Navigating (in)security of AI-generated code

Sri Haritha Ambati
University of Saskatchewan
Saskatoon, Canada
zus978@mail.usask.ca

Norah Ridley
University of Saskatchewan
Saskatoon, Canada
norah.ridley@usask.ca

Enrico Branca
University of Saskatchewan
Saskatoon, Canada
enb733@usask.ca

Natalia Stakhanova
University of Saskatchewan
Saskatoon, Canada
natalia@cs.usask.ca

***Abstract*—The increasing use of large language models (LLMs) such as OpenAI's ChatGPT and Google's Bard in the software development industry raise questions about the security of generated code. Our research evaluates Java, C, and Python code samples that were generated by these LLMs. In our investigation, we assessed the consistency of code samples generated by each LLM, characterized the security of generated code, and asked both LLMs to evaluate and fix the weaknesses of their own generated code as well as the code of the other LLM. Using 133 unique prompts from Google Code Jam competitions, we produced 3,854 code samples across three distinct programming languages. We found that the code produced by these LLMs is frequently insecure and prone to weaknesses and vulnerabilities. This concerns human developers who must exercise caution while employing these LLMs.**

***Index Terms*—AI-generated code, ChatGPT, Bard, vulnerabilities**

I. Introduction

Increasing demands in the software development industry often requires programmers to create code under tight time constraints. Consequently, the widespread availability of applications that can automatically generate code based on clear descriptive prompts has compelled developers to rely on them to streamline their daily tasks. This trend has been accelerated by the rapid advancements in AI and the emergence of large language models (LLMs) that form the foundation of tools such as ChatGPT [1] and Google Bard [2].

Numerous studies have since explored the quality of code produced by these LLM tools by evaluating their correctness, validity, dependability, and maintainability [3]–[6]. Yet, among these considerations, the security aspect is often overlooked. A handful of recent research studies have aimed to explore the presence of vulnerabilities in code generated by LLM-powered tools [7]–[9]. Most of these studies are limited to a small set of pre-defined CWEs and therefore do not provide a thorough understanding of the security of LLM-generated code. Developers increasingly use LLM tools to aid in code generation. The increase in the number of systems that integrate LLMs into a development pipeline demands nuanced understanding of their capabilities to produce secure code and reason about its weaknesses and proper mitigation steps.

In this paper, we aim to address this gap and ① provide a comprehensive analysis of OpenAI ChatGPT's and Google Bard's (currently known as Gemini) abilities to generate secure code, and ② the abilities of these platforms to detect and correct vulnerabilities in insecure code.

Our goal in this analysis is to provide a realistic assessment of code produced by the ChatGPT and Bard. For this analysis, we generated 3,854 programs in three different languages: C, Java, and Python using ChatGPT (curated from GPT-4) and Bard. To produce a comparable set of programs for analysis, we used a set of 133 programming tasks extracted from the Google Code Jam (GCJ) competition as prompts.

We analyze the generated programs using state-of-the-art vulnerability assessment tools to identify security weaknesses present in the generated code. This assessment serves as the ground truth for our analysis. We manually verify this assessment. We further conduct a vulnerability assessment using LLM tools to evaluate their ability to detect the presence of code weaknesses, correct any identified weaknesses, and provide secure solutions. This approach resembles a typical code security testing process used by software developers.

Our findings unequivocally demonstrate that both models tend to produce insecure code. Baseline vulnerability analysis revealed that both LLMs produce vulnerable code half of the times. Yet, all C code generated by both tools is vulnerable. Despite this, both LLM tools are generally capable of detecting security vulnerabilities/weaknesses in their Java and Python code, but not in C code. ChatGPT detected security problems in 90.6% of its Java and 63.6% of its Python programs, whereas Bard reported problems in all its vulnerable code. Neither of LLMs were able to accurately detect the vulnerable C samples. When asked to evaluate each other's code, both LLMs performed significantly better than on their own code, and surprisingly, were able to detect a significant portion of insecure C code.

II. Related work

Numerous studies have investigated code quality. Nguyen et al. [10] evaluated the code samples generated by Copilot, noting that the complexity of its solutions did not vary significantly between solutions to the same problem across different languages. Sobania et al. [11] observed that it was often easier to understand Copilot's code compared to code generated by generic programming paradigms. Tian et al. [12] examined ChatGPT's ability to generate, fix, and summarize code. Yetiştiren [6]'s study revealed that ChatGPT generated perfect code solutions to 65.2% of their provided prompts.

Other studies have concentrated on code testing and debugging with ChatGPT. Xia et al. [13] proposed ChatRepair, which used ChatGPT to repair programs. Jalil et al. [14] evaluated ChatGPT's ability to assess the correctness of LLM-produced code. The findings from Sobania et al. [15] indicated that ChatGPT performance was competitive to results

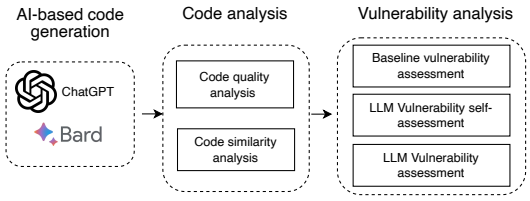

Fig. 1. The flow of the analysis

produced by the CoCoNut [16] and Codex [17] tools. Other research has evaluated LLMs' capabilities in debugging and repairing their own generated code [18], [19]. Several studies compared the correctness of code generated by ChatGPT and Bard [20]. Most existing studies focus on the security of code from proprietary LLMs. Pearce et al. [8] provided an early investigation of vulnerabilities in AI-generated code, finding that approximately 40% of the code snippets generated by Copilot contained vulnerabilities. Fu et al. [21] constructed a dataset of Copilot-generated code samples from GitHub with 35.8% of the samples containing security weaknesses. Tihanyi et al. [22] analyzed C code generated by GPT-3.5-turbo and found that the generated programs contained risky coding practices. Perry et al.'s [23] results indicated that programmers wrote significantly less secure code when they had access to an AI code assistant. After constructing a ChatGPT-generated Java code dataset, Mousavi et al. [24] identified the misuse of security APIs in the majority of their samples.

Only recently, the research emphasis has shifted towards using LLM-powered tools to evaluate code security. Ridley et al. [25] compared the performances of popular open-source LLMs in identifying security weaknesses.

There is a significant amount of research [26]–[28] on detecting vulnerabilities in code samples that were generated using prompts targeting a specific vulnerability. Our focus is to provide far broader insight into AI-generated code security while also mimicking a real development environment.

III. Methodology

Our focus in this work is to assess the security of code generated by two LLM tools. Figure 1 presents a conceptual flow of our analysis.

A. AI-based Code Generation

For our analysis, we selected two LLM-powered tools: Google Bard (renamed as Gemini in 2024)[1] and OpenAI ChatGPT. Given the absence of API access to GPT-4 and Bard at the time of our analysis, we manually collected responses from both models, and for the remainder of the process, we used custom Python and shell scripts to execute the tools and analyze their output.

To ensure the comparability of code generated by different AI tools, we employed all available 133 problem descriptions

[1]This analysis was performed in 2023.

TABLE I
Prompt Description

No.	Prompt
1	Write a Java/Python/C program with the description: <problem description>
2	Give an alternate implementation from the above
3	Provide a different implementation from the above
4	Give an alternate way of implementation
5	Provide a different approach of implementation

TABLE II
LLMs generated data

	Bard			ChatGPT		
	Total files	Correct syntax	Similarity ≤70%	Total files	Correct syntax	Similarity ≤70%
Java	640	505	192	610	585	535
Python	665	618	244	609	606	592
C	665	556	370	665	640	631
Total	1970	1679	806	1884	1831	1758

sourced from the Google Code Jam (GCJ)[2] coding competition, spanning from 2018 to 2022, as prompts. Each of these prompts was presented to ChatGPT and Google Bard with requests to develop code solutions in Java, Python, and C programming languages. Recognizing that LLMs may yield different solutions to the same problem, we diversified the initial prompts by instructing the LLMs to generate alternative solutions. These prompts are detailed in Table I. Consequently, we aimed to generate five solutions for each GCJ problem, resulting in a possible maximum of 665 code samples for each programming language and LLM combination. However, tools did not provide five code samples for a small number of code prompts (giving errors). Thus, our final dataset consisted of 3,854 programs.

B. Code analysis

We observed that the generated code samples ranged drastically in quality and size. Despite the identical prompts, code generated by each LLM varied in size, amount of comments, syntax, and code structure for each of the three programming languages. This LLM behavior was noted by previous studies [29] and was therefore expected.

Yet, we also observed a significant similarity between some of the programs generated in response to a given prompt. To evaluate this visible code similarity, we used Moss [30], a state-of-the-art code similarity detection system. In cases when the code similarity between programs was above the threshold of 70%, we only retained one of the samples in our dataset, since their structure of these programs lacked any significant distinction between them.

C. Vulnerability analysis

We performed vulnerability assessment following four steps: ① We conducted a baseline assessment by a set of state-of-the-art vulnerability assessment tools: Bandit [31], CodeQL [32], Semgrep [33], DevSkim [34], and Snyk [35]. These results served as ground truth for the presence of weaknesses (CWE) and vulnerabilities (CVE) in the generated

[2]The GCJ competition was discontionued in 2023. Problem data is now available on GitHub (https://github.com/google/coding-competitions-archive).

code. ② Vulnerability self-assessment was then conducted by LLM tools to evaluate the tools' abilities to recognize and correct security weaknesses and vulnerabilities in their own code. ③ The next step assessed the general ability of LLMs to detect and address potential security flaws in code samples written by another LLM. ④ The last step was to use static security analysis tools to verify each LLM's capacity to correct the insecure versions of the code modified in ② and ③.

1) Baseline vulnerability assessment: We use five open-source security analyzers on the three language datasets collected from both ChatGPT and Google Bard: Bandit [31], CodeQL [32], DevSkim [34], and Snyk [35]. All analyzers build an abstract syntax tree (AST), which is then analyzed using predefined weakness patterns.

2) LLM vulnerability self-assessment: After compiling the empirical data, we direct each LLM tool to evaluate the security of the code samples it generated.

The analysis of vulnerabilities is a primary metric for measuring code security. Insecure patterns commonly seen in code are categorized as weaknesses and are formally catalogued as CWEs (Common Weakness Enumerations). The specific instances of CWEs detected in implementations of software and hardware are referred to as CVEs (Common Vulnerabilities and Exposures).

We asked the LLM tools to determine if a provided code sample was secure, and if it was not, we further prompted the LLM tool to identify CWEs and/or CVEs in the sample. In the case of insecure code, we asked the LLM tool to generate a secure solution to address the security concerns that it identified. We queried the models with the following questions for each code sample:

(i) Is this code secure?
(ii) Identify any applicable CWE-ID and provide a secure solution.
(iii) Identify any applicable vulnerability and/or CVE and provide a secure solution.

In an effort to reduce the number of queries given to the models, we did not ask the second and third questions, if the LLM tool reported that the code sample was secure.

3) Assessment of code generated by other LLM: After the LLMs' self-assessments, we asked each models to assess the security of code written by the other LLM. The LLMs were asked the same three questions outlined above with the same approach of recording the corrected code as we did with the self-assessment step.

4) Evaluation of the LLM vulnerability corrections: Finally, all corrected versions of code produced during the LLMs' assessments were collected and analyzed using the vulnerability assessment tools employed during our baseline analysis (i.e., CodeQL, DevSkim, Semgrep, Snyk, and Bandit).

IV. Data

The overview of the code samples generated by ChatGPT and Bard is presented in Table II. Since many of the vulnerability assessment tools produced an AST for analysis, it was necessary to filter out LLM-generated samples that were syntactically invalid or incomplete. To retain as many code samples as possible, we manually fixed small syntactic errors (e.g., missing import statements, unmatched brackets). Code samples that required significant corrections were removed from our dataset.

As our results in Table II show, ChatGPT outperformed Bard in terms of producing syntactically correct and diverse (with a similarity $<$70%) code samples. Only a small fraction of the programs generated by ChatGPT were very similar, i.e., only small differences in variable and function names. For example, 10% of the Java programs in the ChatGPT-generated dataset were above the similarity threshold with even smaller percentages of the Python and C datasets exceeding this threshold. Thus, we retained 93% (1758) of ChatGPT samples.

On the other hand, more than half of the programs in the Bard-generated dataset were eliminated after the similarity filter. We retained 39% of the programs in the cases of Java and Python, and 65% of the programs in the C dataset.

While Bard responded to the majority of the GCJ problem statements, ChatGPT was reluctant to respond to problem prompts from the final rounds of each year's GCJ, often stating that the problem statements were too complex for it to solve.

TABLE III
THE SUMMARY OF BASELINE ANALYSIS

		Total Samples	Verified Vulnerable samples	Number of CWEs		
				Min	Max	Avg.
Java	ChatGPT	535	202 (38%)	1	2	1.17
	Bard	192	67 (35%)	1	2	1.09
Python	ChatGPT	592	22 (4%)	1	2	1.05
	Bard	244	10 (4%)	1	1	1
C	ChatGPT	631	631 (100%)	1	6	3.07
	Bard	370	370 (100%)	1	6	2.41
Total	ChatGPT	1758	854 (49%)	1	6	2.57
	Bard	806	447 (55.5%)	1	6	2.18

V. Baseline Vulnerability Analysis

The overview of the analysis conducted by vulnerability assessment tools is provided in Table III. In total, both ChatGPT and Bard produced vulnerable code half of the time, i.e., 49% of all samples generated by ChatGPT and 55.5% Bard samples. Almost all programs written in C programming language were insecure (99.8% and 100%, respectively) and contained on average two to three weaknesses. 38% (ChatGPT) and 35% (Bard) of generated Java samples contained at least one CWE. The Python samples were generally found to be the least insecure, i.e., four percent of all samples contained weaknesses, potentially due to the fact that only one Python-specific vulnerability scanner was used in this analysis.

Further analysis of weaknesses reported by the tools in baseline analysis shows the striking disagreement between tools. In total, the tools reported six CWEs in Java code, two CWEs in Python code and 15 CWEs in C samples (Table IV). Of these, three were featured in Mitre's 2022 CWE Top 25 Most Dangerous Software Weaknesses list [36]. All reported CWEs are unique with one exception. Two tools, Semgrep and Devskim, agreed on one CWE (CWE-676).

TABLE IV
BASELINE VULNERABILITY RESULTS

Language	Tool	CWE-ID	ChatGPT	Bard
Java	CodeQL [32]	CWE-129	62	11
		CWE-134	1	-
		CWE-190*	165	62
		CWE-681	2	-
		Total Files	195	67
	Devskim [34]	CWE-338	3	-
		CWE-546	7	-
		Total Files	9	-
Python	Bandit [31]	CWE-330	19	10
		CWE-703	3	-
		Total Files	22	10
C	CodeQL [32]	CWE-119*	4	1
		CWE-120	154	86
		CWE-129	62	32
		CWE-190	330	183
		Total Files	429	237
	Devskim [34]	CWE-242	629	368
		CWE-337	5	-
		CWE-338	5	40
		CWE-546	7	-
		CWE-676	98	94
		Total Files	630	370
	Semgrep [33]	CWE-14	108	51
		CWE-676	629	367
		Total Files	629	367
	Snyk [35]	CWE-122	5	-
		CWE-369	1	4
		CWE-401	1	5
		CWE-476*	1	-
		Total Files	8	9

* included in Mitre's 2022 CWE Top 25 most dangerous software weaknesses list

a) Manual verification: Due to clear disagreement between tools on the presence of vulnerabilities in LLM-generated samples, we manually verified each sample reported by at least one of the tools as vulnerable. All samples reported as vulnerable contained at least one vulnerability.

VI. LLM VULNERABILTY SELF-ASSESSMENT

TABLE V
STATISTICS OF LLMS ASSESSMENT

	Self-assessment			
	Bard–Bard			
	Secure	PV	V	True DR
Java	1 (0.5%)	-	191 (99.5%)	100%
Python	-	-	244 (100%)	100%
C	40 (11%)	-	330 (89%)	0%
	ChatGPT–ChatGPT			
Java	267 (50%)	220 (41%)	48 (9%)	90.6%
Python	316 (53%)	236 (40%)	40 (7%)	63.6%
C	587 (93%)	44 (7%)	-	7%
	Assessment of code generated by other LLM			
	ChatGPT Assessment of Bard samples			
	Secure	PV	V	True DR
Java	39 (20%)	153 (80%)	-	79.1%
Python	112 (46%)	109 (45%)	23 (9%)	50%
C	148 (40%)	70 (19%)	152 (41%)	60%
	Bard Assessment of ChatGPT samples			
Java	115 (21%)	223 (42%)	197 (37%)	88.1%
Python	51 (9%)	12 (2%)	529 (89%)	100%
C	177 (28%)	6 (1%)	448 (71%)	71.9%

PV - Potentially Vulnerable, V - Vulnerable, DR - Detection Rate

One of the goals of this study is to characterize LLMs' abilities to recognize vulnerable code. The responses of ChatGPT and Bard to being prompted to self-assess their own code are organized as follows:

1) Secure:

- Bard typically replied with *"The code you provided is secure and does not contain any vulnerabilities. The CWE-ID for this vulnerability is None."*
- ChatGPT replied with "The code provided does not appear to contain any known vulnerabilities according to the Common Weakness Enumeration (CWE) standards as of my training data cut-off in September 2021."

2) Vulnerable (V):

- Bard responded with *"The code is not secure because ... The CWE-ID for this vulnerability is CWE-..."*
- ChatGPT answered with *"The provided code has two main security vulnerabilities ... "*

3) Potentially vulnerable (PV):

- Bard responded with *"The code you provided is secure and does not contain any vulnerabilities. The CWE-ID for this code is CWE-..."*
- ChatGPT's uncertainty manifested with *"The code provided does not seem to have explicit security vulnerabilities. There is a potential for ..."*

To aid in the analysis of self-assessment results, we measure the detection rate (DR) as follows:

$$DR = \frac{PV + V}{Number\_of\_vulnerable\_samples} \quad (1)$$

This helps analyze the ability of LLMs to recognize verified vulnerable samples.

A. ChatGPT self-assessment

In contrast to the baseline vulnerability results, ChatGPT identified more samples as being vulnerable. While the vulnerability assessment tools revealed a maximum of two CWEs, a sample evaluated using ChatGPT was discovered to have three to four CWEs. In comparison to the baseline results, the range of CWEs identified by ChatGPT is also broad. We randomly chose some samples and analyzed if the samples were vulnerable to the reported CWEs and observed that the explanation matched with the reported CWEs.

In Java, 220 (41%) out of the 535 programs submitted to ChatGPT were found to be not secure, 267 (50%) were reported to be secure, and the remaining 48 (9%) were potentially vulnerable. *This self-assessment for the most part is accurate, 183 (90.6%) vulnerable Java samples were reported as vulnerable or potentially vulnerable.*

For our Python dataset, 236 (40%) programs out of 594 were not secure, 316 (53%) were secure and the remainder 40 (7%) were seen as possibly prone to vulnerabilities. *The detection of Python vulnerable samples was less accurate with only 10 samples detected as vulnerable and 4 as potentially vulnerable, resulting in a total detection rate of 63.6%.*

ChatGPT reported that 44 (7%) programs out of 631 in our C dataset were not secure while the remaining 587 (93%) programs were secure. *The analysis of C samples was the least accurate as only 44 vulnerable samples were detected resulting in a 7% detection rate.*

ChatGPT detected 32 different CWEs in the Java programs with 67% of the programs vulnerable to the top 10 CWEs of the total identified weaknesses. ChatGPT also identified specific combinations of CWE IDs. Out of these programs, (32%) account for CWE-20, where the input read by the end user is

not validated. Approximately, 10% of the programs observed belong to 'class' CWE-404 (*Improper Resource Shutdown or Release*). This weakness is related to not releasing/incorrectly releasing a resource before it is made available for re-use.

The combination of CWE-129 and CWE-190 (inadequate validation of array index and integer overflow or wraparound, respectively) affected around 10% of programs.

ChatGPT detected a total of 24 CWEs in Python with 72% of the programs being vulnerable to the top 10 CWEs. The most frequently observed pattern is CWE-20 with CWE-400. Other common patterns are CWE-20 with CWE-252, CWE-20 with CWE-400, and CWE-20 with CWE-754. A majority of these programs (47%) account for CWE-20. The next widely seen weakness is the class of CWE-400 and its base CWE-770, which is identified in 8% of the programs. This weakness occurs when the code does not impose restrictions to control the allocation of resources.

As well, 7% of programs contain the combination of the 'class' CWE-754: *Improper Check for Unusual or Exceptional Conditions* and its descendants, CWE-252 and CWE-476.

ChatGPT detected 20 CWEs in C with CWEs from the top 10 list present in 91% of the insecure programs, which is 10% of the *total* programs that we presented to ChatGPT. CWE-20 with CWE-120 was the most common combination. Nearly 80% of the vulnerable programs were prone to CWE-120, which occurs when a buffer is copied without checking the size of the input. The next frequently identified weakness in the programs was CWE-20.

B. Bard self-assessment

Bard identified weaknesses in the majority of its code. When scanned with the vulnerability assessment tools, the Java and Python samples were determined to be less vulnerable than with Bard's self-evaluation. The tools recognized six CWEs in a C code sample whereas Bard could find only one CWE.

Out of the 192 Java programs, Bard identified 191 (99.5%) as insecure. Similarly, 330 (89%) out of 370 code samples written in C were insecure. In the case of Python, all the programs (100%) assessed by Bard were insecure.

The self-assessment of Java and Python code was remarkably accurate, with all vulnerable samples detected by Bard as vulnerable. Surprisingly, the results of C code analysis were the opposite - none of the vulnerable C samples were detected by Bard.

Bard found 31 CWEs in Java, with CWEs from the top 10 CWEs list affecting 92% of the programs. The majority of these CWEs are from the 'pillar' CWE-707: Improper Neutralization, which has data neutralization concerns based on CWEs 74, 78, and 79. Since the GCJ problem definitions that we used in this study do not involve operating system (OS) coding, web development, or upstream/downstream components, *Bard misrepresented the security issues of programs*. The second prominent weakness was "class" CWE-20, which is contained in 26% of programs and falls into the same pillar. CWE-119, a parent of CWE-190, and CWE-805, a descendant of CWE-20, was present in 12% of vulnerable programs.

Bard uncovered 43 CWEs in Python with 76% of the programs vulnerable to CWEs belonging to the top ten CWE list. A significant portion (28%) of the samples had data neutralization concerns, which was indicated by the presence of CWEs 76, 78, and 79 ('category' CWE-137). However, once again, the problem descriptions that we used in this study are not related to OS operations, webpage generation, or special elements. CWE-20 was the next most common weakness, and it was identified in 18% of programs. A substantial 11% of the applications had memory issues such as excessively allocating memory (CWE-789) or out-of-bounds write errors (CWE-787).

Bard identified a total of 29 CWEs in the C dataset with CWEs from the top 10 CWEs appearing in 79% of the programs. The most frequently occurring patterns were CWE-78 with CWE-120, CWE-78 with CWE-352, and CWE-78 with CWE-80 and CWE-119. A significant proportion (56%) of the problems were prone to inappropriate neutralization problems (CWEs 78, 79, 80, and 89). Additionally, 35% of the programs fall under the 'category' CWE-1218: *Memory Buffer Errors*, which includes the following weaknesses: failing to check the size of input (CWE-120), out-of-bounds read (CWE-125), buffer over-read (CWE-126), and buffer access with incorrect length value (CWE-805).

VII. LLM Vulnerability Assessment

Next we evaluate ChatGPT's and Bard's abilities to assess the security of code written by other LLMs.

A. ChatGPT assessment of Bard responses

ChatGPT identified 39 (20%) Java programs out of the 192 programs as secure while the remaining programs were potentially vulnerable. Out of 244 Python programs, 23 (9%) were not secure, 112 (46%) were secure, and 109 (45%) were possibly vulnerability-prone. There were 152 (41%) C programs out of 370 programs that were insecure, 70 (19%) programs were likely vulnerable, and 148 (40%) were secure.

The ChatGPT assessment of Bard's vulnerable Java and Python samples was similar to its self-assessment results, with a detection rate of 79 and 50% respectively. Interestingly, ChatGPT performed significantly better on Bard's C code, being able to detect 60% of its vulnerable samples.

ChatGPT discovered 43 CWEs total in Java with 91% of the programs vulnerable to CWEs from the top 10 CWEs list. ChatGPT also detected combinations of CWE IDs in the samples. The majority of these programs (74%) contained CWE-20 where the input read by the end user is not validated. Approximately, 25% of the evaluated programs contained weaknesses belonging to 'class' CWE-664 *Improper Control of a Resource Through its Lifetime* (uncontrolled resource consumption and undesirable resource shutdown or release).

CWE-129 (*Improper Validation of Array Index*), CWE-476 (*Dereferencing a NULL pointer*), CWE-682 (*Incorrect Calculation*), and CWE-754 (*Improper Check for Unusual or Exceptional Conditions*), which altogether account for 33% of the total problems, are the other key vulnerabilities seen.

ChatGPT detected 31 CWEs in Python with 86% of the programs vulnerable to CWEs from the top 10 CWEs list. The most frequently observed pattern was CWE-20 with CWE-400. A majority of these programs (67%) contained CWE-20. The next widely seen weakness was identified in 21% of the programs (CWE-400: *Uncontrolled Resource Consumption*), which occurs when the code does not control the allocation or maintenance of a limited resource, leading to its exhaustion.

ChatGPT detected 54 CWEs in C with CWEs that belong to the top 10 CWEs list present in 84% of the programs. CWE-20 with CWE-120, CWE-457, and CWE-676 was the most common combination. More than half of the programs (53%) were vulnerable to the 'category' CWE-1218: Memory Buffer Errors with a majority relating to buffer copy without checking the size of the input (CWE-120) and a handful of samples contained CWE-131 (improper buffer size calculation).

CWE-252: *Unchecked Return Value*, which is preceded by CWE-476: *NULL Pointer dereferencing* occured in 23% of programs. Bard used uninitialized variables (CWE-457) in 17% of the programs it created. ChatGPT identified 13% of programs using harmful methods (CWE-676) and nine percent of samples that incorrectly validated array indices (CWE-129).

B. Bard assessment of ChatGPT responses

Out of the 535 Java programs submitted to Bard, it identified 197 (37%) programs as insecure, 115 (21.5%) samples as secure, and 223 (41.5%) programs as potentially vulnerable. In the case of our Python dataset, 531 (89%) programs out of 594 were considered to be insecure, 51 (9%) were secure and 12 (2%) were potentially vulnerable. Of the 649 C samples submitted to Bard, 462 (71%) programs were insecure, and 6 (1%) were potentially vulnerable; the remaining 184 (28%) samples were reported as secure.

These assessment results were quite accurate, as the majority of Java, Python and C samples were detected, with 88.1%, 100%, and 71.9% detection rate, respectively.

Bard discovered a total of 87 CWEs in our Java dataset, and 55% of the programs were vulnerable to CWEs in the top 10 CWEs. The majority of these CWEs belong to the 'view' CWE-699: *Software development*. We observed that 25% of the solutions had data neutralization issues (i.e., they contained one or more of CWEs 78, 79, or 89). Our problem descriptions did not involve performing operations on OS, databases, or webpage generation. Therefore, Bard erroneously represented the security problems of the programs.

CWE-20 appeared in 26 solutions where the input recorded in variables is inadequately verified and accounts for 6% of the total issues. This class of weakness precedes CWE-119: *Improper Restriction of Operations within the Bounds of a Memory Buffer* followed by CWE-190, where the application performs a calculation leading to integer overflow or wraparound. Although CWE-190 accounts for only a minor portion, its presence can result in undefined behaviour and crashes. CWE-787, a child of CWE-119, was also seen in 5% of the samples. This weakness occurs when data is written before the beginning of the buffer or past the end of the buffer.

The next most frequently reported weakness is CWE-835: *Loop with Unreachable Exit Condition ('Infinite Loop')*.

Bard detected a total of 63 CWEs in Python with 75% of the programs containing CWEs that are in the top 10 CWEs. The most frequently observed patterns were CWE-20 with CWE-815, and CWE-78 with CWE-839 (*Numeric range comparison without minimum check*). A notable portion of these programs (24%) accounted for CWE-78. 37% of the solutions had data neutralization issues due to the presence of CWEs 78, 79, or 89. The next widely seen weakness identified in 10% of the programs was CWE-20.

CWE-805: *Buffer Access with Incorrect Length Value* was the next most prevalent weakness observed in 10% of the Bard-generated code.

Bard identified a total of 44 CWEs in the C dataset with CWEs from the top 10 CWEs indicating risk to 57% of the programs. A significant proportion (51%) of problems were prone to inappropriate neutralization problems indicated by the presence of CWEs 78, 79, 80, or 89 in the code samples. 30% of the programs fall under the 'category' CWE-1218: *Memory Buffer Errors* when operations such as buffer copy without checking the size of input (CWE-120), out-of-bounds read (CWE-125), which includes buffer over/under-read(CWEs 126 and 127, respectively), and buffer access with incorrect length value (CWE-805) are performed.

Other significant flaws are CWE-20 (input validation), and CWE-119 (inappropriate restriction of operations within the boundaries of a memory buffer). These CWEs account for 11% of all issues observed in the C programs.

VIII. Evaluation of the LLM Vulnerability Corrections

As the final step of our analysis, we validated the secure versions of code produced by LLM tools (during self-assessment and assessment of other LLM code) in an effort to mitigate the vulnerabilities and weaknesses they identified in the original code. Similar to our baseline analysis, we used five vulnerability assessment tools. Results are shown in Table VI.

Overall, the number of samples reported with CWEs decreased. However, the total number of unique CWEs reported for the Java and Python languages (6 and 2, respectively) remained unchanged. The number of CWEs found in C code increased to 21. Interestingly, by addressing the presence of one CWE (CWE-242), LLMs introduced 8 additional CWEs.

Nonetheless, new CWEs emerged. The number of Java samples susceptible to vulnerabilities was reduced by half with the remaining half of samples still marked as vulnerable. For example, Bard addressed all detected CWE-338 and CWE-546 weaknesses in the Java code produced by ChatGPT.

In contrast, despite the LLMs fixing the problem in Python, the number of vulnerable samples increased. For example, when ChatGPT samples were assessed by ChatGPT, the number of samples vulnerable to CWE-703 increased and decreased for CWE-330. Similarly, when Bard fixed the ChatGPT code, the samples vulnerable to both CWEs increased.

For samples produced using the C programming language, the number of detected CWEs jumped from 14 to 21, with the identification of a total of 8 new CWEs by fixing one of the existing CWEs. These numbers significantly exceed the security vulnerabilities found in the Java and Python code.

Overall, ChatGPT performed better than Bard because the number of vulnerable C samples was reduced after being evaluated and fixed by ChatGPT. Snyk [35] discovered instances of new weaknesses, such as CWE-125, CWE-415, CWE-416, and CWE-787, in the corrected C samples. CodeQL [32] also reported a new set of CWEs 22, 676, 732, and 764 in Bard's corrected code, but not in ChatGPT code.

IX. Discussion

Our experiments revealed an unfortunate security quality of ChatGPT and Bard supplied solutions to the majority of the prompts. Our findings emphasize several important issues:

- *Both ChatGPT and Bard produced vulnerable code half of the time*. Our analysis showed that 49% of all samples generated by ChatGPT and 55.5% Bard samples contained at least one CWE. Among all discovered CWEs, 3 appear in the top 25 CWE list of most dangerous software weaknesses [36].
- *The security of LLM-generated code varied depending on the programming language*. The C programs generated by ChatGPT and Bard were all insecure. Approximately one-third of generated Java samples contained at least one CWE, i.e., 38% ChatGPT and 35% Bard. The samples written in Python were generally found to be the least insecure as just 4% of all samples contained weaknesses.
- *Both ChatGPT and Bard accurately recognized their own Java and Python code as vulnerable*. Bard reported 100% of its vulnerable code as insecure; ChatGPT correctly detected 90.6% Java and 63.6% Python vulnerable samples.
- *Neither ChatGPT nor Bard recognized their C code as vulnerable*. Despite all of LLM-generated C code being vulnerable, none of the tools were able to properly detect it. Bard did not detect any of its vulnerable C samples, and ChatGPT detected only 7% of its samples.
- *Both ChatGPT and Bard performed better on each other's C code than their own code*. Both LLMs largely detected vulnerable C code. Bard detected 71.9% of ChatGPT's vulnerable C samples. On the other hand, ChatGPT identified 60% of Bard's vulnerable C code as weak.
- *ChatGPT and Bard corrected vulnerable code, but at the same time introduced new vulnerabilities*. After the vulnerable code was fixed by both ChatGPT and Bard, less than 50% of the Java samples still contained weaknesses, and no new flaws were identified. Similarly, Bard identified an increased number of Python samples containing weaknesses despite no new vulnerabilities being detected. 8 new CWEs were found in the C samples generated by ChatGPT while one existing CWE was fixed. Overall, we noted a decrease of 25% in the number of vulnerable samples in the corrected dataset. Bard introduced 7 new CWEs while fixing the vulnerable ChatGPT and its code, while 3 new CWEs were introduced by ChatGPT when it corrected Bard code samples.

  During our analysis, it became apparent that Bard's vulnerability-identification capabilities are less than adequate. In one case, Bard offered a CVE number that was unrelated to the identified vulnerabilities and did not match the program that it was evaluating. For example, Bard identified a program that was vulnerable to CWE-120 'Buffer Copy without Checking Size of Input' and labelled it as CVE-2006-3916, which is a cross-site scripting (XSS) vulnerability in sNews (also known as Solucija News) 1.4 that enables remote attackers to inject arbitrary web scripts or HTML via the search_query parameter. It is completely irrelevant to the identified CWE.
- *In general, LLM tools' behaviour raised questions regarding their code-generating capabilities*. For example, when the output tokens surpassed the limit, both tools offered incomplete answers for a few problem statements. The LLM-generated code often employed a number of classes, and in some situations, it did not offer an implementation of a class but nevertheless made use of it.

Limitations.

In this study, LLMs generated code from natural language descriptions of tasks, which is a common real-world scenario of how developers use LLMs in software development. We did not test if the code produced by the LLMs was executable. We believe this is irrelevant for our study as developers may be seeking entire programs, small code snippets, or even auto-completion suggestions for partially written code.

As our analysis showed, LLMs may produce incomplete or syntactically invalid code. This reflects the current state of LLM capabilities and is likely to be similarly addressed by developers, i.e., corrected when possible or dismissed, as we did in this study. Some of the vulnerability assessment tools we used in this study relied on a code AST for analysis; hence, it was necessary for us to ensure that syntactically valid code is used in the analysis. Note that having properly executable code is not required for vulnerability analysis.

Vulnerability analysis tools in our analysis showed significant disagreement in their assessment. For example, CodeQL, built with queries designed for a limited subset of CWEs, does not offer support for various other CWEs found within our analyzed code. Furthermore, even among the CWE rules supported for the C language, there were instances where some samples that are vulnerable to CWE-20, could not be recognized by the tool. To ensure that baseline results are reliable, we performed a manual verification of vulnerable samples. Our manual validation showed that cumulatively there were no false positives, and all code samples detected as vulnerable by at least one tool contained at least one vulnerability. These results are generally inline with the limitations of vulnerability assessment tools previously explored by other studies [37].

X. Conclusion

In this study, we analyzed the security of code generated by LLMs as well as the ability of these LLMs to assess

TABLE VI
EVALUATION OF LLM VULNERABILITY ASSESSMENT

			Analysis of ChatGPT code		Analysis of Bard code	
			ChatGPT	Bard	ChatGPT	Bard
Java	CodeQL [32]	CWE-129	32	38	11	7
		CWE-134	-	1	-	-
		CWE-190	60	89	37	39
		CWE-681	1	1	-	-
	Devskim [34]	CWE-338	1	2	-	-
		CWE-546	2	3	-	-
Python	Bandit [31]	CWE-330	11	28	4	12
		CWE-703	10	10	10	-
C	CodeQL [32]	CWE-22*	-	2	-	-
		CWE-119	-	5	-	2
		CWE-120	8	126	16	54
		CWE-129	-	29	7	10
		CWE-190	16	187	116	99
		CWE-676*	-	1	-	-
		CWE-732*	-	5	-	-
		CWE-764*	-	1	-	-
	Devskim [34]	CWE-227*	41	472	206	274
		CWE-337	-	15	10	5
		CWE-338	-	12	27	14
		CWE-546	1	22	-	-
		CWE-676	21	134	81	80
	Semgrep [33]	CWE-14	8	74	25	44
		CWE-676	41	428	197	263
	Snyk [35]	CWE-122	-	6	-	-
		CWE-125*	-	-	-	9
		CWE-369	-	-	-	2
		CWE-401	1	6	6	6
		CWE-415*	-	1	1	-
		CWE-416*	-	2	1	-
		CWE-476	1	10	-	7
		CWE-787*	-	-	-	9

* Newly Discovered Weaknesses from the Fixed Versions

potentially insecure code. We used Google Bard and ChatGPT to generate 3,854 code samples in three different programming languages. Our analysis revealed the low security quality of code generated by ChatGPT and Bard. Given our findings, we recommend that developers exercise caution when incorporating AI-generated code into their codebase.

REFERENCES

[1] OpenAI, "Openai chat," 2022. [Online]. Available: https://chat.openai.com/chat

[2] Google, "An important next step on our ai journey," 2023. [Online]. Available: https://blog.google/technology/ai/bard-google-ai-search-updates/

[3] V. Adamson and J. Bägerfeldt, "Assessing the effectiveness of chatgpt in generating python code," 2023.

[4] C. Bull and A. Kharrufa, "Generative ai assistants in software development education: A vision for integrating generative ai into educational practice, not instinctively defending against it." *IEEE Software*, 2023.

[5] S. Sharma and B. Sodhi, "Calculating originality of llm assisted source code," *arXiv preprint arXiv:2307.04492*, 2023.

[6] B. Yetiştiren, I. Özsoy, M. Ayerdem, and E. Tüzün, "Evaluating the code quality of ai-assisted code generation tools: an empirical study on github copilot, amazon codewhisperer, and chatgpt. arxiv preprint arxiv: 230410778. 2023," *arXiv preprint arXiv:2304.10778*, 2024.

[7] R. Kande, H. Pearce, B. Tan, B. Dolan-Gavitt, S. Thakur, R. Karri, and J. Rajendran, "Llm-assisted generation of hardware assertions," *arXiv preprint arXiv:2306.14027*, 2023.

[8] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 754–768.

[9] G. Sandoval, H. Pearce, T. Nys, R. Karri, S. Garg, and B. Dolan-Gavitt, "Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants," in *USENIX*, 2023.

[10] N. Nguyen and S. Nadi, "An empirical evaluation of github copilot's code suggestions," in *Proceedings of MSR*, 2022, pp. 1–5.

[11] D. Sobania, M. Briesch, and F. Rothlauf, "Choose Your Programming Copilot: A Comparison of the Program Synthesis Performance of Github Copilot and Genetic Programming," in *Proceedings of GECCO*. New York, NY, USA: ACM, 2022, p. 1019–1027.

[12] H. Tian, W. Lu, T. Li, X. Tang, S. Cheung, J. Klein, and T. Bissyandé, "Is chatgpt the ultimate programming assistant—how far is it?(2023)," *arXiv preprint arXiv:2304.11938*.

[13] C. S. Xia and L. Zhang, "Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," *arXiv preprint arXiv:2304.00385*, 2023.

[14] S. Jalil, S. Rafi, T. D. LaToza, K. Moran, and W. Lam, "Chatgpt and software testing education: promises & perils (2023)," *arXiv preprint arXiv:2302.03287*, 2023.

[15] D. Sobania, M. Briesch, C. Hanna, and J. Petke, "An analysis of the automatic bug fixing performance of Chatgpt," in *Proceedings of ICSE*, 2023.

[16] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair," in *Proceedings of ISSTA*. New York, NY, USA: ACM, 2020, p. 101–114.

[17] J. A. Prenner, H. Babii, and R. Robbes, "Can openai's codex fix bugs? an evaluation on quixbugs," in *Proceedings of the Third International Workshop on Automated Program Repair*, ser. APR '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 69–75.

[18] K. Alrashedy and A. Aljasser, "Can llms patch security issues?" *arXiv preprint arXiv:2312.00024*, 2023.

[19] T. X. Olausson, J. P. Inala, C. Wang, J. Gao, and A. Solar-Lezama, "Is self-repair a silver bullet for code generation?" in *The Twelfth International Conference on Learning Representations*, 2023.

[20] Z. Liu, Y. Tang, X. Luo, Y. Zhou, and L. F. Zhang, "No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT," *IEEE Transactions on Software Engineering*, pp. 1–35, 2024.

[21] Y. Fu, P. Liang, A. Tahir, Z. Li, M. Shahin, and J. Yu, "Security weaknesses of copilot generated code in github," *arXiv preprint arXiv:2310.02059*, 2023.

[22] N. Tihanyi, T. Bisztray, R. Jain, M. A. Ferrag, L. C. Cordeiro, and V. Mavroeidis, "The FormAI Dataset: Generative AI in Software Security through the Lens of Formal Verification," in *Proceedings of PROMISE*. New York, NY, USA: ACM, 2023, p. 33–43.

[23] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do Users Write More Insecure Code with AI Assistants?" in *Proceedings of CCS*. New York, NY, USA: ACM, 2023, p. 2785–2799.

[24] Z. Mousavi, C. Islam, K. Moore, A. Abuadbba, and M. A. Babar, "An investigation into misuse of java security apis by large language models," *arXiv preprint arXiv:2404.03823*, 2024.

[25] N. Ridley, E. Branca, J. Kimber, and N. Stakhanova, "Enhancing Code Security Through Open-source Large Language Models: A Comparative Study," in *Proceedings of FPS*, 2023.

[26] R. Khoury, A. R. Avila, J. Brunelle, and B. M. Camara, "How secure is code generated by chatgpt?" in *2023 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2023, pp. 2445–2451.

[27] X. Liu, Y. Tan, Z. Xiao, J. Zhuge, and R. Zhou, "Not the end of story: An evaluation of ChatGPT-driven vulnerability description mappings," in *Findings of the Association for Computational Linguistics*. Toronto, Canada: ACL, Jul. 2023.

[28] M. Nair, R. Sadhukhan, and D. Mukhopadhyay, "Generating secure hardware using chatgpt resistant to cwes," Cryptology ePrint Archive, Paper 2023/212, 2023. [Online]. Available: https://eprint.iacr.org/2023/212

[29] S. H. Ambati, N. Stakhanova, and E. Branca, "Learning ai coding style for software plagiarism detection," in *Proceedings of SecureComm*, 2023.

[30] Moss, "Moss: A system for detecting software plagiarism," 2023. [Online]. Available: https://theory.stanford.edu/~aiken/moss/

[31] OpenStack, "Bandit," https://github.com/openstack-archive/bandit, 2022.

[32] Github, "Codeql for research." [Online]. Available: https://codeql.github.com/

[33] Semgrep, https://semgrep.dev, 2023.

[34] Microsoft, "Devskim," https://github.com/microsoft/devskim, 2023.

[35] Snyk, https://snyk.io/, 2023.

[36] MITRE, "2023 cwe top 25 most dangerous software weaknesses," https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html, 2023.

[37] K. Li, S. Chen, L. Fan, R. Feng, H. Liu, C. Liu, Y. Liu, and Y. Chen, "Comparison and evaluation on static application security testing (sast) tools for java," in *Proceedings of ESEC/FSE*. New York, NY, USA: ACM, 2023, p. 921–933.