LLM vs. SAST: A Technical Analysis on Detecting Coding Bugs of GPT4-Advanced Data Analysis

Madjid G. Tehrani [1], Eldar Sultanow [3], William J. Buchanan [2], Mahkame Houmani [1], Christel H. Djaha Fodja [1]

[1] The George Washington University, Washington DC, USA
[2] Blockpass ID Lab, Edinburgh Napier University, Edinburgh, UK
[3] Capgemini Deutschland GmbH, Nuremberg, Germany

Abstract. With the rapid advancements in Natural Language Processing (NLP), large language models (LLMs) like GPT-4 have gained significant traction in diverse applications, including security vulnerability scanning. This paper investigates the efficacy of GPT-4 in identifying software vulnerabilities compared to traditional Static Application Security Testing (SAST) tools. Drawing from an array of security mistakes, our analysis underscores the potent capabilities of GPT-4 in LLM-enhanced vulnerability scanning. We unveiled that GPT-4 (Advanced Data Analysis) outperforms SAST by an accuracy of 94% in detecting 32 types of exploitable vulnerabilities. This study also addresses the potential security concerns surrounding LLMs, emphasising the imperative of security by design/default and other security best practices for AI.

1 Introduction

One of the most significant threats within cybersecurity is the application of zero-day threats. These tend to be caused by unforeseen vulnerabilities in code and which can open up significant weaknesses within our society. For example, Heartbleed [1] was one significant example of this, and where the OpenSSL software package allowed an adversary to capture part of the running memory on a server, and which could reveal passwords and cryptographic keys. And, so, while many developers are diligent in their approach, there are still many cases of code being released without a strong focus on cybersecurity testing. A well-defined trait is that software developers often have an eye for usability and in producing workable code rather than for it to be safe from future vulnerabilities [2].

This paper aims to understand if machine-learning-aided testing for cybersecurity could enhance our existing static approaches to software testing. A core contribution of the paper is in the testing of the accuracy performance of the machine-learning approach of GPT-4 against the usage of static testing tools, along with a well-defined methodology that can be used in the assessment for the accuracy of bug finding. It uses a sampling-based approach to code examples and focuses on well-defined classifications of software bugs as defined within the MITRE ATT&CK™ framework.

The remainder of this paper is organised as follows. Related work (section 2) will present past work on this topic. Then, we will evaluate the hypothesis that GPT-4 identifies vulnerabilities more effectively than SAST tools through the methodology and security scenario (Section 3). Section 4 presents the results. Section 5 presents the discussion and future research. Sections 6 and 7 present the limitations and conclusion.

1.1 Background

Machine learning in code analysis for cybersecurity has been elaborated very well [3]. Recent progress in natural language processing (NLP) has given rise to potent language models like the GPT (Generative Pre-trained Transformer) series, encompassing large language models (LLM) like ChatGPT and GPT-4 [4]. Traditionally, Static Application Security Testing (SAST) is a method that employs Static Code Analysis (SCA) to detect possible security vulnerabilities. We are interested in seeing whether SAST or GPT could be more efficient in decreasing the window of vulnerability. The window of vulnerability is defined as when the most vulnerable systems apply the patch minus the time an exploit becomes active. The precondition is met if two milestones that assume the detection of vulnerabilities verify their effectiveness, along with the vendor patch [5].

Laws in some countries, like China, ban the reporting on zero-days (see articles 4 and 9 of [6]), and contests like the Tianfu Cup [7], which is a systematic effort to find zero days, proliferate zero-day discovery continuously. Therefore, this precondition may not be satisfied in a timely manner, especially if the confirmation of vulnerabilities is not verified. A wide window of vulnerability threatens national security if a zero-day is exploited against critical infrastructures. DARPA introduces an important challenge that may help overcome this threat: (AIxCC) [8]). Moreover, this topic touches a part of the BSI studies [9, 10], and where we can define two main classifications of software testing for cybersecurity bugs as:

- Static Application Security Testing (SAST). This is often referred to as White Box Testing, which is a set of algorithms and techniques used for analysing source code. It operates automatically in a non-runtime environment to detect vulnerabilities such as hidden errors or poor source code during development.
- Dynamic Application Security Testing (DAST). This follows the opposite approach and analyses the program while it is operating. Functions are called with values in the variables as each line of code is checked, and possible branching scenarios are guessed. Currently, GPT-4 and other LLMs can not provide DAST capabilities because the code needs to run within the runtime for this to work, requiring many deployment considerations.

2 Related Work

Dominik Sobania et al. [11] explored automated program repair techniques, specifically focusing on ChatGPT's potential for bug fixing. According to them, while initially not designed for this purpose, ChatGPT demonstrated promising results on the QuixBugs benchmark, rivalling advanced methods like CoCoNut and Codex. ChatGPT's interactive dialogue system uniquely enhances its repair rate, outperforming established standards. Wei Ma et al. [12] noted that while ChatGPT shows impressive potential in software engineering(SE) tasks like code and document generation, its lack of interpretability raises concerns given SE's high-reliability requirements. Through a detailed study, they categorised AI's essential skills for SE into syntax understanding, static behaviour understanding, and dynamic behaviour understanding. Their assessment, spanning languages like C, Java, Python, and Solidity, revealed that ChatGPT excels in syntax understanding (akin to an AST parser) but faces challenges in comprehending dynamic semantics. The study also found ChatGPT prone to hallucinations, emphasising the need to validate its outputs for SE dependability and suggesting that codes from LLMs are syntactically correct but potentially vulnerable.

Haonan Li et al. [13] discussed the challenges of balancing precision and scalability in static analysis for identifying software bugs. While LLMs show potential in understanding and debugging code, their efficacy in handling complex bug logic, which often requires intricate reasoning and broad analysis, remains limited. Therefore, the researchers suggest using LLMs to assist rather than replace static analysis. Their study introduced LLift, an automated system combining a static analysis tool and an LLM to address use-before-initialisation (UBI) bugs. Despite various challenges like bug-specific modelling and the unpredictability of LLMs, LLift, when tested on real-world potential UBI bugs, showed significant precision (50%(and recall (100%). Notably, it uncovered 13 new UBI bugs in the Linux kernel, highlighting the potential of LLM-assisted methods in extensive real-world bug detection.

Norbert Tihani et al. [14] introduced the FormAI dataset, comprising 112,000 AI-generated C programs with vulnerability classifications generated by GPT-3.5-turbo. These programs range from complex tasks like network management and encryption to simpler ones, like string operations. Each program comes labelled with the identified vulnerabilities, pinpointing the type, line number, and vulnerable function. To achieve accurate vulnerability detection without false positives, the Efficient SMT-based Bounded Model Checker (ESBMC) was used. This method leverages techniques like model checking and constraint programming to reason over program safety. Each vulnerability also references its corresponding Common Weakness Enumeration (CWE) number.

Codex, introduced by Mark et al. [15], represents a significant advancement in GPT language models, tailored specifically for code synthesis using data from GitHub. This refined model underpins the operations of GitHub Copilot. When assessed on the HumanEval dataset, designed to gauge the functional accuracy of generating programs based on docstrings, Codex achieved a remarkable 28.8% success rate. In stark contrast, GPT-3 yielded a 0% success rate, and GPT-J achieved 11.4%. A standout discovery was the model's enhanced performance through repeated sampling, with a success rate soaring to 70.2% when given 100 samples per problem. Despite these promising results, Codex does exhibit certain limitations, notably struggling with intricate docstrings and variable binding operations. The paper deliberates on the broader ramifications of deploying such potent code-generation tools, touching upon safety, security, and economic implications.

In a technical evaluation, Cheshkov et al. [16] found that the ChatGPT and GPT-3 models, despite their success in various other code-based tasks, performed on par with a dummy classifier for this particular challenge. Utilising a dataset of Java files sourced from GitHub repositories, the study emphasised the models' current limitations in the domain of vulnerability detection. However, the authors remain optimistic about the potential of future advancements, suggesting that models like GPT-4, with targeted research, could eventually make significant contributions to the field of vulnerability detection.

A comprehensive study conducted by Xin Liu et al. [17] investigated the potential of ChatGPT in Vulnerability Description Mapping (VDM) tasks. VDM is pivotal in efficiently mapping vulnerabilities to CWE and MITRE ATT&CK Techniques classifications. Their findings suggest that while ChatGPT approaches the proficiency of human experts in the Vulnerability-to-CWE task, especially with high-quality public data, its performance is notably compromised in tasks such as Vulnerability-to-ATT&CK, particularly when reliant on suboptimal public data quality. Ultimately, Xin Liu et al. emphasise that, despite the promise shown by ChatGPT, it is not yet poised to replace the critical expertise of professional security engineers, asserting that closed-source LLMs are not the conclusive answer for VDM tasks. Last but not least, the OWASP top 10 for LLMs [18] introduced ten secu-

rity risks as follows: Prompt Injection, Insecure Output Handling, Training Data Poisoning, Model Denial of Service, Supply Chain Vulnerabilities, Sensitive Information Disclosure, Insecure Plugin Design, Excessive Agency, Over reliance, and Model Theft.

3 Methodology and security scenarios

3.1 Experiment Design and Data

We selected two different Static Application Security Testing (SAST) tools to ensure a fair comparison. The first is SonarQube [19], a well-established open-source platform that has been in maintenance since 2006. SonarQube supports 29 languages and offers continuous inspection of code quality. It conducts automatic reviews through static analysis to detect bugs, code smells, and other issues across the 29 supported languages. The platform provides insights on duplicated code, coding standards, unit tests, code coverage, code complexity, comments, bugs, and security vulnerabilities.

On the other hand, the second tool is a relatively new, paid Software-as-a-Service (SaaS) that began operations in 2020, named Cloud Defence [20]. Its mission is "to shield Cloud Native Applications from Zero Day Attacks." To derive a comprehensive evaluation, we combined the outcomes of both tools using an 'OR' operation and named this consolidated result the "SAST result." Consequently, a positive outcome (indicated by a '1') in the SAST result signifies that either of the tools successfully detected the security vulnerability.

For this study, we selected 32 known security pitfalls that developers might inadvertently introduce, potentially leading to zero-day vulnerabilities. Based on our observations, we formulated the following hypotheses:

- H_0: GPT-4-Advanced Data Analysis detects vulnerabilities with the same or worse performance than the SAST tools.
- H_1: GPT-4-Advanced Data Analysis detects vulnerabilities with better performance than the SAST tools.

Design of the Experiment We have taken the code samples from GitHub or Snyk. We ran each code sample independently through the GPT-4-Advanced Data Analysis web and the SAST tool. It is unclear whether the API used in the beta version of the GPT Advanced Data Analysis(formerly code interpreter) is "8K context", as there is no API available for the Advanced Data Analysis and token/size pricing model at the time of writing this paper. However, there are indications [21] that the 8K context model is being used. Detailed records of their responses were maintained and are accessible [22]. For each tool's detection ability, outcomes were categorised binarily: 1 denoting correct detection and 0 indicating a miss. Given the comparative nature of the study, the Chi-Squared Test for Independence was chosen. This test facilitates the determination of significant differences between the detection capabilities of GPT-4 and the SAST tool. We made a 2x2 contingency table that was formulated as follows:

	GPT-4 Correct	GPT-4 Incorrect
SAST Correct	a	b
SAST Incorrect	c	d

Wherein:

- a: Represents vulnerabilities correctly identified by both tools.

– b: Represents vulnerabilities exclusively detected by the SAST tool.
– c: Represents vulnerabilities exclusively identified by GPT-4.
– d: Represents vulnerabilities that remained undetected by both entities.

Interpretation of Outcomes We used McNemar's test [23], and the p-value below 0.05 was established as the benchmark for statistical significance. If attained, it would signify a superior performance of GPT-4 over the SAST tool in the scope of vulnerability detection.

3.2 Security Vulnerabilities and Data

We explored the security scenarios that are listed in Table 1 and examined GPT4 and a SAST for each. Code snippets and answers of GPT and SAST are documented [22].

Table 1. Brief Descriptions of Various Attacks and sample code

ID	Coding security mistakes	sample	Description
1	Buffer overflow	[24]	Overwriting memory by overflowing a buffer.
2	SQL Injection	[25, 26]	injecting malicious SQL code into a query.
3	Cross-Site Scripting (XSS)	[27]	Injecting malicious scripts into web pages viewed by users.
4	Broken Access Control	[28]	Improperly enforcing what users can or cannot do.
5	Insecure deserialization	[29]	Exploiting unsafe data unmarshalling.
6	Log4J	[30]	Exploiting the Log4J Java logging library.
7	Unrestricted upload	[31]	Uploading malicious files without restrictions.
8	Improper input validation	[32]	Not verifying the user's input properly.
9	Memory Leak	[33]	Unintentional memory consumption leading to crashes.
10	Mass assignment	[34]	Overwriting object properties without restrictions.
11	Server-side request forgery	[35]	Making the server run unauthorized actions.
12	Insecure temporary files	[36]	Exploiting insecurely created temporary files.
13	Cleartext storage in a cookie	[37]	Storing sensitive data unencrypted in cookies.
14	XPath injection	[38]	Injecting malicious XPath queries.
15	Weak password recovery	[39]	Exploiting inadequate password recovery systems.
16	Logging vulnerabilities	[40]	Inadequately protecting or revealing logs.
17	Insecure Randomness	[41]	Using predictable random number generators.
18	NoSQL injection attack	[42]	Injecting malicious code into NoSQL queries.
19	Code injection	[43]	Injecting malicious code into an application.
20	No rate limiting	[44]	Overloading systems by not capping request rates.
21	Vulnerable components	[45]	Using outdated or flawed software components.
22	Insecure design	[46]	Designing systems without security in mind.
23	Insecure hash	[47]	Using weak hashing methods.
24	ReDoS	[48]	Exploiting regex to cause denial-of-service.
25	XML external entity injection	[49]	Attacking parsers with external XML entities.
26	Cross site request forgery	[50]	Making users unknowingly submit a malicious request.
27	DOM XSS	[51]	Injecting malicious scripts via the Document Object Model.
28	Open redirect	[52]	Redirecting users to malicious sites.
29	Directory traversal	[53]	Accessing files outside of the intended directory.
30	Prototype pollution	[54]	Altering prototype objects.
31	Container capabilities	[55]	Containers retaining unnecessary capabilities.
32	Container privileged mode	[56]	Running containers with full system privileges.

4 Results

The comparison results between the online SAST tool and GPT-4 for detecting security vulnerabilities are presented in Table 2. Vulnerabilities ranged from common issues like Buffer Overflow and SQL Injection to more specific ones like Prototype Pollution. GPT-4 consistently detected most vulnerabilities correctly, as indicated by a "1" under the "GPT-C" column. In contrast, tools like sonarcloud.io and clouddefense.ai had varied results, with some vulnerabilities detected correctly and others not.

To enhance the generalizability of our method to encompass a broader range of SAST tools, we introduced new columns named SAST-Correct and SAST-Incorrect into Table 3. We constructed the contingency matrix using these columns and GPT-Correct and GPT-Incorrect.

$$SAST_{\text{correct}} = \bigcup_{SAST} \text{Result}_{SAST}^{\text{correct}} \; ; SAST_{\text{incorrect}} = 1 - SAST_{\text{correct}}$$

Utilising McNemar's test [23], a comparative evaluation of vulnerability detection performance between GPT-4 and SAST tools was conducted. The test yielded a Chi-square value of 20.046 with an associated p-value of 0.000007562 using the in the appendix.

Given this result and adopting a significance level of 0.05%, we can reject the null hypothesis, and our experiment supports the alternative hypothesis:

- H_0: GPT-4-Advanced Data Analysis has the same or worse performance than SAST tools.
- H_1: GPT-4-Advanced Data Analysis has better performance than SAST tools.

5 Discussion and future research

GPT-4 has shown a promising ability to detect vulnerabilities that traditional SAST tools might miss. This revelation is significant for several reasons:

- Evolution of Detection Tools: As software development processes evolve, so too must the tools that ensure their security. The capabilities of GPT-4 in our experiment suggest that language models can serve as powerful supplements, if not alternatives, to traditional SAST tools.
- Cost Implications: Traditional SAST tools, especially proprietary ones, can be expensive. If language models can provide comparable or even superior performance, organisations might be able to reduce costs associated with security testing.
- Time Efficiency: The rapid analysis capabilities of models like GPT-4 could reduce the time taken for security assessments, especially in continuous integration/continuous deployment (CI/CD) environments.

However, while GPT-4's performance is commendable, it is essential to approach these findings with caution. Language models, no matter how advanced, are not infallible. They operate based on patterns in the data they have been trained on. If a novel vulnerability emerges after their training cut-off, they might not recognise it. Integration of language models into existing software development lifecycles requires careful consideration, especially concerning reliability, false positives/negatives, and the model's interpretability.

Looking ahead, there are multiple avenues for expanding upon this research:

Table 2. Comparison results for various security vulnerabilities. C: Correct detection; I: Incorrect detection. Tools compared include GPT-4 Advanced Data Analysis (GPT), sonarcloud.io (SQ), and clouddefense.ai (CDA). Vulnerabilities are referenced by their Common Weakness Enumeration (CWE) ID, available at https://cwe.mitre.org/.

ID	Security vulnerability inside a code snippet with its CWE	GPT-C	GPT-I	SQ-C	SQ-I	CDA-C	CDA-I	CWE
1	Buffer overflow [57]	1	0	1	0	1	0	121
2	SQL Injection [58]	1	0	1	0	1	0	564
3	Cross-Site Scripting (XSS) [59]:	1	0	0	1	0	1	79
4	Brocken Access Control [60]	1	0	0	1	0	1	284
5	Insecure deserialization [61]	1	0	0	1	0	1	502
6	log 4J [61]	1	0	0	1	0	1	502
7	Unrestricted upload of dangerous files [62]	1	0	0	1	0	1	434
8	Improper input validation [63]	1	0	0	1	0	1	20
9	Memory Leak [64]	1	0	0	1	1	0	401
10	Mass assignment with secret leak [65]	1	0	0	1	0	1	915
11	Server-side request forgery [66]	1	0	1	0	1	0	918
12	Insecure temporary file [67]	1	0	0	1	0	1	377
13	Plaintext storage of sensitive information in cookies [68]	1	0	0	1	0	1	315
14	XPath injection [69]	1	0	0	1	0	1	643
15	Weak password recovery [70]	1	0	0	1	0	1	640
16	Logging vulnerabilities [71]	1	0	0	1	0	1	532
17	Insecure Randomness [72]	1	0	0	1	0	1	330
18	NoSQL injection attack [73]	1	0	0	1	0	1	89
19	Code injection [74]	1	0	1	0	1	0	94
20	No rate limiting [75]	1	0	0	1	0	1	770
21	Vulnerable and outdated components [76]	0	1	0	1	0	1	1352
22	Insecure design [77]	0	1	0	1	0	1	657
23	Insecure hash [78]	1	0	0	1	0	1	328
24	ReDoS [79]	1	0	0	1	0	1	185
25	XML external entity injection [80]	1	0	1	0	1	0	611
26	Cross-site request forgery [81]	1	0	0	1	0	1	352
27	DOM XSS [82]	1	0	1	0	1	0	80
28	Open redirect [83]	1	0	1	0	1	0	601
29	Directory traversal [84]	1	0	0	1	1	0	23
30	Prototype pollution [85]	1	0	0	1	0	1	1321
31	Container does not drop default capabilities [86]	1	0	0	1	0	1	250
32	Container is running in privileged mode [86]	1	0	0	1	0	1	250

	GPT-4 Correct	GPT-4 Incorrect
SAST Correct	11	0
SAST Incorrect	22	2

Table 3. Comparison of SAST and GPT-4 Detection Abilities

1. Broader SAST Tool Comparison: While our study focused on two specific SAST tools, future research could incorporate a broader range of tools to provide a more comprehensive comparison.
2. Usability in Real-world Application: It would be beneficial to test GPT-4's detection capabilities in real-world scenarios, such as live software development environments, to assess its practical applicability and compare its output with expert opinion.
3. Integration with Development Environments: Research could explore how GPT-4 or similar models can seamlessly and securely integrate into popular development environments and platforms.
4. Security-trained LLMs: While GPT-4 is a generalised model, there might be benefits in training custom language models specifically focused on security vulnerability detection.
5. LLMs-trained using Fault-Tolerant Quantum Computers(FTQC):
6. Jens Eisert et al. [87] provided a resource estimation for large machine learning models trained over Fault-Tolerant Quantum Computers (FTQC), focusing on significant computational expenses, power, and time consumption challenges. They demonstrated that FTQCs could offer efficient resolutions for generic (stochastic) gradient descent algorithms, scaling as $O(T^2 \times \text{polylog}(n))$, where n is the size of the models and T is the number of iterations in training. The effectiveness depended on the models being sufficiently dissipative and sparse with minimal learning rates. The authors also explored the practical application, benchmarking models ranging from seven million to 103 million parameters, and found potential for quantum enhancement in sparse training after model pruning. This paper opens a new avenue for researching the resources and impact of training security-focused LLMs using FTQCs.

Additionally, we must recognise that these advancements have both benefits and risks. While these models can help defence, attackers could also use them to find new vulnerabilities, introducing an asymmetry in Offensive Cyber Operations (OCO) that necessitates vigilant monitoring and research, which we introduce in the next section.

5.1 Security concerns of LLMs

CISA has emphasised that AI should adhere to the principle of "Secure by Design" [88], suggesting a comprehensive threat model tailored for domain-specific LLMs, such as GPTs specialised in vulnerability scanning. BSI has outlined several threats pertinent to AI security [9]. Furthermore, it is crucial to recognise that many MLOps solutions rely on open-source frameworks. This fact introduces heightened security vulnerabilities, especially concerning supply chain attacks on open-source resources and undetectable hidden backdoors [89]. Shafi Goldwasser et al. shared an AI-era wisdom like *Reflections on Trusting Trust* [90], which showcases undetectable backdoors in AI.

We outline the various attacks linked to LLMs; however, creating a comprehensive threat map for LLM-enhanced SAST is an important area for future research:

1. **Poisoning Attacks:** Attackers introduce malicious data into the training set to compromise the model's performance [91].
2. **Backdoor Attacks:** Attackers embed a hidden behaviour within a model, triggered by specific inputs during deployment [92].
3. **Supply chain attack:** Malicious activities aimed at tampering with the AI software supply chain to compromise the model or system [93].
4. **Endpoint /API security breach:** Exploiting vulnerabilities in the AI system's access points or interfaces to gain unauthorised access or leak information [94].
5. **Model Stealing Attacks:** For organisations that invested significant resources in developing a commercial or mission-critical AI model, model stealing is a threat [95].
6. **Membership Inference Attacks:** In membership inference attacks, the attacker tries to determine whether a data sample was part of a model's training data [96].
7. **Attribute Inference Attacks:** In attribute inference attacks, the attacker seeks to breach the confidentiality of the model's training data by determining the value of a sensitive attribute associated with a specific individual or identity in the training data [97].
8. **Model Inversion Attacks:** Model inversion attacks aim to recover features that characterise classes from the training data [98].
9. **Denial of Service:** Attackers overload or manipulate the AI system, rendering it non-operational or degrading its performance [9].
10. **Prompt injection:** Manipulating the input prompts to mislead or control the output of models like GPT-4 [99].
11. **Jailbreaks:** Bypassing restrictions or controls put on language models to access broader or hidden functionalities [100].
12. **Privacy breach:** Exploiting the model to reveal sensitive or private information it might have been exposed to during training [101].
13. **GAN-based Attack:** Using Generative Adversarial Networks to confuse or mislead the target AI model into making incorrect predictions or classifications [9].

In this paper, we do not elaborate on the attack surface of LLMs. Instead, our focus is to underscore the significance of adhering to principles such as security by design/default and privacy by design/default. We advocate for integrating **MLSecOps** and emphasise the application of defence-in-depth strategies, notably the Zero Trust Architecture. It is also imperative to consider specific requirements like the Software Bill of Materials (SBOM) and other best practices when leveraging LLM-enhanced vulnerability scanning. These considerations are not just recommended; they are indispensable. Also, GPT4 and code-LLMs may generate insecure codes that warn about over-reliance on LLMs [102].

As revealed in the Vulkan files [103], reconnaissance systems are an undeniable component of cyber warfare. Therefore, it's imperative to enhance the resilience and robustness of LLM-enhanced SASTs through Federated Learning (FL-LLM), as these systems will become targets if they are not already. However, introducing FL-LLMs might also present new security challenges [104]. While using FL-LLM for training on a European scale/transatlantic scale might be feasible, training using a *reliable dataset* is crucial as the dataset's quality will directly reflect the final performance of the model. Model hyperparameter tuning, retraining, and pruning will require substantial resources. Therefore, developing a high-quality European or even transatlantic dataset is unavoidable. Without such datasets, LLMs risk becoming costly failures due to the resource-intensive nature of training, model serving, and inference, leading to the potential for undertrained or poisoned models in cyber defence [105, 106]. Present datasets [107] often lack comprehensive coverage of all known CWEs, proper labelling, and multi-language data.

6 Conclusions

This study emphasises the superior capabilities of GPT-4 (Advanced Data Analysis beta) in identifying software vulnerabilities compared to traditional SAST tools. However, integrating Language Models (LLMs) like GPT-4 into vulnerability scanning requires a comprehensive understanding of associated security concerns and a commitment to evolving security best practices from MLOps to MLSecOps. Important resources should be dedicated to acquiring high-quality datasets and ensuring resource-efficient model training, inference, and serving using a certified MLSecOps toolset. It's important to remember that there's no silver bullet in vulnerability scanning; the window of vulnerability will remain a challenge unless datasets, tooling, and skills are optimised to leverage vulnerability scanning LLMs effectively in DevSecOps. One recommendation is that European entities consider initiating challenges similar to AIxCC. This will facilitate the UK/EU to address security issues and harness the potential of next-generation vulnerability scanning LLMs before they emerge as an asymmetric capability in offensive cybersecurity operations.

Since both LLMs and SAST are evolving and progressing daily, our research subject is a moving target. An *LLM-enhanced SAST* or *vulnerability scanning LLMs* may soon become de facto, and their comparative studies become the next research subject [108]. Another limitation is that we did not cover all CWEs and all SASTs/LLMs.

Bibliography

[1] Imran Ghafoor, Imran Jattala, Shakeel Durrani, and Ch Muhammad Tahir. Analysis of openssl heartbleed vulnerability for embedded systems. In *17th IEEE International Multi Topic Conference 2014*, pages 314–319. IEEE, 2014.

[2] Ravi Sen. Challenges to cybersecurity: Current state of affairs. *Communications of the Association for Information Systems*, 43(1):2, 2018.

[3] Tushar Sharma, Maria Kechagia, Stefanos Georgiou, Rohit Tiwari, Indira Vats, Hadi Moazen, and Federica Sarro. A survey on machine learning techniques for source code analysis. *arXiv preprint arXiv:2110.09610*, 2021.

[4] OpenAI. Gpt-4 technical report, 2023. URL https://arxiv.org/abs/2303.08774.

[5] Håvard D Johansen and Robbert van Renesse. Firepatch: Secure and time-critical dissemination of software patches. *IFIP*, pages 373–384, 01 2007. https://doi.org/10.1007/978-0-387-72367-9_32. URL https://link.springer.com/chapter/10.1007/978-0-387-72367-9_32.

[6] Regulations on the management of network product security vulnerabilities, 2021. URL https://www.gov.cn/gongbao/content/2021/content_5641351.htm.

[7] Tianfu cup international cybersecurity contest, 2022. URL https://www.tianfucup.com/2022/en/.

[8] DARPA. Artificial intelligence cyber challenge (aixcc), 2023. URL https://www.dodsbirsttr.mil/topics-app/?baa=DOD_SBIR_2023_P1_C4.

[9] BSI. Ai security concerns in a nutshell, 03 2023. URL https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/KI/Practical_AI-Security_Guide_2023.pdf?__blob=publicationFile&v=5.

[10] BSI. Machine learning in the context of static application security testing - ml-sast, 02 2023. URL https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/ML-SAST/ML-SAST-Studie-final.pdf?__blob=publicationFile&v=5.

[11] Dominik Sobania, Carol Hanna, Martin Briesch, and Justyna Petke. An analysis of the automatic bug fixing performance of chatgpt, 01 2023. URL https://arxiv.org/pdf/2301.08653.pdf.

[12] Wei Ma, Shangqing Liu, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, and Yang Liu. The scope of chatgpt in software engineering: A thorough investigation, 05 2023. URL https://arxiv.org/pdf/2305.12138.pdf.

[13] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. The hitchhiker's guide to program analysis: A journey with large language models, 08 2023. URL https://arxiv.org/pdf/2308.00245.pdf.

[14] Norbert Tihanyi, Tamas Bisztray, Ridhi Jain, Mohamed Ferrag, Lucas Cordeiro, and Vasileios Mavroeidis. The formai dataset: Generative ai in software security through the lens of formal verification *, 07 2023. URL https://arxiv.org/pdf/2307.02192.pdf.

[15] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code, 2021.

[16] Anton Cheshkov, Pavel Zadorozhny, and Rodion Levichev. Technical report: Evaluation of chatgpt model for vulnerability detection, 04 2023. URL https://arxiv.org/pdf/2304.07232.pdf.

[17] Xin Liu, Yuan Tan, Zhenghang Xiao, Jianwei Zhuge, and Rui Zhou. Not the end of story: An evaluation of chatgpt-driven vulnerability description mappings, 2023. URL https://aclanthology.org/2023.findings-acl.229.pdf.
[18] Owasp top 10 for large language model applications — owasp foundation, 2023. URL https://owasp.org/www-project-top-10-for-large-language-model-applications/.
[19] Sonarcloud, 2023. URL https://www.sonarsource.com/products/sonarcloud/.
[20] Clouddefense.ai, 2023. URL https://www.clouddefense.ai/.
[21] nikola. Jailbreaking gpt-4's code interpreter, 07 2023. URL https://www.lesswrong.com/posts/KSroBnxCHodGmPPJ8/jailbreaking-gpt-4-s-code-interpreter.
[22] Madjid G. Tehrani, Eldar Sultanow, William J Buchanan, Houmani Mahkameh, and H. Djaha Fodja Christel. Source code, gpt results: Gptvssast. https://github.com/Sultanow/vulnerability-detector-llm, 2023.
[23] Quinn McNemar. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 12:153–157, 06 1947. https://doi.org/10.1007/bf02295996. URL https://link.springer.com/article/10.1007/BF02295996.
[24] Buffer overflow, 2023. URL https://github.com/pikulet/mem-attacks-example/blob/master/buffer_overflow/buffer_overflow.c.
[25] Sql injection attack, 04 2019. URL https://github.com/doublehops/sql-injection-attack-example.
[26] What is sql injection (sqli)?, 2021. URL https://learn.snyk.io/lesson/sql-injection/.
[27] What is cross-site scripting (xss)?, 2021. URL https://learn.snyk.io/lesson/xss/.
[28] Broken access control, 2022. URL https://learn.snyk.io/lesson/broken-access-control/.
[29] Insecure deserialization, 2022. URL https://learn.snyk.io/lesson/insecure-deserialization/.
[30] Log4shell, 12 2021. URL https://www.lunasec.io/docs/blog/log4j-zero-day/.
[31] What is unrestricted file upload?, 2023. URL https://learn.snyk.io/lesson/unrestricted-file-upload/.
[32] What is improper input validation?, 2023. URL https://learn.snyk.io/lesson/improper-input-validation/.
[33] What are memory leaks?, 2023. URL https://learn.snyk.io/lesson/memory-leaks/.
[34] What is mass assignment?, 2023. URL https://learn.snyk.io/lesson/mass-assignment/.
[35] What is ssrf?, 2023. URL https://learn.snyk.io/lesson/ssrf-server-side-request-forgery/.
[36] What is an insecure temporary file?, 2023. URL https://learn.snyk.io/lesson/insecure-temporary-file/.
[37] The dangers of storing cleartext sensitive information in a cookie?, 2023. URL https://learn.snyk.io/lesson/cleartext-sensitive-information-in-cookie/.
[38] What is an xpath injection?, 2023. URL https://learn.snyk.io/lesson/xpath-injection/.
[39] What is weak password recovery?, 2023. URL https://learn.snyk.io/lesson/weak-password-recovery/.

[40] What are logging vulnerabilities?, 2023. URL https://learn.snyk.io/lesson/logging-vulnerabilities/.
[41] What is insecure randomness?, 2023. URL https://learn.snyk.io/lesson/insecure-randomness/.
[42] Nosql injection attack, 2022. URL https://learn.snyk.io/lesson/nosql-injection-attack/.
[43] What is code injection?, 2022. URL https://learn.snyk.io/lesson/malicious-code-injection/.
[44] No rate limiting, 2022. URL https://learn.snyk.io/lesson/no-rate-limiting/.
[45] How to manage vulnerable and outdated components, 2022. URL https://learn.snyk.io/lesson/vulnerable-and-outdated-components/.
[46] Insecure design, 2022. URL https://learn.snyk.io/lesson/insecure-design/.
[47] What is an insecure hash?, 2022. URL https://learn.snyk.io/lesson/insecure-hash/.
[48] Redos, 2022. URL https://learn.snyk.io/lesson/redos/.
[49] Xxe attack, 2022. URL https://learn.snyk.io/lesson/xxe/.
[50] Csrf attack, 2022. URL https://learn.snyk.io/lesson/csrf-attack/.
[51] Dom based xss, 2022. URL https://learn.snyk.io/lesson/dom-based-xss/.
[52] Open redirect vulnerability, 2022. URL https://learn.snyk.io/lesson/open-redirect/.
[53] What is directory traversal?, 2021. URL https://learn.snyk.io/lesson/directory-traversal/.
[54] What is prototype pollution?, 2021. URL https://learn.snyk.io/lesson/prototype-pollution/.
[55] Container does not drop all default capabilities, 2021. URL https://learn.snyk.io/lesson/container-does-not-drop-all-default-capabilities/.
[56] Container runs in privileged mode, 2021. URL https://learn.snyk.io/lesson/container-runs-in-privileged-mode/.
[57] MITRE Corporation. Cwe-121: Stack-based buffer overflow, 2023. URL https://cwe.mitre.org/data/definitions/121.html. Accessed: 21-08-2023.
[58] MITRE Corporation. Cwe-564: Sql injection, 2023. URL https://cwe.mitre.org/data/definitions/564.html. Accessed: 21-08-2023.
[59] MITRE Corporation. Cwe-79: Improper neutralization of input during web page generation ('cross-site scripting'), 2023. URL https://cwe.mitre.org/data/definitions/79.html. Accessed: 21-08-2023.
[60] MITRE Corporation. Cwe-284: Improper access control, 2023. URL https://cwe.mitre.org/data/definitions/284.html. Accessed: 21-08-2023.
[61] MITRE Corporation. Cwe-502: Deserialization of untrusted data, 2023. URL https://cwe.mitre.org/data/definitions/502.html. Accessed: 21-08-2023.
[62] MITRE Corporation. Cwe-434: Unrestricted upload of file with dangerous type, 2023. URL https://cwe.mitre.org/data/definitions/434.html. Accessed: 21-08-2023.
[63] MITRE Corporation. Cwe-20: Improper input validation, 2023. URL https://cwe.mitre.org/data/definitions/20.html. Accessed: 21-08-2023.
[64] MITRE Corporation. Cwe-401: Missing release of memory after effective lifetime, 2023. URL https://cwe.mitre.org/data/definitions/401.html. Accessed: 21-08-2023.
[65] MITRE Corporation. Cwe-915: Improperly controlled modification of dynamically-determined object attributes, 2023. URL https://cwe.mitre.org/data/definitions/915.html. Accessed: 21-08-2023.

[66] MITRE Corporation. Cwe-918: Server-side request forgery (ssrf), 2023. URL https://cwe.mitre.org/data/definitions/918.html. Accessed: 21-08-2023.

[67] MITRE Corporation. Cwe-377: Insecure temporary file, 2023. URL https://cwe.mitre.org/data/definitions/377.html. Accessed: 21-08-2023.

[68] MITRE Corporation. Cwe-315: Cleartext storage of sensitive information in a cookie, 2023. URL https://cwe.mitre.org/data/definitions/315.html. Accessed: 21-08-2023.

[69] MITRE Corporation. Cwe-643: Improper neutralization of data within xpath expressions ('xpath injection'), 2023. URL https://cwe.mitre.org/data/definitions/643.html. Accessed: 21-08-2023.

[70] MITRE Corporation. Cwe-640: Weak password recovery mechanism for forgotten password, 2023. URL https://cwe.mitre.org/data/definitions/640.html. Accessed: 21-08-2023.

[71] MITRE Corporation. Cwe-532: Insertion of sensitive information into log file, 2023. URL https://cwe.mitre.org/data/definitions/532.html. Accessed: 21-08-2023.

[72] MITRE Corporation. Cwe-330: Use of insufficiently random values, 2023. URL https://cwe.mitre.org/data/definitions/330.html. Accessed: 21-08-2023.

[73] MITRE Corporation. Cwe-89: Improper neutralization of special elements used in an sql command ('sql injection'), 2023. URL https://cwe.mitre.org/data/definitions/89.html. Accessed: 21-08-2023.

[74] MITRE Corporation. Cwe-94: Improper control of generation of code ('code injection'), 2023. URL https://cwe.mitre.org/data/definitions/94.html. Accessed: 21-08-2023.

[75] MITRE Corporation. Cwe-770: Allocation of resources without limits or throttling, 2023. URL https://cwe.mitre.org/data/definitions/770.html. Accessed: 21-08-2023.

[76] MITRE Corporation. Cwe-1352: Owasp top ten 2021 category a06:2021 - vulnerable and outdated components, 2023. URL https://cwe.mitre.org/data/definitions/1352.html. Accessed: 21-08-2023.

[77] MITRE Corporation. Cwe-657: Violation of secure design principles, 2023. URL https://cwe.mitre.org/data/definitions/657.html. Accessed: 21-08-2023.

[78] MITRE Corporation. Cwe-328: Use of weak hash, 2023. URL https://cwe.mitre.org/data/definitions/328.html. Accessed: 21-08-2023.

[79] MITRE Corporation. Cwe-185: Incorrect regular expression, 2023. URL https://cwe.mitre.org/data/definitions/185.html. Accessed: 21-08-2023.

[80] Cwe-611: Improper restriction of xml external entity reference, 2023. URL https://cwe.mitre.org/data/definitions/611.html. Accessed: 21-08-2023.

[81] Cwe-352: Cross-site request forgery (csrf), 2023. URL https://cwe.mitre.org/data/definitions/352.html. Accessed: 21-08-2023.

[82] Cwe-80: Improper neutralization of script-related html tags in a web page (basic xss), 2023. URL https://cwe.mitre.org/data/definitions/80.html. Accessed: 21-08-2023.

[83] MITRE Corporation. Cwe-601: Url redirection to untrusted site ('open redirect'), 2023. URL https://cwe.mitre.org/data/definitions/601.html. Accessed: 21-08-2023.

[84] Cwe-23: Relative path traversal, 2023. URL https://cwe.mitre.org/data/definitions/23.html. Accessed: 21-08-2023.

[85] Cwe-1321: Improperly controlled modification of object prototype attributes ('prototype pollution'), 2023. URL https://cwe.mitre.org/data/definitions/1321.html. Accessed: 21-08-2023.

[86] Cwe-250: Execution with unnecessary privileges, 2023. URL https://cwe.mitre.org/data/definitions/250.html. Accessed: 21-08-2023.

[87] Jens Eisert, Junyu Liu, Minzhao Liu, Jin-Peng Liu, Ziyu Ye, Yuri Alexeev, and Liang Jiang. Towards provably efficient quantum algorithms for large-scale machine learning models. *Research Square (Research Square)*, 2023. https://doi.org/10.21203/rs.3.rs-2860733/v1. URL https://www.researchsquare.com/article/rs-2860733/v1.

[88] CISA. Software must be secure by design, and artificial intelligence is no exception — cisa, 08 2023. URL https://www.cisa.gov/news-events/news/software-must-be-secure-design-and-artificial-intelligence-no-exception.

[89] Shafi Goldwasser, Michael P Kim, Vinod Vaikuntanathan, and Or Zamir. Planting undetectable backdoors in machine learning models. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 931–942. IEEE, 2022.

[90] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8): 761–763, 1984.

[91] Loc Truong, Chace Jones, Brian Hutchinson, Andrew August, Brenda Praggastis, Robert Jasper, Nicole Nichols, and Aaron Tuor. Systematic evaluation of backdoor data poisoning attacks on image classifiers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2020.

[92] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning, 2017. URL https://arxiv.org/abs/1712.05526.

[93] Daniel Williams, Chelece Clark, Rachel McGahan, Bradley Potteiger, Daniel Cohen, and Patrick Musau. Discovery of ai/ml supply chain vulnerabilities within automotive cyber-physical systems. In *2022 IEEE International Conference on Assured Autonomy (ICAA)*, pages 93–96. IEEE, 2022.

[94] Maria Korolov. Why api attacks are increasing and how to avoid them, 07 2023. URL https://www.csoonline.com/article/646557/why-api-attacks-are-increasing-and-how-to-avoid-them.html.

[95] I know what you trained last summer: A survey on stealing machine learning models and defences — acm computing surveys, 2023. URL https://dl.acm.org/doi/10.1145/3595292.

[96] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. *arXiv (Cornell University)*, 05 2017. https://doi.org/10.1109/sp.2017.41. URL https://ieeexplore.ieee.org/document/7958568.

[97] Yuheng Zhang, Ruoxi Jia, Hengzhi Pei, Wenxiao Wang, Bo Li, and Dawn Song. The secret revealer: Generative model-inversion attacks against deep neural networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 253–261, 2020.

[98] Akshay Chawla, Hongxu Yin, Pavlo Molchanov, and Jose Alvarez. Data-free knowledge distillation for object detection. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 3289–3298, 2021.

[99] Fábio Perez and Ian Ribeiro. Ignore previous prompt: Attack techniques for language models. *arXiv preprint arXiv:2211.09527*, 2022.

[100] Xinyue Shen, Zeyuan Chen, Michael Backes, Yun Shen, and Yang Zhang. "do anything now": Characterizing and evaluating in-the-wild jailbreak prompts on large language models, 2023. URL https://arxiv.org/abs/2308.03825.

[101] Xudong Pan, Mi Zhang, Shouling Ji, and Min Yang. Privacy risks of general-purpose language models. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1314–1331. IEEE, 2020.

[102] Owura Asare, N Asokan, and Meiyappan Nagappan. Copilot security: A user study, 08 2023. URL https://arxiv.org/pdf/2308.06587.pdf.

[103] Craig Timberg, Ellen Nakashima, Hannes Munzinger, and Hakan Tanriverdi. Secret trove offers rare look into russian cyberwar ambitions, 03 2023. URL https://www.washingtonpost.com/national-security/2023/03/30/russian-cyberwarfare-documents-vulkan-files/.

[104] Chaochao Chen, Xiaohua Feng, Jun Zhou, Jianwei Yin, and Xiaolin Zheng. Federated large language model : A position paper, 2023. URL https://arxiv.org/pdf/2307.08925.pdf.

[105] Fuzhao Xue, Yao Fu, Wangchushu Zhou, Zangwei Zheng, and Yang You. To repeat or not to repeat: Insights from scaling llm under token-crisis, 05 2023. URL https://arxiv.org/pdf/2305.13230.pdf.

[106] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models, 2022.

[107] Yizheng Chen, Zhoujie Ding, Xinyun Chen, and David Wagner. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. *arXiv preprint arXiv:2304.00409*, 2023.

[108] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 06 2022. https://doi.org/10.1145/3520312.3534862.

7 Appendix

```
import pandas as pd
from statsmodels.stats.contingency_tables import McNEmar

# 1. Read the data from the Excel file
df = pd.read_excel('securitybugsresults_hypothesis-testing.xlsx')

# 2. Prepare the contingency matrix for McNemar's test
b = df[(df["GPT–Correct"] == 1) & (df["SAST–Incorrect"] == 1)].
    shape[0]
c = df[(df["GPT-Incorrect"] == 1) & (df["SAST–Correct"] == 1)].
    shape[0]
table = [[0, b], [c, 0]]

# 3. Perform McNemar's test
result = McNamara(table, exact=False, correction=True)
```

```
14
15 # 4. Provide the result of the hypothesis testing
16 if result.pvalue < 0.05:
17     hypothesis_result = "Reject the null hypothesis (H0). GPT-4
           has better performance than the SAST tools."
18 else:
19     hypothesis_result = "Fail to reject the null hypothesis (H0).
           GPT-4 does not perform statistically significantly better
           than the SAST tools."
20
21 print(f"Chi-Squared: {result.statistic}")
22 print(f"p-value: {result.pvalue}")
23 print(hypothesis_result)
```