools (Combined)
A01	32	0 (0%)	0 (0%)	0 (0%)	4 (12.5%)
A02	68	9 (13.2%)	2 (2.9%)	1 (1.4%)	19 (28%)
A03	84	2 (2.3%)	0 (0%)	0 (0%)	18 (21.4%)
A04	41	4 (9.7%)	0 (0%)	0 (0%)	9 (22%)
A05	22	0 (0%)	0 (0%)	0 (0%)	10 (45.4%)
A06	1	0 (0%)	0 (0%)	0 (0%)	0 (0%)
A07	25	2 (8%)	2 (8%)	0 (0%)	11 (44%)
A08	62	6 (9.6%)	1 (1.6%)	1 (1.6%)	13 (21%)
A09	17	0 (0%)	0 (0%)	0 (0%)	5 (29.4%)
A10	2	0 (0%)	0 (0%)	0 (0%)	1 (50%)

Table 4.2 Efficiency of Each Tool

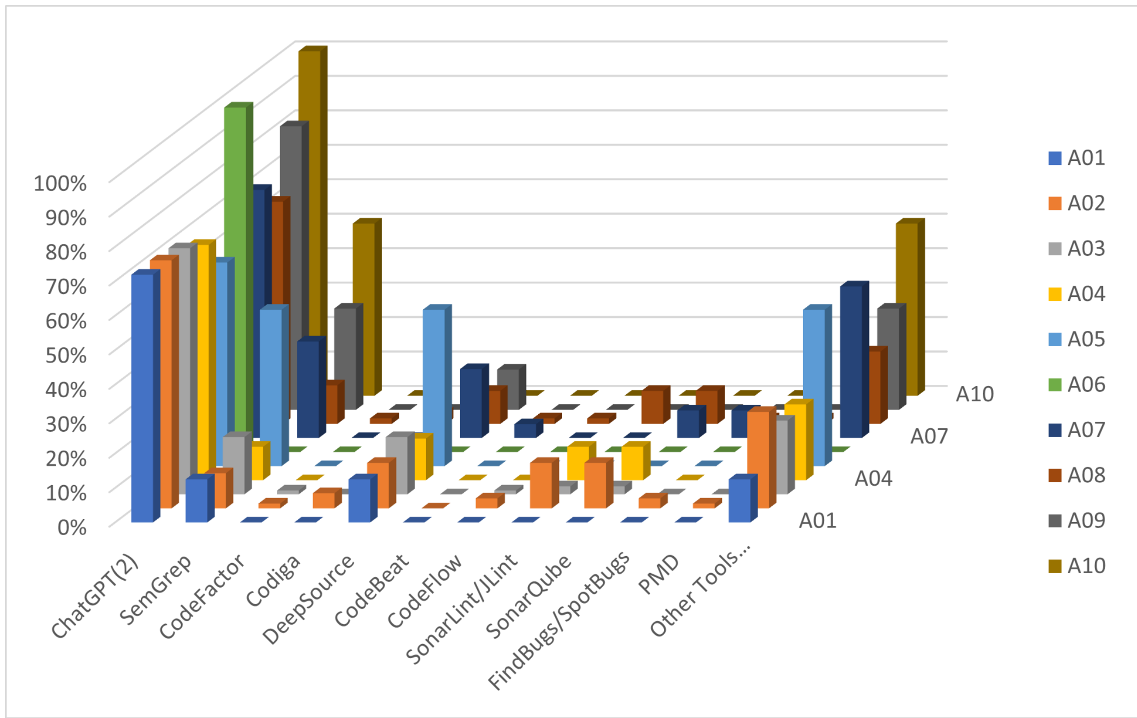

Figure 4.4 OWASP TOP 10 Vulnerabilities Detection Rates (Recalls) for Each Tool

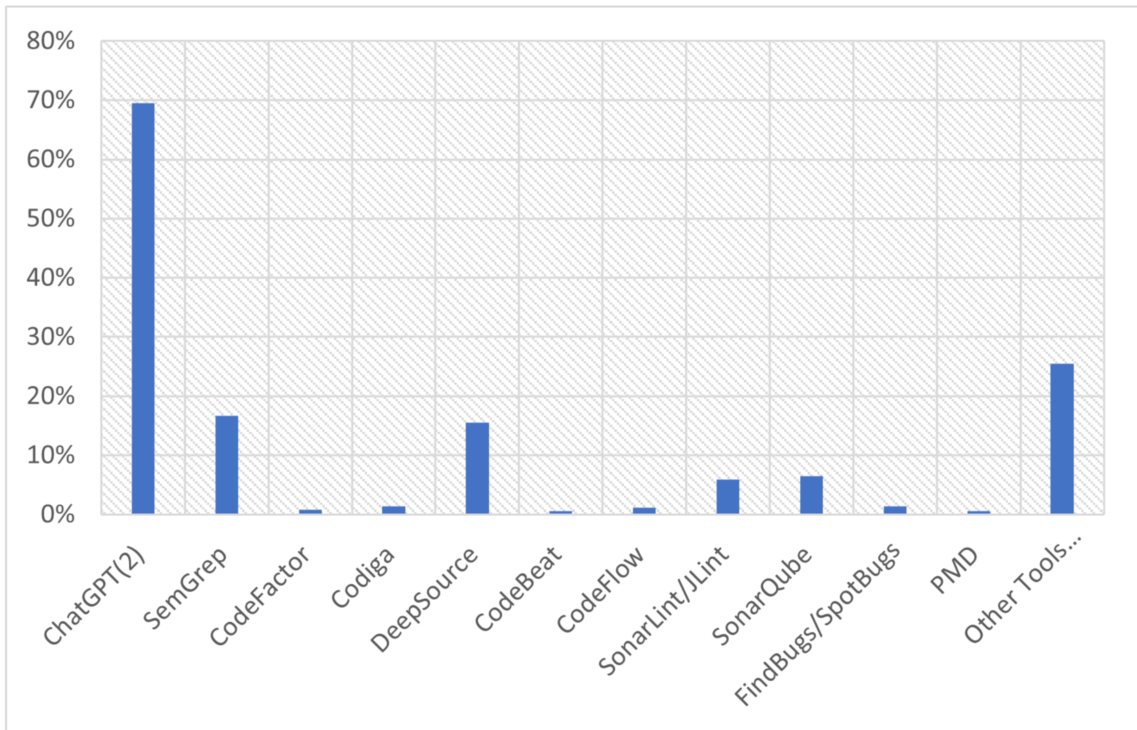

Figure 4.5 OWASP TOP 10 Vulnerabilities Detection Rates (Recalls) for Each Tool

OWASP	Total	Only Tools	Only ChatGPT	Overlap
A01	32	1 (3.1%)	20 (62.5%)	3 (9.3%)
A02	68	6 (8.8%)	36 (52.9%)	13 (19.1%)
A03	84	5 (5.9%)	47 (55.9%)	13 (15.4%)
A04	41	4 (9.7%)	23 (56.1%)	5 (12.2%)
A05	22	3 (13.6%)	6 (27.2%)	7 (31.8%)
A06	1	0 (0%)	1 (100%)	0 (0%)
A07	25	3 (12%)	10 (40%)	8 (32%)
A08	62	6 (9.6%)	33 (53.2%)	7 (11.2%)
A09	17	1 (5.8%)	10 (58.8%)	4 (23.5%)
A10	2	0 (0%)	1 (50%)	1 (50%)

Table 4.3 Comparison of Vulnerability Detection Rates between Only Tools and Only ChatGPT

egory, which was the highest rate for a single category when all other tool's detection rates were combined. These suggest that different tools may have varying levels of effectiveness in detecting vulnerabilities across different categories. Figure 4.5 presents the overall detection rates for ChatGPT and static code analysers about OWASP TOP10 vulnerabilities. ChatGPT outperforms all other tools, demonstrating its superiority in vulnerability detection.

Table 4.3 shows vulnerabilities solely found by ChatGPT and other tools and their overlap. ChatGPT alone has the highest detection rates for all OWASP categories. However, the results also indicate that using ChatGPT alone may not be enough, as other tools can find various vulnerabilities that ChatGPT cannot capture. For example, in category A01, other tools were able to detect one broken access control vulnerability that ChatGPT failed to detect, while other tools were successful in identifying it. This particular vulnerability had the potential to lead to a data breach. Moreover, in categories A02, A03, A04, A05, A07, and A08, the number of vulnerabilities exclusively detected by static code analysers was higher compared to the results obtained in category A01, implying that combining the outputs of different tools can lead to more comprehensive vulnerability detection.

Moreover, Table 4.4 presents information on the categories of CWEs that were primarily detected by static code analysers, and it compares the vulnerability detection rates of static code analysers and ChatGPT. The table includes 11 CWE categories; in all of them, the vulnerability detection rate of static code analysers exceeds that of ChatGPT. This suggests that, for these specific CWE categories, static code analysis is a more effective method for detecting code vulnerabilities than ChatGPT. For one of the CWE categories, CWE 384, static code analysers could detect all of the vulnerabilities, while ChatGPT could not detect any. In the remaining 10 CWE cat-

CWE	OWASP	#	Only ChatGPT	Only Tools
80	A3	3	0	1
325	A2	3	0	1
384	A7	2	0	2
398	A8	3	0	1
459	A7	3	0	1
478	A2	3	0	1
484	A3	3	0	1
500	A2	1	0	1
563	A1	3	1	2
582	A2	1	0	1
698	A4	3	0	1

Table 4.4 The categories of CWEs that were primarily detected by Static code Analysers

egories, the detection rate between static code analysers and ChatGPT was lower, with static code analysers detecting more vulnerabilities than ChatGPT.

In conclusion, we observed that ChatGPT has detected a higher percentage of vulnerabilities compared to the other tools combined in all categories. ChatGPT has proven to be an effective tool for detecting vulnerabilities, but more is needed. Other tools should also be used with ChatGPT to increase the likelihood of capturing a wider range of vulnerabilities.

4.3 Efficacy of Vulnerability Patching

Unlike static code analysers, ChatGPT also offers fixes and patches for some vulnerable codes. We only ask a question regarding vulnerability detection. However, ChatGPT returns a fixed code in some cases. In this section, we investigate ChatGPT's vulnerability fix rate and accuracy.

Table 4.5 displays the results of vulnerability identification as well as the accuracy of the fixes made for both ChatGPT runs. The table shows that a high percentage of issues were found and fixed in both runs, with accuracy of fixes being 100% in both cases. The table shows that 57% of issues were found and 17.51% were fixed, with a fix accuracy of 100% in the first run. During the second round of testing, a total of 152 vulnerabilities that had gone unnoticed in the previous run were reevaluated.

Phase	Total	Found	Fixed (without prompting)	Fix Accuracy
ChatGPT(1)	354	202 (57%)	62 (17.5%)	100%
ChatGPT(2)	152	46 (13%)	10 (2.8%)	100%
Combined	354	248 (70%)	72 (20.3%)	100%

Table 4.5 ChatGPT Phase and Fix Results

As a result, an additional 13% of vulnerabilities were discovered, and 2.8% of them were successfully resolved. It is noteworthy that the accuracy of the fixes remained at 100%. When we combined the results the percentage of vulnerabilities found and fixed was 70% and 20.3%, respectively, with 100% fix accuracy. These results demonstrated ChatGPT can provide accurate fixes to vulnerabilities, once they are identified correctly. This provides additional advantage over other approaches and tools that may not have achieved such high percentages in identifying and fixing vulnerabilities, and highlights the potential usefulness of ChatGPT in similar contexts.

4.4 False Positive Generation Analysis for Patched Codes

We took the initiative to address the issue of false positives by actively patching all 354 of our vulnerable code samples. To further evaluate the effectiveness of vulnerability detection, we conducted tests using both GPT-3.5 and GPT-4. The results were highly encouraging, as both models yielded negligible false positive rates. These findings demonstrate that ChatGPT, alongside its predecessor GPT-3.5, excels in minimising false positive generation when compared to traditional static analysis tools.

5. DISCUSSION

In this section, we provide a brief statement of the primary outcomes and potential issues and directions for future research that can be pursued based on these findings.

5.1 Effectiveness of Vulnerability Detection

Static analysis tools play a crucial role in ensuring the security of software systems, but their effectiveness is often limited. Our research showed that despite their potential, many static analysis tools are incapable of detecting many commonly known major and critical vulnerabilities. On the other hand, ChatGPT surpasses the collective detection capability of all static code analysers, as it can identify over 57% of issues in the initial scan, while the combined results of static code analysers only provide a maximum of 25% detection. Moreover, the detection rate increased to 70% in ChatGPT's second run. Furthermore, our experiments repeated on GPT-4 took the detection rate to 90%. Our research has shown that ChatGPT can be used as a state-of-the-art static analysis tool, providing reliable and accurate results in detecting software vulnerabilities. Moreover, static code analysers should not be disregarded for now, as they are still capable of identifying vulnerabilities that may elude detection by the ChatGPT. However, as AI-based tools like ChatGPT continue to advance and enhance their vulnerability detection capabilities, they may eventually render the use of rule-based static code analysers obsolete, making way for more efficient and accurate security practices.

5.2 Privacy Issues and Concerns

This research represents a significant contribution to the field of software security, as it marks the first evaluation of ChatGPT's effectiveness in detecting software vulnerabilities on a large scale. The results of our analysis demonstrate a significant improvement in detection rates compared to other traditional static analysis tools. However, several privacy and security concerns might cause developers to avoid using ChatGPT for vulnerability detection and fixing. While using ChatGPT, developers can provide various sensitive information that can reveal the organisation name, their product ideas, as well as the source code of their product and sensitive information inside of it, such as API keys, passwords or other hard-coded information. For example, a Redis bug in ChatGPT system revealed personal information of the paying customers [44].

OpenAI stated that data from ChatGPT web users might be utilized to enhance their models[45]. Users can prevent this by adjusting their settings. After opting out, new conversations will not contribute to model training. Furthermore, OpenAI asserts that data from the API and playground is not used for model training. Conversely, OpenAI also noted that the system keeps API inputs and outputs for a duration of 30 days to detect misuse[46]. These actions and previous compromises raise suspicions in large organizations as a result of this they prefer blocking ChatGPT during business communications as they might leak sensitive information about the organization. This will also result in blocking or disliking products that use ChatGPT in their background. To reduce this concern, ChatGPT can create a trusted client list where they don't store any of the data coming and going to their clients. This would ease some of these companies' concerns about privacy. Alternatively, a local version can be offered to be executed on the concerned organizations' servers. This can be arranged in different versions of ChatGPT.

Despite the OpenAI's apparent disregard for the significant privacy implications of its system, the company may soon be compelled to take corrective measures due to the regulatory requirements imposed by privacy laws in the EU and other jurisdictions. For instance, much of the data that ChatGPT has processed is likely to fall under the purview of the EU's General Data Protection Regulation (GDPR), which imposes strict limitations on the permissible uses of personal data that go beyond the purposes for which it was originally collected. While acknowledging the scope for further research in multiple directions, one possible area for future investigation is examining the potential privacy risks and associated consequences.

OpenAI[47], the organisation behind ChatGPT, has mentioned that it does not use the data provided by the users for training or improving the models and sharing with its partners. However, it still retains the data for 30 days for monitoring abuse and misuse. This data retention can be a big concern whether we are free to communicate any data with ChatGPT or not. It uses a channel on Transport Layer and customer-to-OpenAI requests and responses are encrypted.

5.3 Usability and Other Issues

We found out that ChatGPT is one of the most effective free solutions for vulnerability detection and patching. However, we test the tool by providing relatively small Java code pieces. While testing, we input the entire code into ChatGPT and evaluated the output. In large software products, detecting vulnerabilities with this approach will consume more time than static code analyser scans as code analysers are designed to scan entire software projects. The latest version of ChatGPT can only accept code from input fields where the whole project can not be fitted. This could discourage developers from scanning the entire project where the remaining code could have vulnerabilities.

Another issue is related to ChatGPT's output. Typically, static code analysers indicate potential vulnerabilities through the warnings they generate. However, these warnings may not always use concise language because they don't want to mislead developers when there is a false positive. Meanwhile, ChatGPT's output utilises a more conclusive and precise language even when there is false positives or false negatives. This output could easily mislead developers into thinking that there is an issue when there is none or create false sense of security when there is a vulnerability in the system.

Lastly, ChatGPT is trained using extensive text data collection, which may include smelly codes, bad coding practices, bugs and vulnerabilities. As a result, the ChatGPT might occasionally produce responses that unintentionally exhibit bias.

5.4 Threats to Validity

The use of off-the-shelf freeware or free versions of paid tools in this research represents a potential challenge to the validity of our findings. While these tools were used for the purposes of this study, it is possible that using commercial tools with more robust analysis capabilities could have produced different results. This underscores the importance of conducting additional research using a diverse range of tools, including both freeware and commercial solutions, to fully assess the accuracy and reliability of static analysis tools in detecting vulnerabilities.

The scope of vulnerable codes tested in this research may impact the validity of our findings. By evaluating the performance of the static analysis tools using only a limited set of vulnerable codes, it is possible that our results may not fully reflect their abilities in detecting security flaws in real-world software systems. To address this, future studies should consider incorporating a broader range of vulnerable codes in their analysis to better represent the diversity of security risks faced by software systems.

While this study offers insights into the capabilities of ChatGPT, it is important to note that the performance analysis is limited to the detection of vulnerabilities present in the training data available up to the point of the ChatGPT model's cutoff date. The research does not address dynamic analysis techniques or cover vulnerabilities introduced after the model's training period.

5.5 Enhancing ChatGPT for Static Analysis

As our research shows that ChatGPT gives us much better results in the identification of vulnerabilities in the JAVA language as compared with traditional static analysis tools, we wanted to research how we can further enhance the capabilities of ChatGPT for static analysis. As discussed in the introduction of ChatGPT, it has not been specifically designed for cybersecurity-related tasks. This shows that it would need multiple algorithmic enhancements, data improvements, and fine-tuning strategies to increase its capabilities in terms of static analysis.

ChatGPT should be trained on a specialized dataset that would be specifically focused on static analysis-related problems. This would include a large set of vulnerable codes having different vulnerabilities, coding styles, and patterns. Apart from that, the model of ChatGPT should be fine-tuned for a better understanding of the syntax and semantics of programming languages. Different techniques like Abstract Syntax Trees (ASTs) or Code2Vec should be incorporated into the model. The model will need to be regularly updated to take in recent vulnerabilities that previous models have not seen.

To be truly effective, ChatGPT should seamlessly integrate with popular integrated development environments (IDEs) and static analysis tools, providing real-time feedback and assistance to developers. Regular updates and continuous learning are essential to keep the model up-to-date with the latest coding practices, languages, and security threats.

Our results showed 29 vulnerabilities that the static code analysis tools detected but ChatGPT was not able to detect them. Creating an application that can analyze the code through the static analysis tools and ChatGPT simultaneously and in real time would be extremely beneficial for the detection of vulnerabilities in the development phase. We can use a version of ChatGPT specifically trained for static analysis and merge it with other selected tools, that gave comparatively better results, into one stand-alone application. This application can keep track of the responses from all the tools incorporated in it, and provide the user with recommendations and explanations in real-time.

6. CONCLUSION AND FUTURE WORK

In this study, we explored the effectiveness of ChatGPT and ten static code analysers in terms of detecting and fixing 354 Java Vulnerabilities. We found out that ChatGPT achieved much higher vulnerability detection rates than all other static code analysers combined. On top of that, we observed that each previously undetected vulnerability can be detected with new attempts. This further increases the effectiveness of ChatGPT. We also discovered that neither ChatGPT nor any of the static code analysers were able to identify the Log4j vulnerability. This highlights a significant challenge faced by AI-based and rule-based security tools, as they can be particularly ineffective in uncovering new and unseen vulnerabilities.

Moreover, we discover that around 8% of these vulnerabilities are only detected by the static code analysers, while ChatGPT detects the majority.

As a result, security practitioners should still pay attention to rule-based static code analysers at present since they can still detect security weaknesses that might escape the notice of ChatGPT. Nonetheless, as these AI-driven systems become more sophisticated and effective in detecting security flaws, the reliance on rule-based static code analysers is expected to diminish rapidly, ultimately leading to their obsolescence.

In addition to that finding, we also observe that ChatGPT can be used to fix the vulnerabilities. Unlike static code analysers, the nondeterministic nature of ChatGPT also returns very accurate patches that can be applied immediately. Our results showed that ChatGPT could provide very 100% accurate fixes for 20% of the vulnerabilities even when users did not ask. We present evidence that ChatGPT can be used to identify the vulnerability inside the code and provide a solution. More studies can be conducted to observe the effectiveness of this mechanism.

In the light of the encouraging results, we urge the security community to conduct more comprehensive experiments with AI-powered tools to identify vulnerabilities and address their usability and privacy concerns.

BIBLIOGRAPHY

[1] GRAMMATECH, "The role of static analysis in a secure software development life cycle (sdlc)." https://blogs.grammatech.com/the-role-of-static-analysis-in-a-secure-software-development-lifecycle, 2022. (Accessed on 03/08/2023).

[2] "What is cybersecurity? everything you need to know | techtarget." https://www.techtarget.com/searchsecurity/definition/cybersecurity. (Accessed on 03/08/2023).

[3] G. B. Imbugwa, L. J. P. de Araújo, M. Khazeev, E. Enombe, H. Saliu, and M. Mazzara, "A case study comparing static analysis tools for evaluating swiftui projects," in *Journal of Physics: Conference Series*, vol. 2134, p. 012022, IOP Publishing, 2021.

[4] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies? an empirical study on the impact of security advisories on library migration," *Empirical Software Engineering*, vol. 23, pp. 384–417, 2018.

[5] C. Chahar, V. S. Chauhan, and M. L. Das, "Code analysis for software and system security using open source tools," *Information Security Journal: A Global Perspective*, vol. 21, no. 6, pp. 346–352, 2012.

[6] A. G. Bardas *et al.*, "Static code analysis," *Journal of Information Systems & Operations Management*, vol. 4, no. 2, pp. 99–107, 2010.

[7] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," in *WODA 2003: ICSE Workshop on Dynamic Analysis*, pp. 24–27, 2003.

[8] N. Imtiaz, S. Thorn, and L. Williams, "A comparative study of vulnerability reporting by software composition analysis tools," in *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1–11, 2021.

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

[10] A. Walker, M. Coffey, P. Tisnovsky, and T. Cerny, "On limitations of modern static analysis tools," in *Information Science and Applications*, pp. 577–586, Springer, 2020.

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

[12] N. Rutar, C. B. Almazan, and J. S. Foster, "A comparison of bug finding tools for java," in *15th International symposium on software reliability engineering*, pp. 245–256, IEEE, 2004.

[13] V. Lenarduzzi, F. Pecorelli, N. Saarimaki, S. Lujan, and F. Palomba, "A critical comparison on six static analysis tools: Detection, agreement, and precision," *Journal of Systems and Software*, vol. 198, p. 111575, 2023.

[14] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 672–681, IEEE, 2013.

[15] Y. Movahedi, M. Cukier, and I. Gashi, "Vulnerability prediction capability: A comparison between vulnerability discovery models and neural network models," *Computers & Security*, vol. 87, p. 101596, 2019.

[16] C. Theisen and L. Williams, "Better together: Comparing vulnerability prediction models," *Information and Software Technology*, vol. 119, p. 106204, 2020.

[17] R. Halepmollası, K. Hanifi, R. F. Fouladi, and A. Tosun, "A comparison of source code representation methods to predict vulnerability inducing code changes.," in *ENASE*, p. In Press, 2023.

[18] T. Coskun, R. Halepmollasi, K. Hanifi, R. F. Fouladi, P. C. De Cnudde, and A. Tosun, "Profiling developers to predict vulnerable code changes," in *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, pp. 32–41, 2022.

[19] K. Hanifi, R. F. Fouladi, B. G. Unsalver, and G. Karadag, "Software vulnerability prediction knowledge transferring between programming languages," *arXiv preprint arXiv:2303.06177*, 2023.

[20] K. Chowdhary and K. Chowdhary, "Natural language processing," *Fundamentals of artificial intelligence*, pp. 603–649, 2020.

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[22] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 292–303, 2021.

[23] M. Fu and C. Tantithamthavorn, "Linevul: a transformer-based line-level vulnerability prediction," in *Proceedings of the 19th International Conference on Mining Software Repositories*, pp. 608–620, 2022.

[24] D. Hin, A. Kan, H. Chen, and M. A. Babar, "Linevd: statement-level vulnerability detection using graph neural networks," in *Proceedings of the 19th International Conference on Mining Software Repositories*, pp. 596–607, 2022.

[25] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, *et al.*, "Training language models to follow instructions with human feedback," *arXiv preprint arXiv:2203.02155*, 2022.

[26] "Evaluation of chatgpt model for vulnerability detection." https://arxiv.org/abs/2304.07232. (Accessed on 07/14/2023).

[27] O. S. Ozturk, E. Ekmekcioglu, O. Cetin, B. Arief, and J. Hernandez-Castro, "New tricks to old codes: Can ai chatbots replace static code analysis tools?," EICC '23, (New York, NY, USA), p. 13–18, Association for Computing Machinery, 2023.

[28] T. Boland and P. E. Black, "Juliet 1. 1 c/c++ and java test suite," *Computer*, vol. 45, no. 10, pp. 88–90, 2012.

[29] "Top 25 software errors | sans institute." https://www.sans.org/top25-software-errors/. (Accessed on 03/01/2023).

[30] "Cve." https://www.cve.org/. (Accessed on 03/08/2023).

[31] "Cwe - 2022 cwe top 25 most dangerous software weaknesses." https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html. (Accessed on 03/08/2023).

[32] "Owasp top ten | owasp foundation." https://owasp.org/www-project-top-ten/. (Accessed on 03/01/2023).

[33] CodeFlow, "Automated testing tool." https://www.getcodeflow.com/, 2022.

[34] "Semgrep — find bugs and enforce code standards." https://semgrep.dev/. (Accessed on 03/08/2023).

[35] "Docs Home." https://semgrep.dev/docs/, 2023.

[36] "Codebeat - automated code review for mobile and web." https://codebeat.co/. (Accessed on 03/01/2023).

[37] "Codiga: Static code analysis in real-time." https://www.codiga.io/. (Accessed on 03/01/2023).

[38] "Deepsource: The code health platform." https://deepsource.io/. (Accessed on 03/01/2023).

[39] "Pmd." https://pmd.github.io/. (Accessed on 03/08/2023).

[40] "Findbugs™ - find bugs in java programs." https://findbugs.sourceforge.net/. (Accessed on 03/08/2023).

[41] SonarQube, "Java." https://docs.sonarqube.org/latest/analyzing-source-code/languages/java/. (Accessed on 03/08/2023).

[42] "Jlint - find bugs in java programs." https://jlint.sourceforge.net/. (Accessed on 03/24/2023).

[43] L. Floridi and M. Chiriatti, "Gpt-3: Its nature, scope, limits, and consequences," *Minds and Machines*, vol. 30, pp. 681–694, 2020.

[44] "Chatgpt bug leaked payment data, conversation titles of users, confirms openai | mint." https://www.livemint.com/technology/apps/chatgpt-bug-leaked-payment-data-conversation-titles-of-users-confirms.html. (Accessed on 09/18/2023).

[45] "How your data is used to improve model performance | openai help center." https://help.openai.com/en/articles/5722486-how-your-data-is-used-to-improve-model-performance. (Accessed on 09/18/2023).

[46] "Enterprise privacy." https://openai.com/enterprise-privacy. (Accessed on 09/18/2023).

[47] "Api data usage policies." https://openai.com/policies/api-data-usage-policies. (Accessed on 06/09/2023).