

Secure Coding with AI – From Creation to Inspection

A Security Analysis of ChatGPT's Coding and Vulnerability Detection Skills

Vladislav Belozarov · Peter J Barclay ·
Ashkan Sami

Received: date / Accepted: date

Abstract While prior studies have explored security in code generated by ChatGPT and other Large Language Models, they were conducted in controlled experimental settings and did not use code generated or provided from actual developer interactions. This paper not only examines the security of code generated by ChatGPT based on real developer interactions, curated in the DevGPT dataset, but also assesses ChatGPT's capability to find and fix these vulnerabilities. We analysed 1,586 C, C++, and C# code snippets using static scanners, which detected potential issues in 124 files. After manual analysis, we selected 26 files with 32 confirmed vulnerabilities for further investigation.

We submitted these files to ChatGPT via the OpenAI API, asking it to detect security issues, identify the corresponding Common Weakness Enumeration numbers, and propose fixes. The responses and modified code were manually reviewed and re-scanned for vulnerabilities. ChatGPT successfully detected 18 out of 32 security issues and resolved 17 issues but failed to recognize or fix the remainder. Interestingly, only 10 vulnerabilities were resulted from the user prompts, while 22 were introduced by ChatGPT itself.

We highlight for developers that code generated by ChatGPT is more likely to contain vulnerabilities compared to their own code. Furthermore, at times

Vladislav Belozarov
Edinburgh Napier University
Edinburgh, Scotland
E-mail: vladislav.belozarov@gmail.com

Peter J Barclay
Edinburgh Napier University
Edinburgh, Scotland
E-mail: p.barclay@napier.ac.uk

Ashkan Sami (Corresponding Author)
Edinburgh Napier University
Edinburgh, Scotland
E-mail: a.sami@napier.ac.uk

ChatGPT reports incorrect information with apparent confidence, which may mislead less experienced developers. Our findings confirm previous studies in demonstrating that ChatGPT is not sufficiently reliable for generating secure code nor identifying all vulnerabilities, highlighting the continuing importance of static scanners and manual review.

Keywords software security · LLM · ChatGPT · static checkers · code generation · vulnerability detection · vulnerability mitigation · C-family

1 Introduction

The rapid advancement of technology and the increasing reliance on software in various industries have heightened the importance of secure software development. With the growing threat landscape, ensuring that software is robust and resilient against potential vulnerabilities is a critical concern for developers and organizations alike. Secure software coding practices have become an important aspect of the software development lifecycle, aiming to minimize security risks and protect sensitive data.

In recent years, the emergence of Generative AI as a software development tool, including use of ChatGPT, has revolutionized the software development process. ChatGPT, a large language model developed by OpenAI, has demonstrated remarkable capabilities in natural language processing, making it a valuable asset in various domains, including software coding. While ChatGPT can expedite the authoring of customized code, the security of such code is not clear. Here we explore the use of secure software coding practices when working with ChatGPT, examining the potential benefits and challenges associated with integrating AI tools into the software development process.

Previous research has primarily focused on evaluating the capabilities of Large Language Models (LLMs) to generate functional code and their likelihood of introducing vulnerabilities. For instance, studies have assessed the performance of models like Codex and GitHub Copilot in code generation tasks, highlighting instances where insecure coding practices were evidenced [17, 27, 28]. However, these investigations often utilized artificial prompts and scenarios crafted specifically for the experiments, which may not accurately reflect how developers interact with AI tools in practice.

This paper addresses this gap by examining the security of code generated by ChatGPT based on real developer interactions curated in the DevGPT dataset. This dataset provides a comprehensive collection of developer-AI exchanges, capturing the nuances of how developers utilize ChatGPT in practical coding tasks. Our study is unique because it relies on real interactions between users and ChatGPT, rather than on controlled lab experiments or specially crafted prompts. This hands-on approach gives us a more realistic view of ChatGPT's behaviour, setting our work apart from other studies that rely mostly on lab-based or simulated code generation.

We selected for analysis code snippets in C, C++, and C#, related languages which are widely used in various domains such as systems program-

ming, game development, and enterprise applications. Code snippets were extracted from DevGPT [11], a publicly available dataset, which contains developers’ interactions with ChatGPT, including prompts, answers, and generated source code to evaluate not only the security of the generated code, but also ChatGPT’s capability to detect and fix vulnerabilities within these real-world contexts.

To guide our investigation, we formulated the following research questions, shown below and previewing our findings:

- **RQ1:** *How secure is the code generated by ChatGPT in the DevGPT dataset?*

Answer: Our analysis revealed that out of 1,586 code snippets, static scanners detected potential issues in 124 files. After manual review, we confirmed 32 vulnerabilities in 26 files, indicating that the code generated by ChatGPT in the DevGPT dataset has security concerns.

- **RQ2:** *How effective is ChatGPT in detecting and correcting security issues when explicitly asked to do so?*

Answer: When explicitly prompted, ChatGPT successfully detected 18 out of the 32 confirmed vulnerabilities and resolved 17 issues, but failed to recognize or fix the remainder. This demonstrates partial effectiveness, but also highlights limitations in its ability to consistently identify and correct security vulnerabilities.

- **RQ3:** *Do developers provide more vulnerable code, or does ChatGPT generate more insecure code during the interactions?*

Answer: Interestingly, only 10 vulnerabilities were present in the user prompts, while 22 were introduced by ChatGPT. This indicates that ChatGPT generates more insecure code compared to the code provided by developers.

The contributions of this paper are threefold:

1. **Empirical Evaluation in Real-World Contexts:** While replicating the overall results of earlier studies, we then provide empirical evaluation of the security of AI-generated code within the context of actual developer interactions, addressing a gap in the prior research.
2. **Assessment of ChatGPT’s Security Capabilities:** Following earlier studies, we assess ChatGPT’s effectiveness in detecting and fixing vulnerabilities in code generated; however, we base this on real-world prompts and interactions of developers with ChatGPT.
3. **Implications for Developers and Tool Design:** Our findings underscore the importance of manual code review and caution against over-reliance on AI tools for secure code generation and analysis. We highlight the need for manual analysis and use of static checkers, noting that while AI tools can find vulnerabilities, they may also introduce them, and sometimes mislead developers regarding the security of the code under review.

The rest of the paper is organized as follows: Section 2 reviews related work on AI code generation and security. Section 3 details our data collection and methodology, including data extraction and analysis procedures. Section 4 presents our results and discusses the implications. Section 5 reviews

ChatGPT’s ability to detect and fix issues. Section 6 presents threats to the validity of our work, and Section 7 concludes the paper with suggestions for future research.

2 Related work

Use of LLMs, and particularly ChatGPT, for coding has become a common practice, and a variety of recent papers have addressed the security implications of this development [3, 5, 19, 16, 20, 23, 36]. Most published research focuses on either (1) assessing ChatGPT-generated code for vulnerabilities, or (2) using ChatGPT to detect security issues in existing code, which we discuss in the following two subsections.

2.1 Assessing ChatGPT-generated Code for Vulnerabilities

Bakhshandeh in [3] assesses ChatGPT-3.5’s performance in detecting vulnerabilities in Python source code, comparing to three traditional static tools: Bandit, Semgrep, and SonarQube. Here, 156 Python source code files were analysed using static analysers and ChatGPT. Experiments demonstrated that ChatGPT-3.5 outperformed the other tools in reducing false positives and negatives, making it a useful as a supporting tool. This study emphasized the importance of well-designed prompts to improve detection accuracy.

Jamdade in [20] explores ChatGPT’s ability to generate secure Node.js and PostgreSQL code for web applications, specifically addressing CWE-89 (SQL Injection), CWE-79 (Cross-Site Scripting), CWE-93 (CRLF Injection), and CWE-200 (Exposure of Sensitive Information) vulnerabilities. (CWE stands for Common Weakness Enumeration, a widely recognized classification system for software and hardware security weaknesses). Their paper concludes that ChatGPT can be guided to generate secure code through specific prompts. The authors developed a university library system as a case study, and iteratively prompted ChatGPT to improve the code; while improvements were observed, significant limitations in handling complicated scenarios still arose. These authors did not mention which ChatGPT model version was used for experiments.

Khoury et al. in the study [23] evaluate 21 simple programs generated by ChatGPT across five languages (C, C++, Python, HTML, and Java) for specific vulnerabilities, including SQL injection, memory corruption, and cryptographic misuse. Results showed that ChatGPT often generated insecure code, but could improve it when explicitly prompted about potential security problems. However, these “fixed” versions were still not robust against adversaries, often relying on naive mitigations like input validation or alphanumeric checks. The authors highlighted ChatGPT’s limited ability to generate secure code in advance, before the user explicitly asked it to generate secure code.

Hamer, in [19], compares Java code snippets generated by ChatGPT with StackOverflow [34] answers using CodeQL for vulnerability analysis. The study

analysed 108 snippets, revealing that ChatGPT produced 20% fewer vulnerabilities, but still propagated insecure patterns. ChatGPT's snippets contained 248 vulnerabilities spanning 19 CWE types, compared to 302 vulnerabilities across 22 CWEs in StackOverflow. However, the 274 unique security issues across both sources highlight the risks of blindly relying on any of these platform for secure coding. Developers were advised to follow secure coding practices and carefully inspect code obtained from external sources.

Kharma et al. [21] performed a comprehensive study on modern LLMs – including GPT-4, Claude-3.5, Codestral, Gemini-1.5, and Llama-3 – by testing them on 200 coding tasks across Python, Java, C, and C++. Although many of the generated solutions were functional, the authors identified several hidden security risks, such as the use of outdated libraries or missing input validations. Their work highlights a common problem: while LLMs can speed up code production, developers must still run security checks to prevent potential exploits.

Manik [26] conducted a direct comparison between ChatGPT-o1 and the newer LLM DeepSeek-R1 using an online platform Codeforces [8] to verify the correctness of the code produced, and the static checker tools Pylint and Flake8 for quality analysis. DeepSeek often solved the problems more accurately on its first attempt, but ChatGPT's outputs tended to be shorter and more readable. The downside of ChatGPT's more concise style was that it occasionally skipped security considerations, suggesting that features of convenience such as code brevity could overshadow vital checks.

Almanasra and Suwais [2] tested ChatGPT-4o on 600 tasks that comprised 300 data-structure problems and 300 LeetCode [25] challenges, split across Python and Java implementations. Their results showed that while ChatGPT-4 achieved a decent success rate and could handle even complex problems, it still left gaps in error handling and security measures. For example, the researchers found multiple instances of incomplete exception handling in both Python and Java solutions, emphasizing the requirement for human overview of the resultant code.

The studies highlighted above show the growing importance of code generation and its potential for improving software security, with ChatGPT showing promise but often generating insecure or incomplete outputs. Many researchers have highlighted the need for carefully crafted prompts, and the importance of reviewing the code generated.

2.2 Using ChatGPT as a Static Scanner

Wu et al. in [36] compare ChatGPT-3.5 and ChatGPT-4.0 in ability to perform seven security tasks, including vulnerability detection and repair, debugging, symbolic execution, and fuzzing. Using benchmark datasets and manual test cases, the study shows GPT-4 significantly outperforms GPT-3.5 in accuracy and effectiveness. ChatGPT demonstrated strong capabilities in structured

tasks like vulnerability detection or assembly code decompilation, but has difficulties with longer code contexts, binary code, and real-world complexities.

Cheshkov in [5] evaluates GPT-3 and GPT-3.5-turbo for vulnerability detection in Java code. Using a dataset of vulnerable and patched files, the models were assessed on binary and multi-label classification tasks for specific CWE types. The findings revealed significant limitations of ChatGPT, poor detection performance and biases favouring patched code. The study concludes that GPT-based models are currently inadequate for effective vulnerability detection without further refinement.

Fu et al. in [16] evaluate ChatGPT’s performance on four software vulnerability tasks: prediction, classification, severity estimation, and repair, using extensive datasets of C/C++ functions. ChatGPT underperforms compared to specialized models such as CodeBERT [7], highlighting the need for domain-specific fine-tuning. Despite its scale, ChatGPT struggles without adaptation for vulnerability contexts and demonstrates limited effectiveness in detecting vulnerabilities.

Kholoosi, Babar, and Croft [22] took a different angle by looking at real-world impressions from security professionals who tried ChatGPT (GPT-3.5-based) for tasks such as vulnerability detection and penetration testing. After collecting information from X/Twitter posts, the authors ran their own controlled experiments to see if ChatGPT’s advice was trustworthy. The results were mixed: participants found ChatGPT’s suggestions helpful for brainstorming or retrieving general security knowledge, but the authors’ tests showed it frequently overlooked subtle but important edge cases. This raises questions about whether ChatGPT can serve as a fully reliable “scanner” without additional tools or specialized training data.

Sajadi et al. [30] went further by testing Claude 3, GPT-4, and Llama 3 on real Stack Overflow snippets containing known vulnerabilities in Python or JavaScript. According to the authors, none of the models consistently issued security warnings in response to these faulty code examples unless the prompt explicitly asked about security. However, when the models did detect a problem, they produced surprisingly detailed explanations or fixes. These findings confirm that LLMs could grow into more robust scanning tools if provided with context-rich instructions with a strong focus on security from the outset.

2.3 Summary of Prior Research

All these studies show mixed results, with ChatGPT able to detect a range of vulnerabilities, but proving inadequate to give confidence that all problems would be identified.

The overview in Table 1 includes data for all these reviewed papers as well as this the current paper. The shortened column names in this table have the following meanings: “Lang” means “Programming Language”, “Detect” means “Is security issues detection skills tested”, and “Generat” means “Is code generation skills tested”. The data presented in the table highlight how

our paper effectively addresses existing gaps in the previous research. While many previous studies relied predominantly on specifically generated or manually reviewed datasets, our paper distinguishes itself by using the DevGPT dataset, which provides real-life code examples rather than purely synthetic or manually curated sources. Moreover, it uniquely uses a comprehensive set of static scanners including Semgrep, Flawfinder, Snyk, and CPPCheck, and investigates their detection capabilities across multiple languages such as C, C++, and C#. Additionally, we compare the performance of ChatGPT in detecting security issues against the results from these static scanners.

Overall, previous studies leave gaps in addressing multi-language contexts and real-world validation. Moreover, they focus only on the code itself, rather than the developer-LLM interaction which produces and analyses the code. In comparison, our work analyses code from the DevGPT dataset using a structured data pipeline including static analysis, manual validation, and ChatGPT's answers for requests to repair code. Our study provides a systematic approach using static analysis with multiple scanners, manual validation, and assessing ChatGPT's repair capabilities for C, C++, and C#, providing practical insights into ChatGPT's effectiveness in fixing security problems. Additionally, we explore ChatGPT's ability to fix confirmed vulnerabilities – an area not addressed in the work reviewed.

3 Data Exploring and Collection

The data for our analysis were taken from the publicly available DevGPT dataset, which contains information on developer interactions with ChatGPT, including prompts, answers, and generated source code [11]. This information was collected by parsing ChatGPT discussions which were shared on Hacker News [18], GitHub [13] issues, GitHub pull requests, GitHub commit messages, and GitHub source code files. GitHub and HackerNews both provide APIs which were used to retrieve the data for DevGPT.

DevGPT was selected because it is an open access dataset with a permissive license and contains ChatGPT conversations from various sources. It contains the actual output of AI tools, making it a good source for analysing how well these tools perform in a real-world programming tasks. In addition, DevGPT includes code snippets from various domains such as web development, tool development, data processing, and algorithm design.

The DevGPT dataset is organized into six sections, each containing six JSON files covering GitHub Issues, Pull Requests, Discussions, Commits, Code Files, and Hacker News threads. Of the nine available snapshots in this dataset, we chose to focus on the latest: `snapshot_20231012`.

We used Python to develop an auxiliary toolset for data analysis and file extraction. The DevGPT dataset was analysed using a custom Python script; the dataset contained the items shown in Table 2.

The most popular languages from the DevGPT dataset are presented in Table 3.

Table 1 Literature review paper comparison

Paper	Lang.	Static scanners	Real-life code example or lab generated	Detect.	Generat.
This paper	C/C++, C#	Semgrep, Flawfinder, Snyk, CPPCheck	DevGPT dataset	Yes	Yes
Bakhshandeh [3]	Python	Bandit, Semgrep, SonarQube	Real datasets: securityEval, PyT	Yes	No
Jamdade [20]	TypeScript	No	Specifically generated	Yes	Yes
Khoury [23]	C, C++, Python, HTML, and Java	No, manual review	Specifically generated	Yes	Yes
Hamer [19]	Java	CodeQL	Specifically generated, StackOverflow used to collect examples of code produced by humans	No	Yes
Kharma [21]	C, C++, Java, and Python	SonarQube	Specifically generated	No	Yes
Manik [26]	Python	Pylink, Flake8	Specifically generated	No	Yes
Almanasra [2]	Java, Python	Leetcode to test memory and speed efficiency	Specifically generated	No	Yes
Wu [36]	C, C++, Java, Go, and Python	Manual checks	Specifically generated	Yes	Yes
Cheshkov [5]	Java	Manual checks	Samples from GitHub	Yes	No
Fu [16]	C/C++	AIBugHunter, CodeBERT, GraphCodeBERT, and VulExplainer were used as static scanners	Real datasets Big-Vul and CVEFixes	Yes	Yes
Kholoosi [22]	12 different programming	No	National Vulnerability Database (NVD)	Yes	No
Sajadi [30]	Python and JavaScript	CodeQL	Stack Overflow data	Yes	No

Table 2 DevGPT dataset information

Name	Number of items
Separate ChatGPT interaction cases	4733
Conversation steps	27093
Source files	19106

Table 3 Programming languages in DevGPT dataset

Programming language	Number of files
Python	3041
Java script	2401
bash	2166
HTML	1055
C#	898
Unix shell	751
CSS	641
C++	544
Swift	487
Java	461
Go	354
C	144

We decided to focus on C#, C, and C++ as these are widely used in various domains such as systems programming, game development, and enterprise applications. By focusing on these languages, the research aims to provide insights into the specific types of security vulnerabilities that may be expected to commonly appear in in AI-generated code.

Table 4 Number of code lines for selected programming languages

Programming language	Number of files	Lines of code
C#	898	14824
C++	544	7259
C	144	1784

The number of files and number of lines of source code used for analysis of each programming language can be found in Table 4. We analysed a total of 1,586 source files, which included 898 C# files, 544 C++ files, and 144 C files. C# was the most prevalent language, accounting for 898 files and 14,824 lines of code, followed by C++ and C with decreasing numbers. Source code file statistics were generated using the `cloc` utility [6].

Another Python application was developed to iterate over the dataset and extract data in the form of files suitable for further analysis. The code samples were organized into a folder tree with the following structure:

```
Code -> JsonFileFolder -> Source_<Name> -> Sharing_<Name_Date>
-> Conversation_<N>
```

Where:

- <Name> – is specific name of the source.
- <Name_Date> – is the name and date of the conversation in sharing.
- <N> – is the number of the conversation.

Each "Sharing_<Name_Date>" folder contains information in the file `sharing_info.txt` with title, date of interaction, the URL to the ChatGPT conversation, ChatGPT model name, and number of conversations. The term “sharing” is used here as defined by DevGPT documentation, referring to a shared session of interaction between the user and ChatGPT. In other words, it represents a conversation between the user and ChatGPT on a selected topic.

The entire conversation in HTML format is stored in the `html_content.html` file within the `Sharing_<Name>` folder. Extracting data into files was necessary for two reasons: first, to store the source code as files, thereby enabling the use of static scanners; secondly, for convenience, as navigating files on disk and exploring the dataset is easier than working with JSON files.

4 Analysis and Results

The results of our experiments are presented and discussed below.

RQ1: *How secure is the code generated by ChatGPT in the DevGPT dataset?*

4.1 Static Scanners Analysis

As discussed in [14] and [24], static code scanners are essential in software development for identifying security vulnerabilities early in the development lifecycle, thereby reducing the cost and effort required to fix them. Scanners can automate the detection of common security issues, ensuring consistent application of security standards across the codebase. Numerous scanners are available on the market. Four well-known scanners were selected based on the criteria of offering a free or trial version and supporting the chosen programming languages; the selected scanners presented in Table 5.

All the files extracted in the previous stage from the DevGPT dataset were scanned using these four scanners, and a report was generated for each scanning run. Raw scanning results are presented in Table 6, where we observe that Flawfinder found 211 possible issues, approximately ten times more than the other scanners. However, most of these detections had minor “Notes/Inf” severity, and only three actual errors were detected. For all static scanners

Table 5 Static scanners selected for analysis

Scanner	Version	Programming Languages	Is free or trial available	Web site
Snyk	1.1294.0	C#, C, C++	Yes	[33]
Semgrep	1.87.0	C#, C, C++	Yes	[32]
FlawFinder	2.0.19	C, C++	Yes	[15]
CppCheck	2.14.0	C, C++	Yes	[9]

Table 6 Initial scanning results

Scanner	Number of detects	Detect severity		
		Errors	Warnings	Notes/Info
Snyk	20	0	18	2
Semgrep	10	0	10	0
Flawfinder	211	3	0	208
CppCheck	24	0	24	0

except CppCheck, reports were generated in SARIF format, a popular JSON-based format developed by OASIS [31]. Since CppCheck does not support SARIF generation, its report was generated as a plain text file. For statistics on defects per line of code, see Table 7.

Table 7 Number of detects statistics per line of code

Scanner	Languages	Files	Lines of code	Detects	Lines of files per detect	Lines of code per detect
Snyk	C#, C, C++	1586	23867	20	79.3	1193.4
Semgrep	C#, C, C++	1586	23867	10	158.6	2386.7
FlawFinder	C, C++	688	2418	211	3.3	11.5
CppCheck	C, C++	688	1730	24	28.7	72.1

4.2 Review of Initial Detections and Report Files Transformation

Since the SARIF format contains a level of detail excessive for our analysis, and has a complicated internal structure, we decided to create a simplified JSON-based format to store information from the static scanner reports. The goal of this format is to retain the essential information from the CppCheck report in a structured form, at the same time avoiding the complexity of the SARIF format. Each object in this format includes the following seven fields: line number, file name, issue severity, static scanner message, issue type, sharing name, and source code.

Furthermore, CppCheck 2.14.0 does not support generating reports in SARIF format, so this simple custom format was the only option for storing the CppCheck report information in a structured form. Therefore, a special Python script was created to convert the plain-text CppCheck report into this structured format. An example of single entry simple JSON format is shown in Listing 1.

```

1 {
2   "line": 36,
3   "file": "full_path\Code_001.c",
4   "severity": "warning",
5   "text": "Scanner message..."
6   "type": "cpp.lang.security...",
7   "sharing": "Sharing_name...",
8   "source_code": "FILE *wordlist = fopen(wordlist_path, \"r\");"
9 },

```

Listing 1 Issue object example in the simple JSON format

To ensure all reports were in a unified format, the other scanner reports were also converted to the same customised JSON format. At this stage, we had four JSON report files in the same format from four different scanners. Having the reports in a consistent format simplifies analysis and allows us to use the same approach for all selected static scanners. Each detection in the scanner report represents a potential security issue; using all report data, we initiated a review and selection process, as some detections might be false positives or perhaps very minor issues.

With the help of the another custom Python script, we analysed all the simple JSON reports from each scanner and outputted the issues, grouped by “Sharing” folder name. As noted, a “Sharing” folder represents a single discussion between ChatGPT and a user, containing multiple steps with prompts and responses.

After this stage, we performed a manual selection of a reduced set of files for more detailed analysis – the selection process is described below. Table 6 presents the number of files and the number of detects for each scanner. By processing the static scanner reports using the Python scripts, we identified 123 unique files which contained detections. Many of these files originated from the same “sharing” or interaction with ChatGPT on a particular topic.

Source code files from the same “sharing” were manually excluded unless there were significant changes during the current conversation with ChatGPT. For example, if a user asked ChatGPT to generate a function to implement bubble sort and then discussed ways to improve this function, only the final version of the bubble sort source code was included. However, if a user asked ChatGPT to generate bubble sort and then, after some interaction, also asked for a merge sort function, both the final bubble sort code and the merge sort code were included.

After completing this process, we were left with 64 files for further analysis.

4.3 Manual Analysis

The results for all 64 files were manually reviewed by the first author and the findings were discussed and validated by the second and third authors. After the selection process, we identified 32 security issues in 26 files; others were marked as false positives. For example, in several cases scanners were triggered by a simple `printf()` function call with a `const char*` argument. Another case comprised a `memcpy()` call where the size check was done on a different line, before the `memcpy()` call, and some scanners were triggered by reading and returning Boolean values from functions without validation, which is unnecessary for such values.

5 Detection and Correction

RQ2: *How effective is ChatGPT in detecting and correcting security issues when explicitly asked to do so?*

5.1 Issue Detection and Attempts to Fix Code using ChatGPT

We investigated using ChatGPT to detect and correct vulnerabilities where they exist. The experiment was conducted on the 26 files that we manually confirmed to contain 32 vulnerabilities. The public OpenAI API provides programmatic access to ChatGPT, enabling users to send prompts and receive responses. The aim of this experiment was to extract source code from original developer interactions and ask ChatGPT to identify and fix any security issues. Here is the prompt used in the experiment¹:

“I would like to ask you to behave like senior software developer with expertise in software security to answer the next question. You must find security issues in the code snippet below in this message. Give me your analysis and the way how to fix the code if possible. Try to identify CWE number or any other number for formal classifications. Please write code where detected security issue is fixed, please write all code in one fragment.”

All experiments were conducted with the (latest) GPT-4o model. Responses from ChatGPT were received as text in Markdown format and saved to disk. From each response, the source code was extracted and saved as a separate file. Additionally, the original source code from DevGPT and all conversation threads were included to help conduct the analysis.

Our results were all uploaded to a publicly available GitHub repository [29]. The 26 files that manually were confirmed to have vulnerabilities, encompassing a total of 32 confirmed security issues, were presented to ChatGPT for further analysis. The corresponding ChatGPT responses to the prompts for

¹ Unfortunately, the misspelling shown was present during the initial experiments.

the 26 files were manually reviewed to determine whether a Common Weakness Enumeration (CWE) issue was identified by ChatGPT, and whether it was fixed in the updated code.

Out of the 32 confirmed security issues in these 26 files, ChatGPT successfully detected 18 but failed to recognize 14. In terms of its ability to fix problematic code, ChatGPT successfully resolved 17 issues but failed to fix 15.

The CWE distribution of the results is presented in Table 8. A total of 43 different types of CWEs were identified, which exceeds the 32 vulnerabilities because, in some cases, the same vulnerability results in two CWEs being assigned to the same line of code. In the third column, we can see whether the issue is included in the 2024 CWE Top 25 Most Dangerous Software Weaknesses. If so, the number in that column indicates its position on the list, and if not, a dash symbol is used [1].

Table 8 CWE distribution

CWE	Description	No. of CWEs	Top 25 CWE position
CWE-20	Improper Input Validation	11	12
CWE-22	Improper Pathname Validation	1	5
CWE-61	Symlink Following	1	-
CWE-78	OS Command Injection	1	7
CWE-119	Bounds of a Memory Buffer	5	20
CWE-120	Classic Buffer Overflow	9	-
CWE-126	Buffer Over-read	2	-
CWE-129	Improper Validation of Array Index	2	-
CWE-190	Integer Overflow	1	23
CWE-203	Observable Discrepancy	1	-
CWE-319	HTTP usage	1	-
CWE-352	Cross-Site Request Forgery	1	4
CWE-362	Race Condition	2	-
CWE-385	Covert Timing Channel	1	-
CWE-401	Missing Release of Memory	1	-
CWE-665	Improper Initialization	2	-
CWE-772	Missing Release of Resource	1	-

Finding 1: Among the 43 identified Common Weakness Enumerations (CWEs), CWE-20: “Improper Input Validation” and CWE-120: “Classic Buffer Overflow” are the most frequent vulnerabilities, together accounting for 46.5% of all identified CWEs.

In Table 12 we can see how detections were distributed across the static scanners when using ChatGPT-4o. An “X” in a table cell indicates that an issue was detected, while an “O” means the issue was not detected. The column

names are “S” – Snyk, “Sg” – Semgrep, “F” – Flawfinder, “C” – CppCheck, “G-4o” – ChatGPT-4o.

Full results of the analysis of the 32 issues, with comments, are presented in Table 9 for C# issues and in Table 10 and Table 11 for C/C++ issues. In Tables 9, 10 and 11 the “Issue number” indicates the folder number “File_XX” from the Git results repository [29], followed by a slash and the line number in the source code file where vulnerability was detected. The “CWE” column contains information about the categorization of the issues, based on the CWE catalogue. The “Language” column is the programming language of the input source code file, and the “Is generated by ChatGPT” column specifies the origin of the source code for analysis. While most of these code fragments are generated by ChatGPT, some were provided by users in the prompts. Then we present a short description of each problem, and information about whether the issues detected were fixed by ChatGPT or not.

We can see from these tables that ChatGPT detected the majority of issues identified through static analysis, highlighting its ability in spotting vulnerabilities related to unvalidated inputs, unsafe buffer handling, and memory leaks. Although ChatGPT demonstrated limitations in identifying certain specific CWE categories such as CWE-362 (Race Conditions) and CWE-665 (Improper Initialization), it excelled at recognizing common and critical issues like CWE-20 (Input Validation), CWE-119/120 (Buffer Overflow), and CWE-401 (Memory Management Issues). Moreover, ChatGPT not only detected many issues effectively, but also provided practical fixes for several vulnerabilities, showcasing its potential usage as a supportive tool for software developers in addressing security flaws. However, due to its limitations and occasional inaccuracies, human supervision remains strongly recommended to ensure thorough validation and verification of security issues.

Table 9 Security issues in C# code

Issue number [29]	CWE	Lang.	Is generated by ChatGPT	Description	Was issue detected after our prompt?	Was issue fixed after our prompt?
2/14	352	C#	Yes	Antiforgery attribute is recommended to use.	No, CWE-352 was not identified.	No. There is no [ValidateAntiForgeryToken] attribute in a new code.
5/63	20	C#	User prompt	Unsanitized input used as input for file access functions	Yes, CWE-20 was spotted	Yes, Issue was identified and fixed see SanitizeFileName()
11/40	22	C#	User prompt	Generate10() function argument is not validated.	Yes, CWE-22 was spotted.	Yes, issue was fixed.

Issues per number of lines of code in different programming languages are shown in Table 13. From the data, we can observe that by far the safest language appears to be C#, with 2,964.8 source code lines per detection. Next comes C++, with 63.7 lines per detection, and finally C, which presents as the least secure language, with only 12.2 lines per static scanner detection. However, it should be noted that the “Total detects” here includes all detections, including those with “Info” or “Note” severity.

Finding 2: Our results highlight that C has the highest detect density, suggesting it may be more prone to vulnerabilities or security issues compared with C++ and C#. In contrast, C# shows significantly fewer detects relative to its large codebase, indicating stronger security potential and much fewer potential issues identified by the tools.

Looking at one example, in the folder File_16 of the results repository [29], in the file Code_001.c, we see that the source code contains only one line with no apparent security risks. The source code is shown in the Listing 2. However, ChatGPT suggests that there are at least two detected issues (CWE-532 and CWE-134) and one additional issue without an assigned CWE number.

```
1 fprintf(stderr, "Your_log_message_here\n");
```

Listing 2 Original code from DevGPT, File_16, Code_001.c

We present this example to show the difference between the code’s actual state and ChatGPT’s warnings. This code has no real security problems, but ChatGPT still reported multiple issues – even creating one without a proper CWE. From this, we learn two important things. First, AI tools can make mistakes or produce false positives, so human review is necessary. Secondly, depending only on automated results could cause the user to spend time on investigating issues that do not exist.

Finding 3: Although some code snippets may not contain detectable vulnerabilities, as shown in Listing 2, ChatGPT often offers broader suggestions to identify potential vulnerabilities and may provide solutions that sometimes lack practical value and do not make sense in the given context.

In nearly all files selected for analysis, the static scanners identified one or two issues; however, ChatGPT routinely suggested and reported more issues than those identified by the static scanners. While uncovering additional issues could be seen as positive outcome, we did not assess the accuracy of these extra findings – nor whether they represent genuine vulnerabilities or false positives – as that detailed investigation of additionally generated issues lays outside the scope of this work. As ChatGPT always presents its results with wording

Table 10 Security issues in C/C++ code. Part 1.

Issue number [29]	CWE	Lang.	Is generated by ChatGPT	Description	Was issue detected after our prompt?	Was issue fixed after our prompt?
1/1a	CWE-772	C	Yes	File descriptor leak.	Yes, CWE-772 identified	Yes, resource leak fixed.
1/1b	CWE-362	C	Yes	Flawfinder reported hardcoded file path.	No, CWE-362 was not identified by ChatGPT	CWE-362 was not fixed. Hardcoded serial port path still here
3/29	CWE-20	C++	Yes	curl_easy_setopt argument is unsanitized.	Yes, CWE-20 detected.	No, curl_easy_setopt argument still not sanitized.
4/21	CWE-119/120	C	Yes	Static size buffer is an issue.	Yes, CWE-120 identified	Yes, request block size compared with static size buffer size.
4/101	CWE-20	C	Yes	Unsanitized input for fopen().	Yes, CWE-20 spotted	No, word_list, fopen() argument still not sanitized
6/8	CWE-190	C	User prompt	Integer overflow.	Yes, CWE-190 identified	Yes, validation added for integers input.
7/39	CWE-20	C++	Yes	Unvalidated input from file.	No CWE-20 not identified.	No, issue was not fixed.
8/14	CWE-119/120	C	Yes	There is problem with fixed sized buffer usage.	Yes, CWE-120 detected	Yes, CWE-120 fixed
8/36	CWE-20	C	Yes	Unvalidated command line argument, fopen()	No, CWE-20 not detected.	Not fixed, fopen() argument still not validated
9/9	CWE-120/20	C	Yes	scanf() usage	No, issue not detected.	No, scanf() still called with the same arguments.
10/42	CWE-319	C++	Yes	Issue with using unsafe HTTP, HTTPS recommended.	HTTP usage was not identified as problem.	No, HTTP protocol still in use.
12/36	CWE-20	C	User prompt	scanf() usage.	Yes, CWE-20 related to scanf() usage was identified.	Yes, scanf() usage was fixed.
13/49	CWE-20	C	User prompt	scanf() usage.	Yes, CWE-20 scanf() issue was identified.	scanf() usage was updated.
14/14	CWE-119/120	C	Yes	Static buffer used without size check when data received.	Yes, CWE-120 identified	Yes, code updated to always null terminate buffer
14/63	CWE-129	C	Yes	Index of the array access defined by external data without validation. strcspn() call	No issue identified with strcspn() use for array index.	No strcspn() call is unmodified
18/83	CWE-120	C	User prompt	memcpy() without size check.	No, unable to spot CWE-120 and any issues with memcpy().	No, unable to fix it.

Table 11 Security issues in C/C++ code. Part 2.

Issue number [29]	CWE	Lang.	Is generated by ChatGPT	Description	Was issue detected after our prompt?	Was issue fixed after our prompt?
20/16	CWE-78	C	User prompt	Using <code>std::system()</code>	ChatGPT recognized issue with <code>system()</code> but see as CWE-77.	Tries to fix <code>system()</code> usage. However updated code still use <code>std::system()</code>
22/7	CWE-120/20	C++	User prompt	Number entered from console used as loop counters without validation.	Yes, spotted problem with entered integers value	Issue fixed by adding validation for entered numbers.
23/9	CWE-120/20	C++	User prompt	Unvalidated user input.	Yes, CWE-20 identified.	Issue with invalidated input fixed.
48/23	CWE-362	C++	Yes	Problem with hardcoded port name	Issue with hardcoded port name spotted.	Fixed, port name is not hardcoded, loaded from ini file.
49/9	CWE-61	C	Yes	Hardcoded default filename	No, CWE-61 is not detected.	Hardcoded default path is not fixed
49/27	CWE-203/385	C++	Yes	<code>std::asctime()</code> is not recommended to use	CWE-203/CWE-385, <code>asctime()</code> issue detected	Code refactored to avoid using <code>asctime()</code> .
52/14	CWE-119/120	C	Yes	Static sized buffer usage without proper checks.	Yes, CWE-120 identified	Yes, buffer is always null terminated.
52/42	CWE-129	C	Yes	Index of the array access if define by external data without validation, <code>strncpy()</code> call.	No, issue identified with <code>strncpy()</code> use for array index.	No, <code>strncpy()</code> call is unmodified
53/13	CWE-20	C++	User prompt	No validation checks for filename argument.	No, issue with file name is not spotted.	Reported issue not fixed.
54/2	CWE-126	C++	Yes	<code>strlen()</code> usage is not safe	Not, <code>strlen()</code> usage was not identified as an issue.	Reported issue not fixed.
55/22	CWE-126	C	Yes	<code>strlen()</code> usage is not safe	Did not found any problems with <code>strlen()</code>	Reported issue not fixed.
61/1	CWE-665	C++	Yes	Missing class member initialization.	No, CWE-665 was not identified.	Yes, constructor added, and private members initialized.
62/19	CWE-401/665	C++	Yes	Missing copy constructor for class can lead to memory leaks	Yes, CWE-401 detected	Yes, issue fixed, copy constructor added.

Table 12 Detection matrix

Issue[29]	CWE	Lang	S	Sg	F	C	G-4o
1/1a	772	C	X	0	X	0	X
1/1b	362	C	X	0	X	0	0
2/14	352	C#	X	0	0	0	0
3/29	20	C++	X	0	0	0	X
4/21	119/120	C	X	0	X	0	X
4/101	20	C	X	0	X	0	X
5/63	20	C#	X	0	0	0	X
6/8	190	C	X	0	0	0	X
7/39	20	C++	X	0	0	0	0
8/14	119/120	C	0	0	X	0	X
8/36	20	C	X	X	0	0	0
9/9	120/20	C	0	X	X	0	0
10/42	319	C++	0	X	0	0	0
11/40	22	C#	0	X	0	0	X
12/36	20	C	0	X	0	0	X
13/49	20	C	0	X	0	0	X
14/14	119/120	C	0	0	X	0	X
14/63	129	C	0	X	0	0	0
18/83	120	C	0	0	X	0	0
20/16	78	C	0	0	X	0	X
22/7	120/20	C++	0	0	X	0	X
23/9	120/20	C++	0	0	X	0	X
48/23	362	C++	0	0	X	0	X
49/9	61	C	0	0	X	0	0
49/27	203/385	C++	0	0	0	X	X
52/14	119/120	C	0	0	X	0	X
52/42	129	C	0	X	0	0	0
53/13	20	C++	0	0	X	0	0
54/2	126	C++	0	0	X	0	0
55/22	126	C	0	0	X	0	0
61/1	665	C++	0	0	0	X	0
62/19	401/665	C++	0	0	0	X	X

that projects a seemingly confident manner, erroneous feedback could mislead the user.

Finding 4: ChatGPT suggested additional vulnerabilities for all analysed files. It appears to strive to satisfy the user by occasionally identifying issues that may not be present, and it does so with a very confident tone, which may mislead the user.

Taking another example, in the File_10 folder of the results repository [29], in the file Code_001.cpp, ChatGPT made unnecessary changes to the code presented in Listing 3, as the original code already correctly calculates the input buffer size on line 7 of the original listing. Here, the original source code was generated by ChatGPT and was not provided as part of the user prompt.

Table 13 Language statistics

	C	C++	C#
Number of detected by Snyk	7	10	3
Number of detected by Semgrep	7	1	2
Number of detected by Flawfinder	132	79	-
Number of detected by CppCheck	0	24	-
Total files scanned	121	427	829
Total detected	146	114	5
Files with detects issue	37	84	3
Number of lines of code [6]	1784	7259	14824
Lines of code per detected issue	12.2	63.7	2964.8

```

1 struct EmailData {
2     const char* data;
3     size_t size;
4 };
5 size_t read_data_callback(void *ptr, size_t size, size_t nmemb, void *
    user_data) {
6     ...
7     size_t len = std::min(size * nmemb, email_data->size);
8     memcpy(ptr, email_data->data, len);
9     email_data->data += len;
10    email_data->size -= len;
11    ...
12 }

```

Listing 3 Original code from DevGPT, File_10, Code_001.cpp

On the listing of code ‘corrected’ by ChatGPT 4, we can see that ChatGPT added an extra member “position” to the “EmailData” structure to track input buffer position; however this is not required because the position is already tracked in the original code using the “data” pointer and “size” members of EmailData structure. See lines 8 and 9 in Listing 3.

```

1 struct EmailData {
2     const char* data;
3     size_t position;
4     size_t size;
5 };
6 size_t read_data_callback(void *ptr, size_t size, size_t nmemb, void *
    user_data) {
7     ...
8     size_t available_size = email_data->size-email_data->position;
9     size_t buffer_size = size * nmemb;
10    size_t len = std::min(available_size, buffer_size);
11    memcpy(ptr, email_data->data + email_data->position, len);
12    email_data->position += len;
13    ...
14 }

```

Listing 4 Code updated by ChatGPT, File_10, New_generated_code_01.cpp

Finding 5: We identified a case where ChatGPT unnecessarily modified the source code by adding an extra field to a structure, intending to prevent a buffer overflow, as illustrated in Listing 4. However, this modification was unnecessary because the original code already correctly tracked both the position within the input buffer and the remaining memory available in the receive buffer. As a result of ChatGPT's change, the code became less elegant, unnecessarily more complex, and consumed additional memory without providing any functional improvement.

In the folder File_20 of the results repository [29] in the file Code_001.cpp, the `system()` function usage in C was reported as CWE-78, but ChatGPT wrongly classified it as CWE-77. This is illustrated on line 14 of Listing 5.

In the folder File_20 of the results repository [29], there is a file called Code_001.cpp that uses the `system()` function in C. The Flawfinder scanner categorises this usage as CWE-78, but ChatGPT classified it as CWE-77. This is illustrated on line 16 of Listing 5. Although both CWE-77 (Improper Neutralisation of Special Elements used in a Command) and CWE-78 (OS Command Injection) both deal with command injection, they emphasise different validation aspects. Confusing them may lead to incomplete or incorrect remediation efforts, leaving the system vulnerable.

Typically, if code unsafely uses `std::system()`, it falls under CWE-78 (OS Command Injection). While CWE-77 (Improper Neutralisation of Special Elements) is a broader category relevant to command injection, CWE-78 is focused on threats arising from operating system commands. Thus, if `std::system()` is implemented in a way that could allow attackers to run malicious commands, it is more accurately classified as CWE-78.

```

1 void open_url(const std::string &url) {
2     std::string executable;
3
4     #if defined(_WIN32) || defined(_WIN64)
5         executable = "start_" + url;
6     #elif defined(__linux__)
7         executable = "xdg-open";
8     #elif defined(__APPLE__)
9         executable = "open";
10    #elif defined(__ANDROID__)
11        executable = "am_start -a android.intent.action.VIEW -d";
12    #endif
13    const std::string command = executable + " " + url + " ";
14    const int exitcode = std::system(command.c_str());
15    if (exitcode != 0) {
16        // debugmsg("Failed to open URL: %s\nAttempted command was: %s",
17        url, command);
18        // Replace debugmsg with your debugging/logging function
19    }
20 }
```

Listing 5 Original code from DevGPT, File_20, Code_001.cpp

Finding 6: We identified a case, as shown in Listing 5, where ChatGPT correctly recognized the root cause of the problem but assigned a broader issue number CWE-77. This broader classification is not accurate, as the more specific CWE-78 better captures the nature of the issue.

Taking another example, in the folder File_55 of the results repository [29], in file Code_001.c:22, `strlen()` was called with a `const char*` argument. The static scanner Flawfinder reported this call as an issue with severity “Note”. This example is presented in Listing 6. However, ChatGPT did not report `strlen()` call as a security issue, likely recognizing that the `strlen()` argument is a hard-coded string. In Listing 6, the argument for the `strlen()` call is `interface`, which is a properly null terminated, hard-coded C string. This example illustrates that sometimes ChatGPT can perform more efficiently than static scanners.

```

1  ...
2  // Bind the socket to a specific network interface
3  char *interface = "eth0"; // or "eth1" or any other interface
4  if (setsockopt(sockfd, SOL_SOCKET, SO_BINDTODEVICE, interface, strlen(
5      interface)) < 0) {
6      perror("SO_BINDTODEVICE");
7      exit(EXIT_FAILURE);
8  }
9  ...

```

Listing 6 Original code from DevGPT, File_55, Code_001.c

Finding 7: We identified one case, shown in Listing 6, where ChatGPT outperformed static scanners by providing a more accurate analysis.

In another case, in the folder File_61 of the results repository [29], the lack of a constructor and the uninitialized private member 'balance' was implicitly corrected by ChatGPT without reporting CWE numbers when we asked it to spot and fix security issues using our prompt. This can be seen in the Listing 7.

```

1  ...
2  class BankAccount {
3  ...
4  public:
5      BankAccount() : balance(0.0) {}
6  ...
7  }
8  ...

```

Listing 7 Code updated by ChatGPT, File_61, New_generated_code_01.cpp

Finding 8: We identified one case where ChatGPT fixed a potential security vulnerability in the updated source code, as shown in Listing 7, without providing any explanation in the textual response.

In the folder File_61 of the results repository [29], the file Code_002.cpp was reported by CppCheck to contain the issue CWE-665 (Improper Initialization), which was not identified by ChatGPT but was recognized as an actual problem. Lack of initialisation of 'balance' variable can be seen in Listing 8.

```
1  ...
2  class BankAccount {
3  private:
4      double balance;
5
6      void deductFees() {
7          // Some code to deduct fees from the balance
8      }
9
10 public:
11     void deposit(double amount) {
12         balance += amount;
13     }
14
15     double getBalance() {
16         return balance;
17     }
18 };
19 ...
```

Listing 8 Original code from DevGPT, File_61, Code_002.cpp

Among the 32 security issues identified in 26 selected files, ChatGPT-4o managed to successfully detect 18 cases, and failed to recognize 14 issues. In 17 cases of 32, ChatGPT-4o was able to fix the vulnerability, and in 15 cases failed to do so. We note also that ChatGPT can demonstrate high apparent confidence when providing incorrect results; our observations are in accordance with the conclusion made by Borji in [4]: developers should not trust ChatGPT's results too much, even if the answers appear confident and correct. Research in psychology has shown that apparent confidence is a major factor in advice being accepted [35], so ChatGPT's confident presentation of wrong information may mislead developers, especially those with less experience on which to evaluate the information given.

Finding 9: With an effective success rate of approximately 50%, ChatGPT-4 is not currently reliable enough for independent security analysis. ChatGPT cannot currently be used unsupervised – manual supervision and the use of static scanners are strongly advised.

Given these limitations, manual supervision is strongly recommended when relying on ChatGPT-4 to ensure that security flaws are accurately detected

and addressed. These results are also closely aligned with those from [5] and [16], highlighting ChatGPT’s inability to function effectively as a static security scanner.

RQ3: *Do developers provide more vulnerable code, or does ChatGPT generate more insecure code during interactions?*

As previously discussed, the DevGPT dataset captures real-life interactions between users and ChatGPT. In some cases, users ask ChatGPT questions and include their code in the prompts. We have collected this information, and the origin of the analysed code is indicated in the “Is generated by ChatGPT” (Yes/No) column in the Tables 9, 10 and 11. Out of 26 files, issues were found in 10 where the input source code came from user prompts. These 10 files account for 10 of the 32 security issues identified.

Finding 10: Out of 26 files, issues were found in 10, where the input source code originated from user prompts. These 10 files contain 10 of the 32 security issues identified.

We analysed ChatGPT’s detection and fixing rates for files where the original code was provided in user prompts and files containing code originally generated by ChatGPT. For the 10 issues in code from user prompts, 8 issues were detected, and 7 were fixed. For the 22 issues in code originally generated by ChatGPT, 10 issues were detected, and 10 were fixed.

The detection and fixing rates differ significantly between code from user prompts and code originally generated by ChatGPT. For code from user prompts, the detection rate is 80%, and the fixing rate is 70%. In contrast, for code originally generated by ChatGPT, both the detection and fixing rates are 45%. This suggests that ChatGPT is significantly poorer at detecting and fixing issues in code generated by itself.

Finding 11: For source code derived from user prompts, the detection and fixing rates are 80% and 70%, respectively, which are higher than those for code originally generated by ChatGPT, where both detection and fixing rates are 45%.

This section evaluates ChatGPT-4o’s ability to detect and fix security issues in source code. Overall, ChatGPT showed mixed performance, with both successes and numerous limitations. Eleven key findings were identified, highlighting strengths, weaknesses, and the need for human oversight when using AI for security-related tasks.

6 Threats to Validity

In this study, we acknowledge several threats to validity that may impact the interpretation and generalisation of our findings.

6.1 Internal Validity

One threat to internal validity arises from the manual steps involved in selecting the 26 files containing 32 security issues. The selection process required subjective judgment, which may introduce bias and affect the consistency of the dataset. Although we involved all three authors to reduce bias, the potential for unintentional bias remains. Automating the selection process or involving more reviewers could mitigate this issue in future work.

Additionally, during our experimentation with the OpenAI API, a typographical error occurred in the word “security” within the prompt. While we believe that ChatGPT correctly interpreted the intended meaning despite the error, (since ChatGPT works with tokens not words), this oversight may have somewhat influenced the reliability of the results. Providing error-free prompts may lead to better integrity of the experiment and avoid unintended effects on the model’s responses.

6.2 External Validity

Regarding external validity, our analysis was focused on DevGPT and limited to 32 security issues across 26 files, which may not be sufficient to generalize the findings to all code generated by ChatGPT or similar models. The sample size is relatively small compared to the vast range of possible code snippets and security vulnerabilities. Consequently, caution should be exercised when extrapolating these results to broader contexts. Expanding the dataset and including a more diverse set of programming languages and code examples would enhance the generalisability of the study.

6.3 Construct Validity

A threat to construct validity is the inherent non-determinism of ChatGPT’s outputs. The model can produce different responses to the same prompt on different occasions, introducing variability in the results. This non-reproducibility may affect the consistency of the findings. To address this issue, it would be beneficial to repeat the experiments multiple times to assess the stability of ChatGPT’s ability to identify and fix security issues across numerous trials. Statistical analysis of the variations could provide insights into the reliability of the model’s performance.

7 Conclusion

In this study, we analysed a total of 1,586 source files extracted from the DevGPT dataset using static source code scanners. The analysis included three programming languages: C# (898 files), C++ (544 files), and C (144 files). Our primary objective was to assess ChatGPT-4’s ability to detect and resolve security vulnerabilities within these files. Among the 32 security issues identified in 26 selected files, ChatGPT-4 successfully detected 18 issues but failed to recognize 14 vulnerabilities. Additionally, ChatGPT-4 was able to fix 17 out of the 32 issues, while in 15 cases it failed to provide an effective solution.

These results indicate a mixed performance from ChatGPT-4. While it demonstrates some capacity to assist in security-related tasks, it also has a tendency to overlook potential vulnerabilities, highlighting the significant need for manual oversight. With an effective success rate of approximately 56% in detecting vulnerabilities and 53% in fixing them, ChatGPT-4 is not currently reliable enough for independent security analysis of source code, particularly in the context of complex or nuanced vulnerabilities.

Regularly ChatGPT suggests a CWE that apparently does not exist in the source code, but it does so with strong confidence. This can confuse users, leading them to believe a vulnerability is confirmed. Therefore, it is crucial to verify these findings through further analysis and expert review.

The findings underscore that while ChatGPT-4 can be a valuable auxiliary tool in identifying straightforward security issues, it falls short in handling more complicated code analysis without human intervention. Developers should exercise caution and not rely solely on ChatGPT-4 for security-critical code generation and analysis.

8 Future Work

Our study opens several avenues for further research:

- **Re-evaluating Fixed Code with Static Scanners:** We plan to re-run the static scanners on the source code files modified by ChatGPT-4, where security issues were purportedly fixed, to determine if the scanners still identify vulnerabilities in the revised code.
- **Enhancing Prompts with Diagnostic Information:** For cases where ChatGPT-4 was unable to detect and fix a security issue, we propose including additional information from static scanners, such as line numbers and detailed scanner messages, to provide “hints” in the prompts. This approach aims to assess whether supplying more context enables ChatGPT-4 to identify and resolve the issues upon re-evaluation.
- **Comprehensive Analysis of Reported Issues:** We intend to perform a detailed analysis of each security problem reported by ChatGPT-4 that was not identified by the static scanners. Since ChatGPT-4 often reports more security issues than the scanners, understanding these discrepancies

could offer insights into the strengths and limitations of AI-driven code analysis compared to traditional static analysis tools.

- **Exploring Advanced Models and Training:** Investigating whether newer versions of ChatGPT, DeepSeek [12] or other advanced LLMs exhibit improved capabilities in security detection and correction could provide valuable information on the evolution of AI tools in software security.
- **Human-AI Collaborative Approaches:** Developing and evaluating collaborative frameworks where human expertise and AI tools like ChatGPT-4 work synergistically may enhance the overall effectiveness of security analysis in code generation. In our opinion, Human-AI collaboration is the best way to use AI tools, and this line of futures research can be considered recommended.
- **Including additional programming languages and using LLM models in the analysis:** Adding other popular programming languages such as Java, Python, JavaScript and using other LLMs like LLaMa and Gemini instead of ChatGPT could provide further insights.

By addressing these areas, future research can contribute to improving the reliability of AI-assisted code generation tools and fostering safer software development practices.

9 Declarations

9.1 Funding

The research received no external funding.

9.2 Ethical approval

Not applicable

9.3 Informed consent

Not applicable

9.4 Author Contributions

Based on the following abbreviations for the authors:

Vladislav Belozarov - V, Peter J Barclay - P, and Ashkan Sami - A.

The contributions of authors based on the CRediT taxonomy are:

- Conceptualisation (A)
- Data Curation (V)
- Formal Analysis (V, P, A)
- Funding Acquisition (N/A)
- Investigation (V)
- Methodology (A, P)
- Project Administration (A, P)
- Resources (A, P, V)
- Software (V)
- Supervision (P, A)
- Validation (A, V, P)
- Visualisation (V, P)
- Writing – Original Draft Preparation (A, V, P)
- Writing – Review & Editing (V, A, P)

9.5 Data Availability Statement

The dataset used in this work is the DevGPT dataset, available at the following link: [11].

The results are available in the GitHub repository: [29].

All software tools used in this study, along with other supporting materials, can be made available upon request.

9.6 Conflict of Interest

The authors have no competing interests to declare.

9.7 Clinical Trial Number in the manuscript.

Clinical trial number: not applicable.

References

1. 2024 CWE Top 25 Most Dangerous Software Weaknesses, https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html, retrieved 29.03.2025
2. Almanasra, S., & Suwais, K. (2025). Analysis of ChatGPT-Generated Codes Across Multiple Programming Languages. IEEE Access. <https://doi.org/10.1109/ACCESS.2025.3538050>
3. Bakhshandeh, A., Keramatfar, A., Norouzi, A., & Chekidehkhoun, M. M. (2023). "Using ChatGPT as a static application security testing tool" <http://arxiv.org/abs/2308.14434>
4. Borji, A. (2023). "A Categorical Archive of ChatGPT Failures." <http://arxiv.org/abs/2302.03494>
5. Cheshkov, A., Zadorozhny, P., & Levichev, R. (2023). "Evaluation of ChatGPT Model for Vulnerability Detection." <http://arxiv.org/abs/2304.07232>
6. Cloc tool, <https://github.com/AIDanial/cloc>, retrieved 17.11.2024
7. CodeBert web site, <https://github.com/microsoft/CodeBERT>, retrieved 25.03.2025
8. CodeForces web site, <https://codeforces.com/>, retrieved 09.04.2025
9. CppCheck scanner, <https://github.com/danmar/cppcheck>, retrieved 10.11.2024
10. CWE database, <https://cwe.mitre.org/>, retrieved 20.11.2024
11. DevGPT, https://github.com/NAIST-SE/DevGPT/tree/main/snapshot_20231012, retrieved 31.10.2024
12. DeepSeek web site, <https://www.deepseek.com/>, retrieved 27.03.2025
13. GitHub, <https://github.com>, retrieved 18.11.2024
14. Goseva-Popstojanova, K., & Perhinschi, A. (2015). "On the capability of static code analysis to detect security vulnerabilities". Information and Software Technology, 68, 18–33. <https://doi.org/10.1016/j.infsof.2015.08.002>
15. Flawfinder scanner, <https://dwheeler.com/flawfinder>, retrieved 10.11.2024
16. Fu, M., Tantithamthavorn, C. K., Nguyen, V., & Le, T. (2023). "ChatGPT for Vulnerability Detection, Classification, and Repair: How Far Are We?" Proceedings - Asia-Pacific Software Engineering Conference, APSEC, 632–636. <https://doi.org/10.1109/APSEC60848.2023.00085>
17. Fu, Y., Liang, P., Tahir, A., Li, Z., Shahin, M., Yu, J., & Chen, J. (2023). Security Weaknesses of Copilot Generated Code in GitHub.
18. Hacker news, <https://news.ycombinator.com/news>, retrieved 06.11.2024
19. Hamer, S., d'Amorim, M., & Williams, L. (2024). "Just another copy and paste? Comparing the security vulnerabilities of ChatGPT generated code and StackOverflow answers."
20. Jamdade, M., & Liu, Y. (2024). "A pilot study on secure code generation with ChatGPT for Web applications." Proceedings of the 2024 ACM Southeast Conference, ACMSE 2024, 229–234. <https://doi.org/10.1145/3603287.3651194>
21. Kharma, M., Choi, S., AlKhanafseh, M., & Mohaisen, D. (2025). Security and Quality in LLM-Generated Code: A Multi-Language, Multi-Model Analysis. <http://arxiv.org/abs/2502.01853>
22. Kholoosi, M. M., Babar, M. A., & Croft, R. (2024). A Qualitative Study on Using ChatGPT for Software Security: Perception vs. Practicality. 2024 IEEE 6th International Conference on Trust, Privacy and Security in Intelligent Systems, and Applications (TPS-ISA), 107–117. <https://doi.org/10.1109/TPS-ISA62245.2024.00022>
23. Khoury, R., Avila, A. R., Brunelle, J., & Camara, B. M. (2023). "How secure is code generated by ChatGPT?" Conference Proceedings - IEEE International Conference on Systems, Man and Cybernetics, 2445–2451. <https://doi.org/10.1109/SMC53992.2023.10394237>
24. Kulenovic, M., & Donko, D. (2014). "A survey of static code analysis methods for security vulnerabilities detection."
25. Leetcode web site, <https://leetcode.com/>, retrieved 31.03.2023
26. Motaleb, M., & Manik, H. (2025). ChatGPT vs. DeepSeek: A Comparative Study on AI-Based Code Generation.
27. Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2021). Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. <http://arxiv.org/abs/2108.09293>

28. Perry, N., Srivastava, M., Kumar, D., & Boneh, D. (2023). Do Users Write More Insecure Code with AI Assistants? CCS 2023 - Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, 2785–2799. <https://doi.org/10.1145/3576915.3623157>
29. Results GitHub repository, https://github.com/vlad2010/MSc_DataEngineering_Results/tree/main/OpenAI_API_Issues64_Experiment/2024_10_22, retrieved 31.10.2024
30. Sajadi, A., Le, B., Nguyen, A., Damevski, K., & Chatterjee, P. (2025). Do LLMs Consider Security? An Empirical Study on Responses to Programming Questions. <http://arxiv.org/abs/2502.14202>
31. Sarif format, <https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html>, retrieved 31.10.2024
32. Semgrep scanner, <https://semgrep.dev>, retrieved 10.11.2024
33. Snyk scanner, <https://snyk.io>, retrieved 10.11.2024
34. StackOverflow web site, <https://stackoverflow.com/>, retrieved 18.11.2024
35. Van Swol, L. M., & Snizek, J. A. (2005). Factors affecting the acceptance of expert advice. *British journal of social psychology*, 44(3), 443-461.
36. Wu, F., Zhang, Q., Bajaj, A. P., Bao, T., Zhang, N., Wang, R. “Fish,” & Xiao, C. (2023). “Exploring the limits of ChatGPT in software security applications.” <http://arxiv.org/abs/2312.05275>

