2023 IEEE 9th International Women in Engineering (WIE) Conference on Electrical and Computer Engineering (WIECON-ECE) | 979-8-3503-1965-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/WIECON-ECE60392.2023.10456393

Vulnerability Detection and Monitoring Using LLM

Vishwanath Akuthota[1], Raghunandan Kasula[2], Sabiha Tasnim Sumona[3], Masud Mohiuddin[4],
Md Tanzim Reza[5], Md Mizanur Rahman[6]
[1,2] Drpinnacle, Hyderabad, India
[3,6]Department of Computer Science and Engineering, Purbachal American City, Kanchon 1460,
Green University of Bangladesh
[4,5]Department of Computer Science and Engineering, BRAC University,66 Mohakhali, Dhaka 1212, Bangladesh
Email: [1]vishwanath.akuthota@gmail.com, [2]nandanr094@gmail.com, [3]sts.sabiha@gmail.com,
[4]er.masud.mohiuddin@gmail.com, [5]tanzim.reza@bracu.ac.bd, [6]rafi79466gmail.com

***Abstract*—Large Language Models (LLMs) have evolved as a cornerstone for intricate code evaluations in the modern realm of artificial intelligence and machine learning. The prioritizing of rigorous security requirements is a crucial requirement for the business in the dynamic and ever-changing world of software development. The current study has used the capabilities of the GPT-3.5-Turbo model to conduct a detailed assessment of various code snippets to find any vulnerabilities. The main objective of the experiment was to introduce continuous monitoring technologies to enhance software security and release control. To obtain reliable results, we used a classification report and a confusion matrix. Out of these validation methods we choose accuracy as an important metric for this validation because in this experiment we need our model to predict the vulnerabilities that are present in the 2740 test cases and we would need our model to focus more on true positives(TP). The ideal goal of this experiment was to predict any kind of vulnerability from the real-world data. Out of all test cases, we were able to have an accuracy of 0.77. This demonstrates the approach's potential efficacy in discovering vulnerabilities. Nonetheless, the study found certain parts that require improvement, emphasizing the importance of continual refinement in the model's methodology to ensure more thorough security assessments. This study lays the groundwork for future research into the use of powerful machine learning models in the assessment of software vulnerabilities. The findings not only highlight the effectiveness of the existing approach but also offer light on prospective future research directions, paving the way for the next generation of models and evaluation techniques.**

***Index Terms*—Language Model Models (LLMs), Vulnerability, ChatGPT, GPT-3.5-Turbo model, OpenAI.**

I. Introduction

The need for reliable and secure software applications is greater than ever in today's digital environment, which is characterized by constant change due to digitalization. One of the most important tasks that arise with the expansion of the digital sphere is the need to strengthen software programs against potential breaches and cyber-attacks. A software vulnerability is akin to an unsecured door that allows unauthorized parties to obtain, misuse, or steal confidential data. To protect against these impending risks, the software community is currently investigating several approaches to find and fix vulnerabilities in software code bases.

Technological advancements in recent times, especially the introduction of sophisticated machine learning models like the Generative Pre-trained Transformers (GPT) series, have brought an intriguing new dimension to this project. Notably, the GPT-3.5-Turbo model has demonstrated a very high level of proficiency in understanding, producing, and evaluating text, making it potentially a very useful tool for software code evaluation.

However, in the pursuit of this cutting-edge technological solution, it is imperative to conduct a meticulous evaluation of its utility, strengths, and areas for potential enhancement as it emerges into the software security stage. The principal objective of this study is to meticulously scrutinize the capabilities of the GPT-3.5-Turbo model, with a particular focus on its proficiency in discerning vulnerabilities within diverse code samples.

Furthermore, the ramifications of this study transcend the realm of vulnerability assessment for the real world. The study project will yield insights that will not only help restructure software release management procedures but also set new software security benchmarks and support the deployment of continuous monitoring systems. This extensive study tries to outline a distinct goal and course for upcoming developments and improvements in the area of software security.

II. Related Work

This technical study compares ChatGPT and GPT-3 models for code vulnerability identification. A real-world dataset was used for CWE vulnerability for binary and multi-label classification tasks. Since the model performed well on other code-based tasks including programming difficulties and high-level code knowledge, the authors chose to assess it. The ChatGPT model could not outperform a dummy classifier for binary and multi-label code vulnerability detection classification tasks [1]. In assessing the potential of ChatGPT for Vulnerability Description Mapping (VDM), the research highlights its proficiency in certain tasks, approaching human expert levels, yet underscores due to its inability to fully supplant professional security engineers in comprehensive vulnerability analysis [2]. Meanwhile, another study introduces BadGPT, the first backdoor attack on language model reinforcement learning (RL) driven fine-tuning. During fine-tuning, a reward model backdoor can jeopardize the language model. The authors showed that BadGPT allowed an attacker to modify generated text for harmful purposes [3]. The authors of the paper [4]

979-8-3503-1965-1/23/$31.00 ©2023 IEEE

argued that UTAAL is a serious threat to the security of LLMs. They suggest that LLM developers need to take steps to mitigate the risk of UTAAL attacks, such as filtering out attack suffixes from queries. The study in [5] investigates the security ramifications associated with the utilization of large language models (LLMs) to aid programming activities. [6].

The capabilities of modern LLMs can be dynamically adjusted with the use of natural language prompts. The authors in the study [7] present a comprehensive methodology for developing natural language instructions that can effectively elicit the intended programmatic responses from Language Model-based Systems (LLMs). These experiments encompassed four distinct attack scenarios, namely phishing email production, celebrity tweet imitation, SQL injection, and cross-site scripting. The study in [8] presents a novel privacy-aware data augmentation technique for LLM-based patient-trial matching (LLM-PTM) that maximizes LLM advantages while protecting sensitive patient data. The article [9] Private Meta-Learning (DP-Meta) trains accurate, privacy-preserving LLMs using differential privacy and meta-learning. [10]

III. METHODOLOGY

A. Tool Configuration and Setup

The research applied the OpenAI interface, a specialized framework designed for interacting with OpenAI models. A reduction in the value of the parameter yields a heightened amount of determinism in the output of the model, whilst an augmentation in the value of the parameter gives rise to an increased degree of unpredictability.

B. Vulnerability Detection Function

A primary function named 'find security issues and generate fix', was designed to be the centerpiece of this analysis. This function was tasked with:

- Scanning the provided code.
- Identifying any security vulnerabilities.
- Proposing potential fixes for detected vulnerabilities.

The function parameters were designed to capture critical information, including:

- Whether a vulnerability was detected.
- The type of identified vulnerability.
- The segment of code that is deemed vulnerable.

Additional comments or descriptions about the detected issue and the proposed remedy.

C. Vulnerability Detection Process

Our research employed a multifaceted approach, harnessing manual techniques, automated tools, and predictive analytics to pinpoint and scrutinize vulnerabilities in web applications. The flowchart in figure 1 shows the steps involved in a vulnerability detection process.

Reading data UHL: The process was initiated by extracting data from the Universal Headers List, a repository of documented vulnerabilities.

Fetching the data from the URL: The URL is the address of the website or web application that is being scanned for vulnerabilities.

Code result: This step involves looking for any code that may be vulnerable to attack.

Mata data result: This step involves looking for any metadata that may be vulnerable to attack.

Predict Vulnerables: This step involves using machine learning or other predictive analytics techniques to identify potential vulnerabilities.

Fuzz rubo: This step involves sending random input to the website or web application in an attempt to trigger a vulnerability.

Saving results on CSV: This file can be used to track the vulnerabilities that have been found and to prioritize remediation efforts.

Analyzing results: This step involves reviewing the results to identify any high-priority vulnerabilities that need to be remediated.

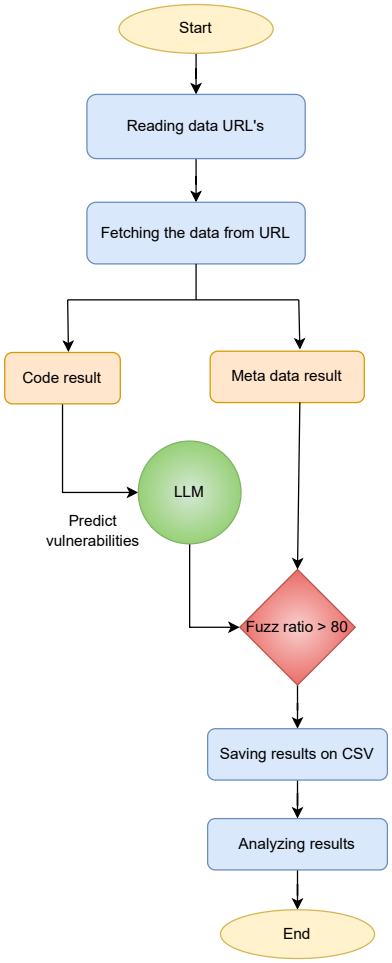

Fig. 1. vulnerability detection process

D. Fuzzy String Matching

The function mentioned above calculates the similarity ratio between two strings by determining the most similar sub-string

and represents this ratio as a numerical value ranging from 0 to 100. By evaluating whether this ratio exceeds 80, it is assuring that a significant level of resemblance exists between the two strings. An often employed heuristic is to set a threshold of 80, indicating that the two strings must have a minimum similarity of 80 Percent.

- We have two strings, str1 which is ”vulnerability” and str2 which is ”potential vulnerability in the system”.
- The goal is to identify the substring in str2 that has the highest similarity ratio with str1.
- In this case, the substring ”vulnerability” within str2 matches str1 perfectly, resulting in a similarity ratio of 100%.

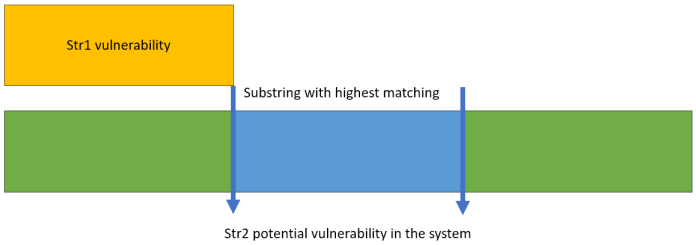

Fig. 2. Fuzzy Compares Two Strings

E. Automated Testing with the OWASP Benchmark

In our efforts to improve software security, the utilization of automated testing tools has been instrumental. We developed a function to streamline the evaluation of software vulnerabilities using the OWASP Benchmark.

The primary objective of this function is to fetch a specified test case's Java code and its associated metadata from the OWASP Benchmark repository. Following this, the code undergoes a rigorous static analysis to detect potential vulnerabilities. The outcomes of this analysis are then juxtaposed with the provided metadata, offering a comprehensive insight into the accuracy and efficiency of the static analysis tool in use.

The process initiates with the determination of base URLs, which are then tailored to point toward specific Java and XML files using the test case number. Upon the successful retrieval of the Java code and metadata, the code is subjected to static analysis, the results of which are rendered in a JSON format. A subsequent comparison of the analysis outcome with the metadata elucidates the congruency between the expected vulnerabilities and those identified by the tool.

This automated testing mechanism fosters a data-driven approach, enabling developers to discern the efficacy of their static analysis tools. By leveraging the function, we anticipate a significant enhancement in the precision of vulnerability detection, fortifying the security of software applications.

Our script performs the following steps:

- It constructs the URLs for the Java file and the metadata XML file for each test case number.
- It fetches the Java code and the metadata from the web and parses the XML file.
- It runs the LLMs on the code and converts the result from JSON to a dictionary.
- It compares the LLMs' result with the metadata and returns the outcome.

IV. Result

A. Able to Predict The vulnerabilities

The experiment, utilizing the GPT-3.5 Turbo model, was capable of analyzing code snippets from the provided dataset and identifying potential vulnerabilities with an accuracy of 0.77. Not only did it detect the presence of vulnerabilities, but it also categorized them into specific types (e.g., SQL Injection, Cross-Site Scripting). This indicates that the model can understand and interpret code to a degree where it can predict security flaws, making it a valuable tool for preliminary code reviews.

B. Developers can use this experiment for production

As we have seen this experiment has given us an accuracy of 0.77 so any developer can deploy any program or application for operational use, it is imperative to ascertain that it is devoid of any security flaws. By incorporating this experiment into the software development pipeline, developers can receive prompt feedback regarding potential security vulnerabilities present in their code. The adoption of a proactive strategy can enhance the security of software by enabling the identification and rectification of vulnerabilities before the commencement of the testing process.

C. Release managers can benefit from this experiment's application

Release managers have the primary responsibility of supervising the process of software product releases, intending to ensure that they adhere to established quality and security criteria. By converting this experiment into a self-contained application or tool, release managers can possess a readily available utility to swiftly examine codebases for vulnerabilities before approving any release, with an accuracy of 0.77.

D. Production-level continuous monitoring can be implemented

Through the implementation of this experiment, businesses can construct a perpetual monitoring system that does periodic scans of the codebase to identify vulnerabilities with an accuracy of 0.77. This practice guarantees the ongoing security of the software, even after the production phase, by rapidly identifying and resolving any newly introduced or previously undetected vulnerabilities. The experiment conducted by the researcher showcases not only the capacity to detect weaknesses in code but also emphasizes the wider range of practical uses for such a system within the software development life-cycle. The potential implications for improving software security, optimizing the release process, and assuring ongoing monitoring are considerable.

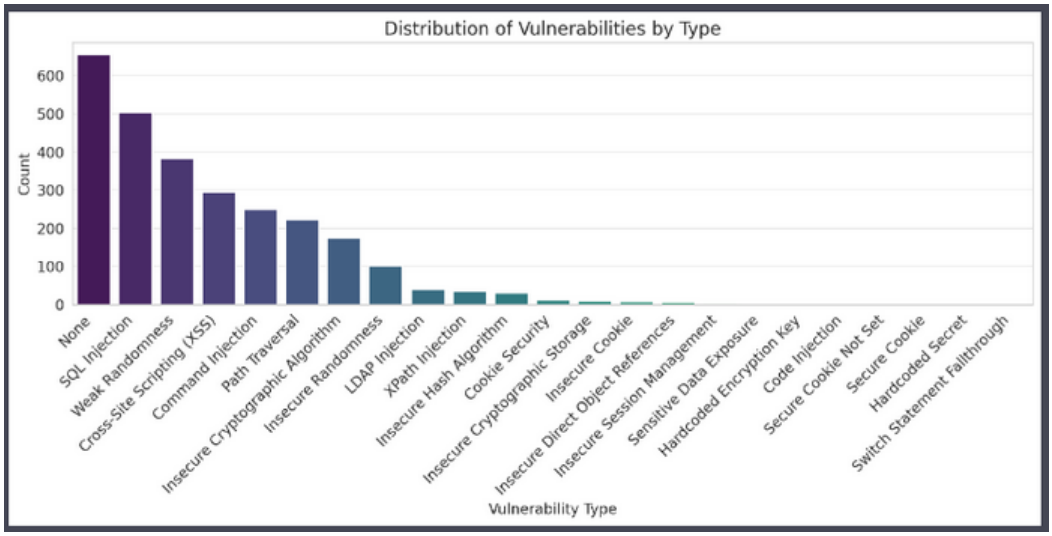

Fig. 3. Distribution of Vulnerabilities by Type

E. New Work Validity

Working with vulnerability detection and monitoring using LLM is a relatively new topic, and very little work has been done on this topic. As a result, we did not find any notable or comparable work to compare with our paper.

F. Correct vs Incorrect Vulnerability

The bar chart compares the correct and incorrect vulnerability type matches visually. There were 1,132 scenarios where the vulnerability type was matched appropriately (”Correct”). There were 1,608 scenarios where the vulnerability type was mismatched (”Incorrect”). This data shows the significance of fine-tuning the vulnerability identification process to improve accuracy.

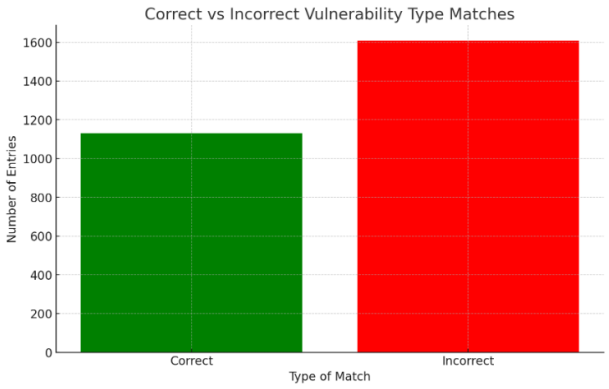

Fig. 4. Correct vs Incorrect Vulnerability

G. Pie Chart of Vulnerabilities Found

The presented pie chart illustrates the relative distribution of code snippets that exhibit vulnerabilities in comparison to those that do not possess any flaws. Based on the statistics presented in the graphic, it is apparent that a substantial proportion of the code snippets within the dataset exhibit vulnerabilities.

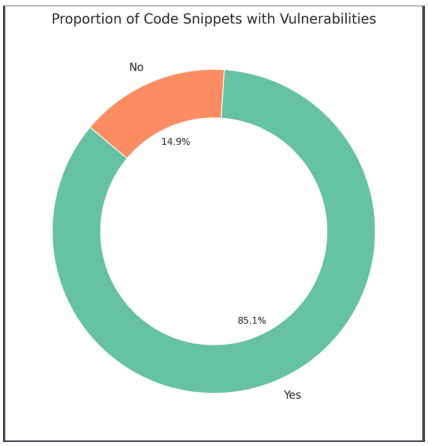

Fig. 5. Pie Chart of Vulnerabilities

H. Vulnerabilities Found

The main form of vulnerability encountered in computer systems is SQL injection, which is then followed by weak randomness, cross-site scripting (XSS), and command injection. Additional common vulnerabilities encompass path traversal, insecure cryptographic algorithms, LDAP injection, XPath injection, insecure hash algorithms, cookie security, insecure cryptographic storage practices, insecure session management, exposure of sensitive data, hardcoded encryption keys, code injection vulnerabilities, and hard-coded secret switch statements. Organizations may improve their risk reduction efforts and safeguard their systems against potential attacks by acquiring knowledge about prevalent vulnerabilities. The diagram proves to be helpful in identifying the prevalent vulnerabilities encountered by organizations. The provided information can

TABLE I
VULNERABILITY TYPES AND COUNTS

Vulnerability Type	Count	Description
None	655	No vulnerability found
SQL Injection	504	An injection attack that exploits vulnerabilities in SQL databases
Weak Randomness	383	The use of weak or predictable random numbers, which can be exploited by attackers
Cross-Site Scripting (XSS)	328	An attach to inject malicious code into a web page, which can then be executed by unsuspecting users
Command Injection	295	An injection attack that executes arbitrary commands on the server, often with administrator privileges
Path Traversal	250	An attack that allows an attacker to access files and directories that they should not be able to access
Insecure Cryptographic Algorithm	175	The use of insecure cryptographic algorithms, which can be easily broken by attackers
Insecure Randomness	102	The use of insecure or predictable random numbers, which can be exploited by attackers
LDAP Injection	40	An injection attack that exploits vulnerabilities in Lightweight Directory Access Protocol (LDAP) servers
XPath Injection	35	An attach to inject malicious XPath expressions into a web page to be executed by the browser
Insecure Hash Algorithm	31	The use of insecure hash algorithms, which can be easily broken by attackers
Cookie Security	12	Vulnerabilities in the way cookies are handled, such as improper encryption or lack of expiration dates
Insecure Cryptographic Storage	9	Insecured sensitive data storage, such as in plaintext or using weak encryption algorithms
Insecure Cookie	8	Vulnerabilities in the way cookies are used, such as improper validation or lack of SameSite flags
Insecure Direct Object References	7	Vulnerabilities in the way objects are referenced, such as by using unsanitized user input
Insecure Session Management	7	Session management Vulnerabilities, such as by using weak passwords or allowing session fixation
Sensitive Data Exposure	3	The exposure of sensitive data, such as credit card numbers or passwords
Hardcoded Encryption Key	2	The use of hardcoded encryption keys, which can be easily compromised by attackers
Code Injection	1	An attack that injects malicious code into a running program
Secure Cookie Not Set	1	The failure to set a secure cookie, which can make it easier for attackers to steal cookies
Secure Cookie	1	The use of a secure cookie, which helps to protect cookies from being stolen by attackers
Hardcoded Secret	1	The use of hardcoded secrets, which can be easily compromised by attackers
Switch Statement Fallthrough	1	A programming error that can allow attackers to execute unintended code

be utilized to allocate security resources effectively and concentrate on addressing the most critical vulnerabilities. These three were discovered the most: SQL Injection (504), Weak Randomness (383), and Cross-Site Scripting (XSS) (328). These three vulnerabilities are so frequent because they are so simple to attack. SQL Injection can be abused by inserting malicious code into a web form. Weak Randomness can be taken advantage of by guessing the value of a random number. Furthermore, XSS can be exploited simply by visiting a rogue website. (Table-1)

I. Confusion Matrix

According to the confusion matrix, the GPT-3.5-Turbo model accurately identified 600 instances as ”Negative” and 1195 instances as ”Positive”. Nevertheless, the model exhibited erroneous classification by labeling 189 data points as ”Positive” and 220 data points as ”Negative”.

The determination of the model’s overall accuracy entails the division of the count of accurately classified data points by the entire count of data points. In this instance, the model’s overall accuracy is 0.77, indicating a commendable level of accuracy. Nevertheless, the model exhibits a discernible inclination towards categorizing data points as ”Positive”.

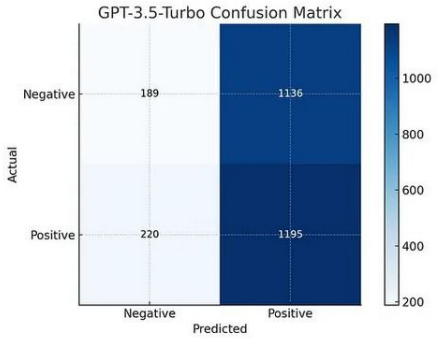

Fig. 6. confusion matrix for a GPT-3.5-Turbo model

CONCLUSION AND FUTURE WORK

Our research demonstrates the utility of adopting advanced models such as GPT-3.5-Turbo in the field of software security for the real world. While the model performed admirably in finding vulnerabilities, there is still a clear bias toward particular classes that require additional development. One of the limitations of this experiment is like LLMs have been majorly trained on plain text-based data, but if we have a future LLM which has been trained on the scoured code-based data from github or gitlab or any other means, that will solve the problem of continuous monitoring of the code. The findings underline the necessity of continual monitoring in software development, urging vigilance in discovering and correcting any vulnerabilities. As software landscapes become more complex, technologies that use cutting-edge models will be critical in guaranteeing comprehensive security. Future research should concentrate on improving the model’s precision, broadening its vulnerability-detecting repertoire, and developing pre-trained LLMs on source code.

We intend to further our study by converting the current codebase into an open-source application, improving accessibility, and encouraging community engagement. Optimizing prompt instructions will also be investigated to find a compromise between false positives and recall. We’re also interested in seeing how temperature changes affect the LLM’s

output quality in vulnerability identification. Finally, using multilingual datasets will allow us to comprehend the model's proficiency across various linguistic circumstances, allowing for a more thorough examination of its capabilities. These efforts attempt to improve and broaden the capabilities of LLMs in the detection of software vulnerabilities.

REFERENCES

[1] Cheshkov, A., Zadorozhny, P. and Levichev, R. (2023) "Evaluation of ChatGPT model for vulnerability detection," arXiv [cs.CR]. Available at: http://arxiv.org/abs/2304.07232 (Accessed: August 11, 2023).

[2] Liu, X. et al. (2023) "Not the end of story: An evaluation of ChatGPT-driven vulnerability description mappings," in Findings of the Association for Computational Linguistics: ACL 2023. Stroudsburg, PA, USA: Association for Computational Linguistics, pp. 3724–3731.

[3] Shi, J. et al. (2023) "BadGPT: Exploring security vulnerabilities of ChatGPT via backdoor attacks to InstructGPT," arXiv [cs.CR]. Available at: http://arxiv.org/abs/2304.12298 (Accessed: August 11, 2023).

[4] Zou, A. et al. (2023) "Universal and transferable adversarial attacks on aligned language models," arXiv [cs.CL]. Available at: http://arxiv.org/abs/2307.15043 (Accessed: August 11, 2023).

[5] Chen, H., Zhang, Y., He, X., Zhou, Y. (2023). Lost at C: A user study on the security implications of large language model code assistants. arXiv preprint arXiv:2208.09727.

[6] Greshake, K. et al. (2023) "Not what you've signed up for: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection," arXiv [cs.CR]. Available at: http://arxiv.org/abs/2302.12173.

[7] Ye, W. et al. (2023) "Assessing hidden risks of LLMs: An empirical study on robustness, consistency, and credibility," arXiv.org. doi: 10.48550/ARXIV.2305.10235.

[8] Bhatt, S., Shen, J., Han, J. and Chang, K.C.-C., 2022. Exploiting programmatic behavior of LLMs: dual-use through standard security attacks. arXiv preprint arXiv:2302.05733.

[9] Yuan, J. et al. (2023) "Large language models for healthcare data augmentation: An example on patient-trial matching," arXiv [cs.CL]. Available at: http://arxiv.org/abs/2303.16756 (Accessed: August 11, 2023).

[10] Tang, R., Chuang, Y.-N. and Hu, X. (2023) "The science of detecting LLM-generated texts," arXiv [cs.CL]. Available at: http://arxiv.org/abs/2303.07205 (Accessed: August 11, 2023).