Security Weaknesses of Copilot-Generated Code in GitHub Projects: An Empirical Study

YUJIA FU, School of Computer Science, Wuhan University, China
PENG LIANG*, School of Computer Science, Wuhan University, China
AMJED TAHIR, Massey University, New Zealand
ZENGYANG LI, School of Computer Science, Central China Normal University, China
MOJTABA SHAHIN, RMIT University, Australia
JIAXIN YU, School of Computer Science, Wuhan University, China
JINFU CHEN, School of Computer Science, Wuhan University, China

Modern code generation tools utilizing AI models like Large Language Models (LLMs) have gained increased popularity due to their ability to produce functional code. However, their usage presents security challenges, often resulting in insecure code merging into the code base. Thus, evaluating the quality of generated code, especially its security, is crucial. While prior research explored various aspects of code generation, the focus on security has been limited, mostly examining code produced in controlled environments rather than open source development scenarios. To address this gap, we conducted an empirical study, analyzing code snippets generated by GitHub Copilot and two other AI code generation tools (i.e., CodeWhisperer and Codeium) from GitHub projects. Our analysis identified 733 snippets, revealing a high likelihood of security weaknesses, with 29.5% of Python and 24.2% of JavaScript snippets affected. These issues span 43 Common Weakness Enumeration (CWE) categories, including significant ones like *CWE-330: Use of Insufficiently Random Values*, *CWE-94: Improper Control of Generation of Code*, and *CWE-79: Cross-site Scripting*. Notably, eight of those CWEs are among the 2023 CWE Top-25, highlighting their severity. We further examined using *Copilot Chat* to fix security issues in Copilot-generated code by providing *Copilot Chat* with warning messages from the static analysis tools, and up to 55.5% of the security issues can be fixed. We finally provide the suggestions for mitigating security issues in generated code.

CCS Concepts: • **Software and its engineering** → **Software development techniques**; • **Security and privacy** → **Software security engineering**.

Additional Key Words and Phrases: Code Generation, Security Weakness, CWE, GitHub Copilot, GitHub Project

1 INTRODUCTION

Code generation tools aim to automatically generate functional code based on prompts, which can include text descriptions (comments), code (such as function signatures, expressions, variable names, etc.), or a combination of text and code [65]. After writing an initial code or comment, developers can rely on code generation tools to

*Corresponding author

Authors' addresses: Yujia Fu, School of Computer Science, Wuhan University, China, yujia_fu@whu.edu.cn; Peng Liang, School of Computer Science, Wuhan University, China, liangp@whu.edu.cn; Amjed Tahir, Massey University, New Zealand, a.tahir@massey.ac.nz; Zengyang Li, School of Computer Science, Central China Normal University, China, zengyangli@ccnu.edu.cn; Mojtaba Shahin, RMIT University, Australia, mojtaba.shahin@rmit.edu.au; Jiaxin Yu, School of Computer Science, Wuhan University, China, jiaxinyu@whu.edu.cn; Jinfu Chen, School of Computer Science, Wuhan University, China, jinfuchen@whu.edu.cn.

ACM 1557-7392/2025/2-ART
https://doi.org/10.1145/3716848

complete the remaining code. This approach can save development time and accelerate the software development process. Automated code generation tools have always been a topic of active research [41, 68]. Some of the earliest work can be traced back to the 1960s, when Waldinger and Lee proposed a program synthesizer called PROW, which automatically generated LISP programs based on specifications provided by users in the form of a predicate calculus [78]. As computer languages continued to evolve, more and more programming languages began to support meta-programming, making automated code generation technology more efficient and flexible. In recent years, the rapid development of artificial intelligence technologies, particularly deep learning models, has accelerated the development of code generation technologies.

Recent advancements in code generation came with the emergence of Large Language Models (LLMs). LLMs are deep learning models trained on a large code/text corpus with powerful language understanding capabilities that can be used for tasks such as natural language generation, text classification, and question-answer systems [6]. Compared to previous deep learning methods, the latest developments in LLMs, such as Generative Pre-trained Transformer (GPT) models, have opened up new opportunities to address the limitations of existing automated code generation technology [47]. Currently, LLM-based code generation tools have also been widely applied, such as Codex by OpenAI [52], AlphaCode by DeepMind [40], and CodeWhisperer by Amazon [3].

These models are trained on billions of public open-source lines of code, which include public code with unsafe coding patterns [31]. Therefore, code generation tools based on such models can pose code quality issues [42], and the AI generated code may also suffer from security weaknesses [66]. For example, GitHub Copilot may produce some insecure code, as its underlying Codex model is pre-trained on untrusted data from GitHub [4], which is known to contain buggy programs [61]. According to the developer security company Snyk, GitHub Copilot may also replicate existing security weaknesses in code to suggest insecure code when the user's existing codebase contains security weaknesses [38].

In addition, the code with vulnerabilities generated by these code-generation tools may continue to be used to train the model, thus further generating code with vulnerabilities, leading to a vicious cycle. Previous research has studied code generation tools, with more focus on the correctness of the results [12, 39, 58, 80], and relatively less attention has been paid to security aspects [55, 56, 67]. To the best of our knowledge, potential security weaknesses in practical scenarios have not been fully considered and addressed in previous work, and GitHub Copilot clarifies that "*the users of Copilot are responsible for ensuring the security and quality of their code*" [24]. Code generation algorithms of Copilot are incentivized to suggest code to be accepted rather than other code qualities, e.g., easy to read and understand, which has an adverse impact on the code quality generated by Copilot [30]. GitHub also provides tools such as CodeQL to help developers scan for security weaknesses in their code.

To this end, we conducted an empirical study on the security weaknesses of the generated code by GitHub Copilot and other AI code generation tools (CodeWhisperer and Codeium), which is available on GitHub. We chose Copilot as our main research subject because it is a commercial instance of AI programming and has gained much attention and popularity among developers since its launch in 2021. The security weaknesses of code generated by Copilot have also gained attention in the research and practice community. Furthermore, thousands of developers in the GitHub community have shared their experiences of using Copilot in open source projects [12]. We also include the code generated by other two popular AI code generation tools, CodeWhisperer and Codeium, for the purpose of increasing the generalizability of this research. We then collected the code generated by Copilot and other generation tools that has been used in projects on GitHub and analyzed the security of the generated code through the lens of open source development environment. Then, we used static analysis tools to perform security analysis on the collected code snippets and classified the security weaknesses in the code snippets using the Common Weakness Enumeration (CWE).

Our findings show that: (1) around 30% of code snippets have security weaknesses; (2) the security weaknesses are diverse and related to 43 different CWEs, in which*CWE-330: Use of Insufficiently Random Values*, *CWE-94: Improper Control of Generation of Code ('Code Injection')* and *CWE-79: Cross-site Scripting* and are the most

frequently occurred; and (3) among the 38 CWEs identified, eight CWEs belong to the currently recognized 2023 CWE Top-25. Six CWEs belong to Stubborn Weaknesses in the CWE Top 25. (4) It is possible to use *Copilot Chat* to fix security issues in Copilot-generated code, but the success of *Copilot Chat*'s fixes varies between CWEs. (5) Providing *Copilot Chat* with a warning message from the static analysis tool resulted in a better fix. Using the fix command can fix 19.3% of security issues, while using the enhanced prompt raises the rate to 55.5%.

Contributions: (1) We curated a dataset of code snippets generated by Copilot that has been used in projects on GitHub (a curated data [21] is made available online) and conducted security checks on them, which can to some extent reflect the frequency of security weaknesses encountered by developers when using Copilot to generate code in actual coding. In addition to this, we also categorized the application areas of these code snippets. (2) We extensively checked all possible CWEs in the code snippets and analyzed them. This can help developers understand the common CWEs caused by using Copilot to generate code in actual coding and how to safely accept the code suggestions provided by Copilot. (3) We utilized *Copilot Chat* to fix code snippets generated by Copilot, demonstrating Copilot's ability to fix security issues to a certain extent.

The rest of this paper is structured as follows: Section 2 presents the background and related work. Section 3 presents the research questions and the research design of this study. Section 4 presents the results of our study, which are further discussed in Section 5. The potential threats to validity are clarified in Section 6. Section 7 concludes this work with future work directions.

2 BACKGROUND AND RELATED WORK

In this section, we present the related work in four aspects, i.e., AI code generation tools (Section 2.1), security of code generation techniques and LLMs (Section 2.2), how AI code generation tools work (Section 2.3), and static analysis tools for analyzing security weaknesses (Section 2.4).

2.1 AI Code Generation Tools

With the rise of code generation tools integrated with IDEs, many studies have evaluated these systems based on transformer models to better understand their effectiveness in open source development scenarios. Previous research mainly focused on whether the code generated by these tools can meet users' functional requirements. Yetistiren *et al.* [82] evaluated the effectiveness, correctness, and efficiency of the code generated by GitHub Copilot, and the results showed that GitHub Copilot could generate valid code with a success rate of 91.5%, making it a promising tool. Sobania *et al.* [71] evaluated the correctness of the code generated by GitHub Copilot and compared the tool with an automatic program generator with a Genetic Programming (GP) architecture. They concluded there was no significant difference between the two methods on benchmark problems. Nguyen and Nadi [50] conducted an empirical study using 33 LeetCode problems and created queries for Copilot in four different programming languages. They evaluated the correctness and comprehensibility of the code suggested by Copilot by running tests provided by LeetCode. They found that Copilot's suggestions have lower complexity. Burak *et al.* [81] evaluated the code quality of AI code generation tools. They compared the improvements between the latest and older versions of Copilot and CodeWhisperer and found that the quality of generated code had improved.

In recent years, researchers have also started to focus on the experience of developers when using AI code generation tools and how the tools can improve productivity by observing their behavior. Vaithilingam *et al.* [77] studied how developer use and perceive Copilot and found that although Copilot may not necessarily improve task completion time or success rate, it often provides a useful starting point. They also noted that the participants had difficulties understanding, editing, and debugging the code snippets generated by Copilot. Barke *et al.* [2] presented the first theoretical analysis of how developer interact with Copilot based on the observations of 20 participants. Sila *et al.* [39] conducted an empirical study on AlphaCode, identifying similarities and performance

differences between code generated by code generation tools and code written by human developers. They argued that software developers should check the generated code for potentially problematic code that could introduce performance weaknesses.

The studies presented above have conducted a relatively extensive evaluation of code-generation tools regarding correctness, effectiveness, and robustness. However, its security still has room for improvement, as detailed below.

2.2 Security of Code Generation Techniques and LLMs

Code security is an issue that cannot be ignored in the software development process. Recent work has primarily focused on evaluating the security of the code generation tools and the security of the LLMs based on these tools.

Pearce *et al.* [55] first evaluated the security of GitHub Copilot in generating programs by identifying known weaknesses in the suggested code. The authors prompted Copilot to generate code for 89 cybersecurity scenarios and evaluated the weaknesses in the generated code. They found that 40% of the suggestions in the relevant context contained security-related bugs (i.e., CWE classification from MITRE [46]). Siddiq *et al.* [67] conducted a large-scale empirical study on code smells in the training set of a transformer-based Python code generation model and investigated the impact of these harmful patterns on the generated code. They observed that Copilot introduces 18 code smells, including non-standard coding patterns and two security smells (i.e., code patterns that often lead to security defects). Khoury *et al.* [37] studied the security of the source code generated by the ChatGPT chatbot based on LLMs and found that ChatGPT was aware of potential weaknesses but still frequently generated some non-robust code. Elgedawy *et al.* [18] compared the capabilities of four code generation models using nine code generation tasks. They collected 61 code outputs and studied their security. The results revealed that the code generated by different LLMs exhibited disparate levels of security robustness.

Several researchers also compared the situation in which code generation tools produce insecure code with that of human developers. Sandoval *et al.* [64] conducted a security-driven user study, and their results showed that the rate at which AI user programming produced critical security errors was no more than 10% of the control group, indicating that the use of LLMs does not introduce new security risks. Asare *et al.* [1] conducted a comparative empirical analysis of these tools and language models from a security perspective and investigated whether Copilot is as bad as humans in generating insecure code. They found that while Copilot performs differently across vulnerability types, it is not as bad as human developers when introducing code vulnerabilities. In addition, researchers have also constructed datasets to test the security of these tools. Tony *et al.* [76] proposed LLMSecEval, a dataset containing 150 natural language prompts that can be used to evaluate the security performance of LLMs. Siddiq *et al.* [68] provided a dataset, SecurityEval, for testing whether a code generation model has weaknesses. The dataset contains 130 Python code samples. Natella *et al.* [49] provide a dataset and conduct an experimental evaluation of the security of three popular LLMs (GitHub Copilot, Amazon CodeWhisperer, and CodeBERT).

Different from prior work, we studied the security weaknesses exhibited by code generation tools in open source development environment (i.e., GitHub). We collected code snippets from GitHub generated by developers using Copilot in daily production as a source of research data, whereas in the study by Pearce *et al.* [55], the research data came from code generated by the authors using Copilot based on natural language prompts related to high-risk network security weaknesses. Additionally, Pearce *et al.* configured CodeQL only to examine CWEs targeted by security weaknesses associated with the provided scenarios. In contrast, we used various static analysis tools to examine all types of CWEs and analyze them extensively. Our research results may help developers understand what common CWEs are prone to result from using Copilot to generate code in real coding.

2.3 How AI Code Generation Tools Work: Copilot as an Example

Existing automated code generation tools, such as Copilot, CodeWhisperer, and Codeium, are often integrated into IDEs. These tools provide developer with real-time code suggestions based on given prompts. Those suggestions can range from code snippets to complete functions. We use GitHub Copilot as an example to demonstrate how these tools are used. When developers write code in their IDE, Copilot continuously scans the program and performs the following actions [10]: (1) complete existing code based on function or variable names; (2) generate code suggestions from inline natural language comments; (3) automatically fill in repetitive code based on the context of existing code; and (4) generate test cases. Figure 1 shows an example of Copilot usage in action. In Figure 1a, Copilot automatically suggests the entire function body (shown in gray text) when the function name is provided. A user can then accept the suggestion if desired. In Figure 1b, a function description is given in the comments (Lines 1-2), and Copilot provides code suggestions starting from Line 3. After accepting a suggestion, Copilot will continue to suggest the next line of code. A user can also view alternative suggestions - Copilot will provide 10 different suggestions at a time, as shown in Figure 2.

2.4 Security Static Analysis

Detecting software vulnerabilities is critical to improve security and ensure quality [34]. Vulnerabilities can be detected through static, dynamic, or hybrid analysis. Static analysis offers greater coverage and allows users to analyze programs without the need to execute them, but it can be unsound and imprecise [72]. On the other hand, dynamic analysis is more sound and precise (as it captures actual program behaviour at runtime) but can be incomplete and lacks coverage [19]. Due to their ease of use and low cost compared to dynamic analysis, static analysis tools are widely used for security analysis [33, 51].

There are several security static analysis tools available for different programming languages and application types. OWASP and Snyk [54, 70] provide a list of commonly used static analysis tools for security analysis. This includes tools like CodeQL, a general-purpose automatic analysis tool; FindBugs/SpotBugs for Java; ESLint for JavaScript programs; Bandit for Python; and GoSec for Go programs. Those tools have been widely used in previous security analysis research [55, 67, 75].

Kaur *et al.* [36] compared static analysis tools for vulnerability detection in analyzing C/C++ and Java source code. Tomasdottir *et al.* [75] conducted an empirical study on ESLint, the most commonly used JavaScript static analysis tool among developers. Pearce *et al.* [55] used CodeQL to scan security weaknesses in the generated Python and C++ code. Siddiq *et al.* [68] used Bandit to check Python code generated using a test dataset. Lisa *et al.* [14] reported on users' goals, motivations, and strategies when using static analysis tools.

These static analysis tools support different analysis algorithms and techniques. By using multiple tools for analysis, potential weaknesses in the code can be discovered from different perspectives and levels, avoiding omissions and improving the accuracy of the analysis. Our study first used CodeQL to scan the collected code snippets. CodeQL is an open-source tool that supports multiple languages, including Java, JavaScript, C++, C#, and Python. It can find weaknesses in a codebase based on known weaknesses/rules. In addition, to obtain more comprehensive analysis results, we supplemented the scan of code in different languages with static analysis tools (i.e., Cppcheck and Bandi) tailored to specific languages.

3 RESEARCH DESIGN

In this section, we describe our research design in detail. In Section 3.1, we first define our Research Questions (RQs), followed by the process of collecting and filtering the code snippets generated by code generation tools such as Copilot in Section 3.2. We then explain the security analysis performed on the identified snippets and the process of filtering the raw results generated by static analysis tools in Section 3.3. Finally, in Section 3.4, we introduce the process of using *Copilot Chat* to fix the security weaknesses identified by the static analysis tools.

```
def CalculateDaysBetweenDates(date1, date2):
    # date1 and date2 are in the format of 'YYYY-MM-DD'
    year1, month1, day1 = map(int, date1.split('-'))
    year2, month2, day2 = map(int, date2.split('-'))
    days = 0
    while year1 < year2 or month1 < month2 or day1 < day2:
        days += 1
        day1 += 1
        if day1 > 30:
            day1 = 1
            month1 += 1
        if month1 > 12:
            month1 = 1
            year1 += 1
    return days
```

(a) Method 1: Copilot completes a function using the function name

```
# read an image using opencv
# and display it using matplotlib
# usage: python3 read_image.py <image_file>
import cv2
import sys
import matplotlib.pyplot as plt

def main():
    if len(sys.argv) < 2:
        print("Usage: python3 read_image.py <image_file>")
        sys.exit(1)

    image_file = sys.argv[1]
    image = cv2.imread(image_file)

    if image is None:
        print(f"Error: could not read image file {image_file}")
        sys.exit(1)

    # convert the image from BGR to RGB
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    plt.imshow(image)
    plt.axis('off')
    plt.show()

if __name__ == "__main__":
    main()
```

(b) Method 2: Copilot implements the function in the comment

Fig. 1. Two methods of using Copilot in action

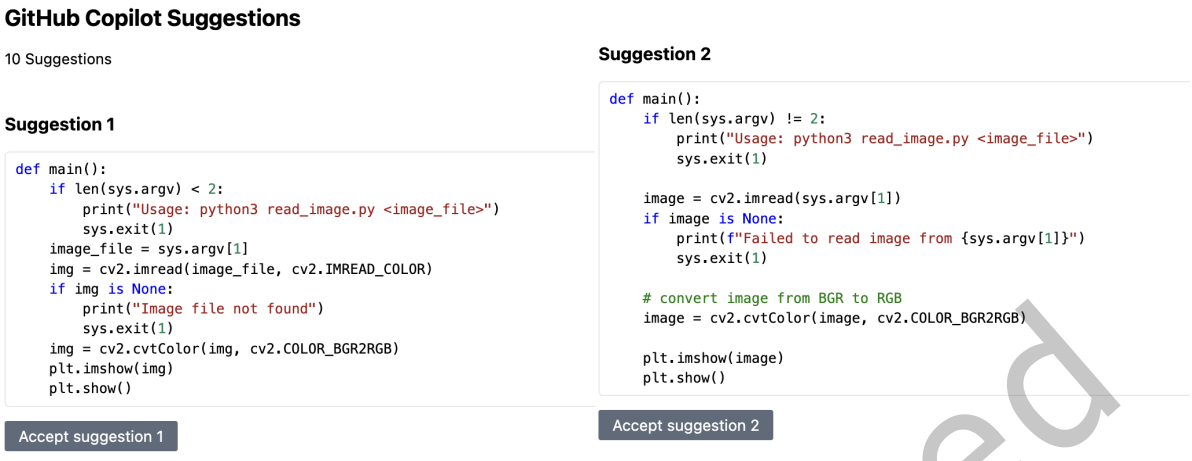


Fig. 2. More suggestions provided by Copilot

3.1 Research Goal and Questions

This study aims to understand the potential security weaknesses in AI-generated code produced in open source GitHub projects. We first collect code snippets generated by code generation tools such as Copilot from GitHub projects, which is our data source. It should be noted that it is not possible to access all the AI-generated code in GitHub projects, as there is no direct way to identify if part of a file was generated by code generation tools such as Copilot (i.e., source files do not contain any signatures to indicate if Copilot generates the code). However, we can identify many code snippets by searching the repository description and the comments provided in the code (see the details in Section 3.2.2).

We first used keyword-based search to collect projects and files containing code generated by Copilot and other AI code generation tools (CodeWhisperer and Codeium) from GitHub and filtered them manually. We subsequently analyzed the functionality and application domains of the collected code snippets to get a comprehensive understanding of the dataset. Next, we performed a static security analysis on the identified code.

After running the analysis and obtaining the analysis results, we manually checked the results to remove false positives reported by the static analysis tools. We then used CWEs to classify the filtered results for further analysis to answer our Research Questions (RQs). In addition, we used *Copilot Chat* [27] to fix code snippets containing security weaknesses. *Copilot Chat* is an interactive tool launched by OpenAI based on the GPT-4 model, designed to enable natural language-driven coding as part of Copilot. Developers can use *Copilot Chat* to perform tasks such as code analysis and fixing bugs. We used the warnings from static analysis tools as prompts and fed them to *Copilot Chat* in order to get the fixes. We then performed a security analysis on the fixed code to evaluate the effectiveness of those fixes.

To conduct our empirical study, we followed the guidelines of Easterbrook *et al.* [17]. The RQs, their rationale, and the research process of this study (see Figure 3) are detailed in the following subsections.

RQ1. How secure is the code generated by Copilot in GitHub projects?

Rationale: Copilot may produce code suggestions that developers accept, but these suggestions may include security weaknesses that could make the program vulnerable. The answer to RQ1 helps understand the frequency of security weaknesses developers encounter when using Copilot in production. We also include the code generated by other two popular AI code generation tools, CodeWhisperer and Codeium, in order to increase the

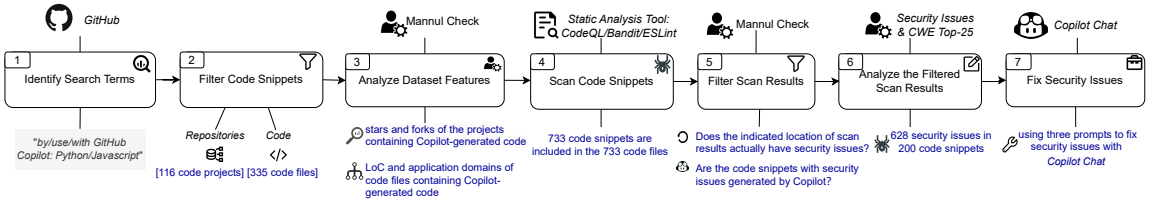


Fig. 3. Overview of the research process

generalizability of this research.

RQ2. What security weaknesses are present in the code generated by Copilot across different application domains?
Rationale: Copilot-generated code may contain security weaknesses [43, 55], and developers should conduct a rigorous security review before accepting the generated code suggestions. Copilot's documentation notes the following: "*users of Copilot are responsible for ensuring the security and quality of their code*" [24]. By identifying common or recurring weaknesses, developers can be more prepared to prevent, mitigate, or fix these security weaknesses. The answer to RQ2 can help developers better understand possible security weaknesses in the code generated by Copilot and other AI code generation tools, as well as the characteristics of code that contains certain types of security weaknesses, allowing them to be vigilant about the generated code before it is integrated into their code base.

RQ3. Can *Copilot Chat* help fix the security weaknesses found in the code generated by Copilot?
Rationale: The purpose of *Copilot Chat* is to assist developers by answering questions about their code or providing fixes for issues identified in the code. *Copilot Chat* is integrated with Copilot to form a powerful AI assistant that can help developers build at their speed of thought in the natural language of their choice [23]. It is claimed that *Copilot Chat* can assist in fixing security weaknesses. In this RQ, we utilize *Copilot Chat* to examine whether it can help fix the security weaknesses identified in the Copilot-generated code because most of the security weaknesses in RQ1 and RQ2 come from Copilot. Our approach is to utilize *Copilot Chat* in the loop by *collect generated code →analyze the code for security weaknesses →check for fixes in Copilot Chat →re-analyze the code to check if the fix is correct.* Answering RQ3 can help to understand the capability of *Copilot Chat* in fixing Copilot-generated code that contains security issues.

3.2 Data Collection and Filtering

We chose GitHub as the primary data source for answering our RQs. As the world's largest code hosting platform, GitHub contains millions of public code repositories and offers access to a large number of code resources, allowing us to cover multiple programming languages and project types in our study [11]. At the same time, thousands of developers in the GitHub community have shared their experiences of using Copilot in open-source projects. We identified code snippets generated by AI code generation tools from GitHub projects.

We chose Copilot as our main research subject as it is a well-known commercial instance of an AI pair programming tool that has gained increased popularity among developers since its launch in 2021. Copilot is one of the most popular general-purpose AI code generation tools and only second to ChatGPT in terms of overall AI assistance tools developers use in practice [53]. The security drawbacks of code generated by Copilot have been investigated in previous studies [15, 44, 55]. To increase the generalizability of the research, we also attempted to collect data from two popular AI code generation tools, CodeWhisperer and Codeium - both are general-purpose,

AI-driven code generators. We did not collect data from specialized AI code generation tools, such as AlphaCode as it focuses on solving complex coding tasks rather than everyday code generation.

3.2.1 Code Snippets Collection. **Step 1.** We utilized GitHub REST API to automatically collect projects and files containing AI-generated code. We started with a pilot search on GitHub to formulate our search keywords. First, we used the names of the code generation tools, such as "GitHub Copilot" and "CodeWhisperer" as the basic search terms. It should be noted that many code snippets in the search results contained the "GitHub Copilot" keyword, but the keyword was not used to declare that the code snippets were generated by Copilot. Developers may use them to describe their experience of using Copilot to generate code or showcase information related to Copilot. These code snippets are not what we seek as they do not directly relate to the code generated by Copilot. By manually analyzing the search results, we found that restricting the search terms provided more accurate results. We decided to use keyword combinations such as {*by, use, with*} + {*GitHub Copilot, CodeWhisperer, Codeium*} as search keywords. This approach enabled us to focus more on the code generated using AI code generation tools (e.g., Copilot) rather than code snippets that contain other content related to the generation tools.

In addition, we further limited the types of code snippets during the search to Python and JavaScript. Both are dynamically typed languages where a full analysis can be conducted without the need for compilation. On the other hand, statically typed languages such as Java and C++ require the code to be compiled first to generate the database required for analysis. Our data collection method makes it difficult to properly compile individual code snippets, given that some code is unavailable when conducting the analysis.

We selected Python and JavaScript as they are currently the two most popular programming languages used by developers [28], and the most frequently used with GitHub Copilot [84]. The popularity of the two languages ensures that we can collect a substantial amount of generated code from GitHub projects, providing a significantly larger sample size and statistically meaningful results compared to other languages. We collected the search results under the filter *Code* with file names, file paths, and forks and stars of the repositories where the files were located. Considering that some projects declared using GitHub Copilot generated code in their README files or project descriptions provided in GitHub, we decided to retain the results from the filter *Repo* with project name, project path, project description, and forks and stars of the projects.

Table 1 shows a breakdown of the search terms we used and the number of search results per language. The same search result may contain multiple keywords, meaning there are duplicate projects in the collected data. After removing duplications, we ended up with 3,589 search results, of which 3,141 were from the *Code* label and 445 from the *Repository* label.

3.2.2 Filtering Code Snippets. **Step 2.** After obtaining the data from the keyword searches, we further filtered them by not only considering the accuracy of the keywords but also by investigating the project's documentation, code comments, and other metadata in the search results to determine whether they were generated by GitHub Copilot and other code generation tools. Additionally, since we aimed to obtain code snippets used in open-source GitHub projects, we manually excluded search results related to solving simple programming practice problems on platforms that provide foundational exercises for enhancing algorithmic thinking and coding skills, such as bisection lookup and fast sorting tasks from LeetCode. We consider these programming exercises to be more closely related to correctness than security.

We begin by explaining the terminology used in data filtering: the search results under the *Repository* label are the projects that contain code files, and the search results under the *Code* label are individual code files. Those code files contain code snippets generated by Copilot. In filtering the projects, we followed **three criteria** including two *inclusion criteria* and one *exclusion criterion. Inclusion Criterion 1*: For search results under the *Repository* label, we identified projects that are fully generated by Copilot, as declared in the project description or the associated README file(s). We retained code files for Python and JavaScript. *Inclusion Criterion 2*: For search results under

Table 1. Search results based on different terms

Search Term	Code (Py)	Code (Js)	Repo (Py)	Repo (Js)	Total
by GitHub Copilot	1320	1256	48	43	**2667**
use GitHub Copilot	1344	1384	84	93	**2905**
with GitHub Copilot	1532	1876	95	177	**3680**
by CodeWhisperer	39	44	0	1	**84**
use CodeWhisperer	41	46	4	2	**93**
with CodeWhisperer	68	39	4	0	**111**
by Codeium	85	73	0	1	**159**
use Codeium	31	26	0	1	**58**
with Codeium	99	69	0	2	**170**
Total	**4559**	**4813**	**235**	**320**	**9927**

the *Code* label, we retained code files with comments showing the code generated by Copilot. *Exclusion Criterion 1*: We excluded code files used to solve simple programming practice problems. Simple programming practice problems mainly refer to those standard coding exercises on platforms such as LeetCode, which are often named BOJ-number. We provide examples for the three criteria in Figure 4, Figure 5, and Figure 6. As shown in the example in Figure 4 for the *Repository* label, we kept all the Python files. In the next example in Figure 5, we kept the entire file where the Copilot-generated code snippet was located. In Figure 6, the code snippet was removed as it was determined that the code just solved a simple algorithmic problem.

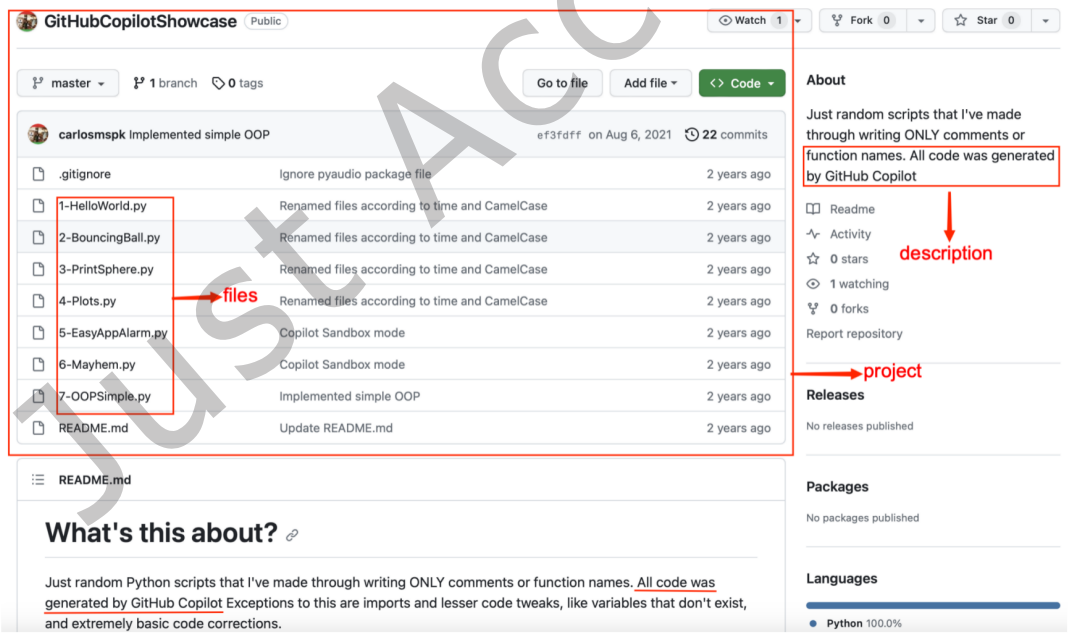


Fig. 4. Example of *Inclusion Criterion 1*: projects fully written by Copilot

```python
# Github Copilot wrote this class with just the class name as prompt!
class EarlyStopper:
    def __init__(self, patience=10, decimal=5):
        self.patience = patience
        self.decimal = decimal
        self.reset()

    def track(self, loss):
        if np.around(loss, self.decimal) >= np.around(self.best_loss, self.decimal):
            self.num_bad_epochs += 1
        else:
            self.num_bad_epochs = 0
            self.best_loss = loss
        if self.num_bad_epochs >= self.patience:
            return True
        return False
```

Prompt message

Fig. 5. Example of *Inclusion Criterion 2*: files with comments showing the code generated by Copilot

Code | Blame | Raw

```cpp
// BOJ 2438 [Printing Stars 1]
// Supported by GitHub Copilot

#include <bits/stdc++.h>
using namespace std;

int main() {
    int N; cin >> N;
    char s[100]; fill(s, s+100, '*');
    for(int i=1; i<=N; i++) s[i] = 0, puts(s), s[i] = '*';
}
```

BOJ refers to an algorithm problem on LeetCode platform

Fig. 6. Example of *Exclusion Criterion 1*: files used to solve simple algorithm problems

We manually filtered all the data, labeling whether they were generated by Copilot and other code generation tools based on the three criteria mentioned above. Two authors independently completed this manual filtration process for all the data within a period of two weeks. The first author compared the labeling results by the two coders and calculated the level of agreement between them using Cohen's Kappa coefficient [9]. Cohen's Kappa coefficient was 0.84, which was higher than 0.8, indicating a high level of agreement between the two coders and ensuring a good accuracy of the labeling results. Besides, for those results where the two authors disagreed on the labeling, they resolved the disagreements through a negotiated agreement approach [5] during a meeting. After manual filtration, we retained 116 projects under the *Repository* label and 335 code files under the *Code* label.

For the code files retained under the *Repository* label, we consider the entire code file as code generated by Copilot and other code generation tools. In other words, we assume that Copilot and other code generation tools generate all the code in the file because it was stated in the README file that it was all generated by Copilot and other code generation tools. For code files retained under the *Code* label, we know that the files contain code snippets, perhaps even just a few lines of code, generated by Copilot and other code generation tools. Instead of identifying the specific AI-generated code in this step, we combine the warning messages from the security analysis and the code comments in the file to determine whether Copilot and other code generation tools generate

the code snippet with the security problem (this process is explained further in Section 3.3.2). As a result, our dataset consisted of 733 distinct code files, with 672 from Copilot, 38 from CodeWhisperer, and 23 from Codeium. Table 2 gives the type and number of code files. A curated dataset, comprising all the data collected during the research process, has been made available [21].

Table 2. Code snippets from GitHub

#	Language	# Code Snippets: Repository	# Code Snippets: Code	Total
L1	Python	190	229	419
L2	JavaScript	208	106	314
Total		398	335	**733**

Step 3. To understand the features of our dataset, we first conducted a statistical analysis of the stars and forks of the projects where AI-generated code was present. The distribution of stars and forks are shown in Figure 7 and Figure 8, respectively. We found that the projects containing AI-generated code often tend to be small and have low popularity in the GitHub community, possibly often developed by individuals or small teams. This finding is consistent with the recent study by Yu *et al.* about the characteristics of LLM-generated code on GitHub [83]. We speculated that the low popularity may be correlated with the age of the project. Most of these projects are relatively new as Copilot itself is also a recent development. Despite the low popularity and user engagement of most projects, there are a few exceptions in the dataset, such as the project `OnmyojiAutoScript`, which received 1,672 stars and 505 forks.

In addition, we also counted the lines of code (LoC) of the collected code files. Figure 9 shows the distribution of LoC in different languages (Python and JavaScript respectively) and then the LoC of all code files (Python + JavaScript) in the dataset. We found that the median LoC of code is 74, while the average reaches 181. Although the files obtained from GitHub are relatively small in size, they are still significantly larger than the size of code generated using designed scenario prompts. For example, when Majdinasab *et al.* [43] replicated the experiment by Pearce *et al.* [55], the median LoC of code generated by their designed prompts through Copilot was 42, with an average of 40. While using the SecurityEval dataset [68] as prompts to test Copilot, the LoC of generated code is concentrated between 5 and 26. This indicates that our collected dataset has an advantage in terms of the size of the generated code.

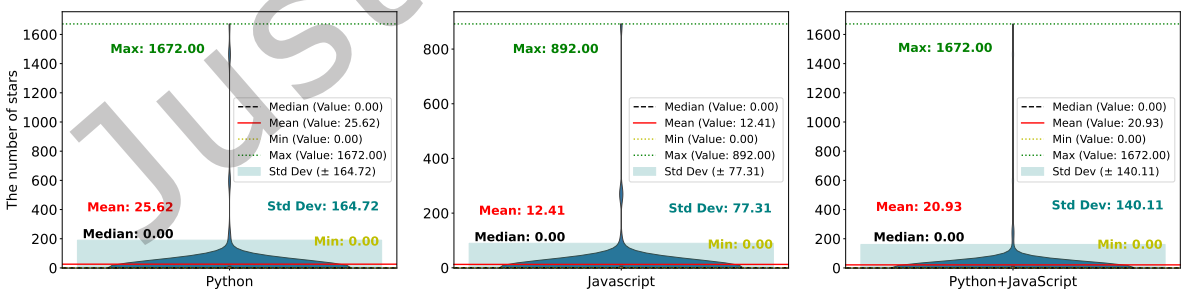


Fig. 7. Distribution of stars of the projects where AI-generated code was present

We also analyzed the application domains of the code snippets in our dataset. To understand the application domains, we classified the code in the dataset based on the project description and the specific functions of the

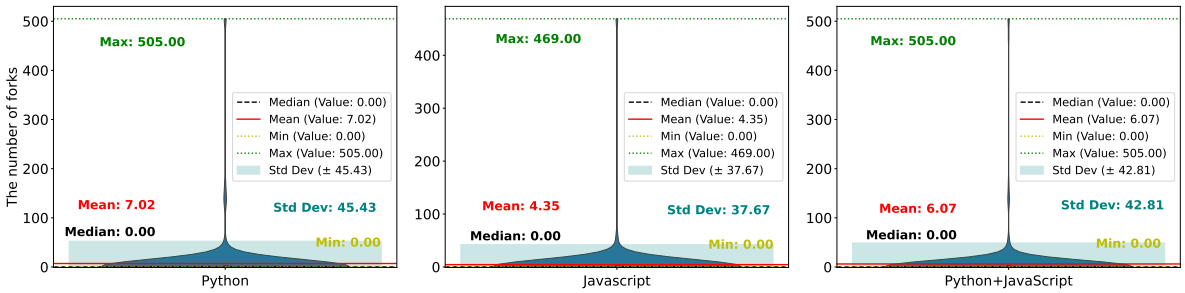


Fig. 8. Distribution of forks of the projects where AI-generated code was present

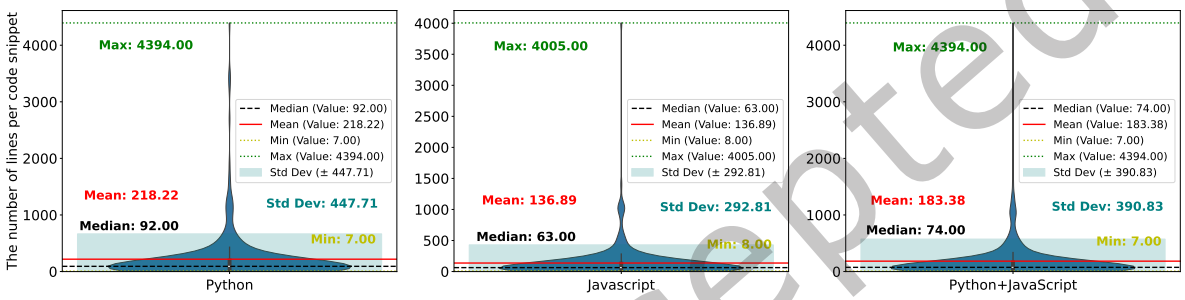


Fig. 9. Distribution of LoC of the files where AI-generated code was present

code files. Two authors independently classified all code snippets within a period of ten days. If their categorization results diverged, the two authors discussed until they reached a consensus. In total, we categorized the generated code into six application domain categories: Game, Web Application, Utility Tool, AI Application, Network Communication, and Others. Table 3 shows the descriptions of application domains and the categorization of code snippets in different programming languages.

3.3 Data Analysis and Results Filtering

3.3.1 Data Analysis. **Step 4.** We used well-known automated static analysis tools suggested by OWASP [54] to scan the collected code snippets. The reason why we chose a static rather than dynamic approach is that the static analysis has higher coverage and is able to analyze programs without executing them [16]. While dynamic analysis offers more precise insights by reasoning about program behavior, it suffers from limited coverage and can be expensive [19, 72, 73]. Static security analysis tools have been widely used by developers and companies [13, 63]. Employing static analysis enables us to analyze program segments (smaller snippets) without requiring the entire program to be executed. Since different static analysis tools may use different algorithms and rules to detect security weaknesses, using multiple tools can increase our chances of discovering security weaknesses in the code. To improve the coverage and accuracy of the results, we used two static analysis tools for security checks on each code snippet (i.e., CodeQL plus one dedicated tool for the specific language, Bandit for Python, and ESLint for JavaScript).

CodeQL is a scalable static security analysis tool that is widely used in practice and allows users to analyze code and detect relevant weaknesses using predefined queries and test suites and supports for multiple languages (including Java, JavaScript, C++, C#, Go, and Python [22]). Before using CodeQL to scan the identified code

Table 3. Application domains of the files where AI-generated code was present

Category	Description	Python	JavaScript
Utility Tool	Involves scripts, command-line tools, and other programs that simplify daily tasks.	39.0%	30.1%
Web Application	Code related to front-end, back-end development, and server-side operations.	15.6%	49.8%
AI Application	Code for machine learning, deep learning, natural language processing, or other intelligent applications.	15.6%	2.8%
Game	Contains code related to game development, using engines such as Unity, Unreal, and Godot.	15.9%	8.5%
Network Communication	Involves code for implementing network protocols, client/server communication.	8.2%	5.7%
Others	Code that does not fit into the above categories.	5.4%	2.8%

snippets for security weaknesses, we needed to create a CodeQL database for the source code. Source code can be directly analyzed for interpreted languages like Python and JavaScript without being compiled into intermediate code. Thus, we directly use the command line to generate the database needed for CodeQL queries. Then, we used CodeQL to analyze the generated database. We can find the available query suites for different programming languages in the `codeql` repository. For example, the `codeql/python-queries` package contains the following query suites [26]:

- `Python-code-scanning.qls`, the standard scanning query for Python. It covers various features and syntax of Python and aims to discover some common weaknesses in the code.
- `Python-security-extended.qls`, which includes some more advanced queries than `Python-code-scanning.qls` and can detect more security weaknesses.
- `Python-security-and-quality.qls`, which combines queries related to security and quality, covering various aspects of Python development, from basic code structure and naming conventions to advanced security and performance weaknesses. It aims to help developers improve the security and quality of their code.

In this study, we scanned code snippets using the `<language>-security-and-quality.qls` test suite. These test suites check for multiple security properties and cover many CWEs. For example, the `python-security-and-quality.qls` test suite for Python provides 168 security checks, and the JavaScript test suite provides 203 security checks. As the query reports only provide the name and description of the security weaknesses, we manually matched the results in the query reports with the corresponding CWE IDs based on the official documentation provided [8].

For Python files, we used Bandit [59] – a Python's static security analysis tool. When running Bandit, we enabled all security rules by default. Bandit had a total of 73 security check rules, and each Bandit rule was typically associated with a corresponding CWE number, helping developers better understand the categories of detected vulnerabilities and align with industry standards. For JavaScript files, we used ESLint [20] – a widely used JavaScript static analysis tool that can detect common coding issues and security vulnerabilities. Since we needed to configure the security check rules ourselves, we used the `eslint-plugin-security` plugin for security checks, which is an ESLint plugin containing 14 security check rules designed to detect common

security vulnerabilities in JavaScript. In addition, we also configured the plugins `eslint-plugin-no-secrets` and `eslint-plugin-xss` with more precise rules to identify security weaknesses more accurately. For example, some of the rules in `eslint-plugin-security` also deal with XSS protection (such as avoiding unsafe `eval()` usage), but `eslint-plugin-xss` provides more precise rules against front-end XSS attacks. Both tools accept raw code snippets as input, and there is no need to pre-process the data before using them. It is worth mentioning that the detection rules of these static analysis tools do not have a one-to-one correspondence with CWEs. For example, multiple detection rules in Bandit may identify security issues that can all be classified under *CWE-78: OS Command Injection*. As a result, we did not explicitly provide the detection rule-to-CWE correspondences. Instead, we manually mapped the detection rules to the CWEs after using these static analysis tools in conjunction with specific warning messages. We have provided the detection rule-to-CWE correspondences in our replication package [21].

As explained in Section 3.2, we considered the code snippet from the *Repository* label to be the entire code file, while the code snippet from the *Code* label exists in the code file, with the exact location of the code snippets unspecified at this stage.

3.3.2 Results Filtering. **Step 5.** We scanned code snippets from the *Repository* and *Code* labels, and we filtered the analysis results before analyzing them. In this step, we adopted part of the strategies used by developers when they perform different tasks with static analysis tools: *warning prioritization* and *determining whether a warning is a false positive or a true report* [14]. We first performed an initial filtering of the results based on the priority of the warnings. Specifically, we manually removed the duplicate analysis results that were reported by the two tools (14 duplicate results by Bandit and 2 duplicate results by ESLint), keeping only one for further analysis. The first author eliminated analysis results that were not security weaknesses. We identified three types of warnings from CodeQL analysis:

- *Recommendation*, which provides suggestions for improving code quality;
- *Warning*, which alerts to potential weaknesses that could cause code to run abnormally or unsafely;
- *Error*, which is the highest level of warning and alert to inform that the error could cause code to fail to compile or run incorrectly.

Since our research primarily focused on security weaknesses, we only counted code snippets that had *warnings* and *errors*, and we ignored *recommendations* on code quality.

After obtaining preliminary analysis results related to security weaknesses, we manually checked the analysis results to confirm that the security weaknesses were actually caused by the automatically generated code. For the initial results obtained under the *Repository* label, we checked them one by one. Specifically, we determined whether the corresponding location of the code snippet indeed had a security weakness based on the line number information provided by the analysis results. For the initial results obtained under the *Code* label, we confirmed whether the security weaknesses truly existed and verified if the security weaknesses were caused by Copilot-generated code. Specifically, after scanning the code file, we pinpointed the code snippet within the file based on the line number of the security weakness indicated in the analysis results. We assessed whether it was generated by Copilot or other generation tools by checking the surrounding comments and determined whether a security weakness existed in that particular context. We further analyzed the filtered analysis results in **Step 6**, detailed in Section 4, according to the specific RQs.

3.4 Fixing Code with Security Weaknesses

Step 7. Based on RQ2 results, we randomly selected a subset of code snippets containing security weaknesses as the repair dataset since our purpose is to explore the feasibility of *Copilot Chat* in fixing the security weaknesses.

These code snippets were all generated by Copilot, as most of the code obtained from GitHub was produced by Copilot. Since we aimed to fix as many security weaknesses as possible, we did not use clean data (i.e., code snippets without known security weaknesses). We randomly selected 50 Python and 40 JavaScript code snippets from the *Repo* label, which contained 163 and 132 security weaknesses, respectively. We utilized *Copilot Chat* to remediate the security weaknesses identified by the static analysis tools. We conducted several experiments on the selected dataset with 295 security weaknesses using three different prompts. *Copilot Chat* provides developers with basic slash commands, avoiding the need to write complex prompts for common scenarios. The slash commands include: `/tests`: generates unit tests for the selected code. `/fix`: provides fix suggestions for issues in the selected code. `/explain`: explains the selected code. `/optimize`: analyzes and improves the runtime of the selected code. We first used the slash command `/fix` to provide fix suggestions for the selected code. Then, we performed the fixes using our designed basic prompts and enhanced prompts with additional context. The prompt designs are as follows:

Basic Prompt

Is there a security weakness with this code snippet? If so, please fix it and return the fixed code.

Enhanced Prompt

This code snippet is known to have some security weaknesses. The warning messages are as follows: `<insert warning messages here>`. *Please fix these security weaknesses and return the fixed code.*

Figure 10 shows an example of how to use *Copilot Chat* to fix the security weaknesses in the code. We first select code snippets with security weaknesses and provide the three prompts (`/fix`, basic prompt, and enhanced prompt) in the IDE chat window. Then, *Copilot Chat* uses the selected code as a reference (input) to generate responses to our prompts, which we can accept or reject. Considering the token limitation issue, we always selected the function block containing the security weaknesses as the reference for *Copilot Chat*. After accepting the fix suggestions from *Copilot Chat*, we used the static analysis tools to recheck the modified code snippets. We recorded the positions of the modified code, indicating where the accepted code is located. Then we examined whether security weaknesses still exist in the corresponding positions of the modified code according to the rechecking results of the static analysis tools.

We provided the complete dataset (including code snippets, full analysis results, and filtered results) in our replication package [21].

4 RESULTS

In this section, We present the results of the three RQs formulated in Section 3.1. For each RQ, we first explain how we analyzed the collected code snippets to answer the RQ, and we then answer each of our three RQs.

4.1 RQ1: How secure is the code generated by Copilot in GitHub projects?

Approach. To answer this RQ, we collected 733 Python and JavaScript code snippets generated by Copilot and two other tools from GitHub projects, with 672 from Copilot, 38 from CodeWhisperer, and 23 from Codeium. We used two static analysis tools (CodeQL + another language-dedicated tool, Bandit for Python, and ESLint for JavaScript) to scan. The aim is to achieve better coverage of security weaknesses. Therefore, as long as one of the tools detected the presence of a security weakness, the code snippet was considered vulnerable.

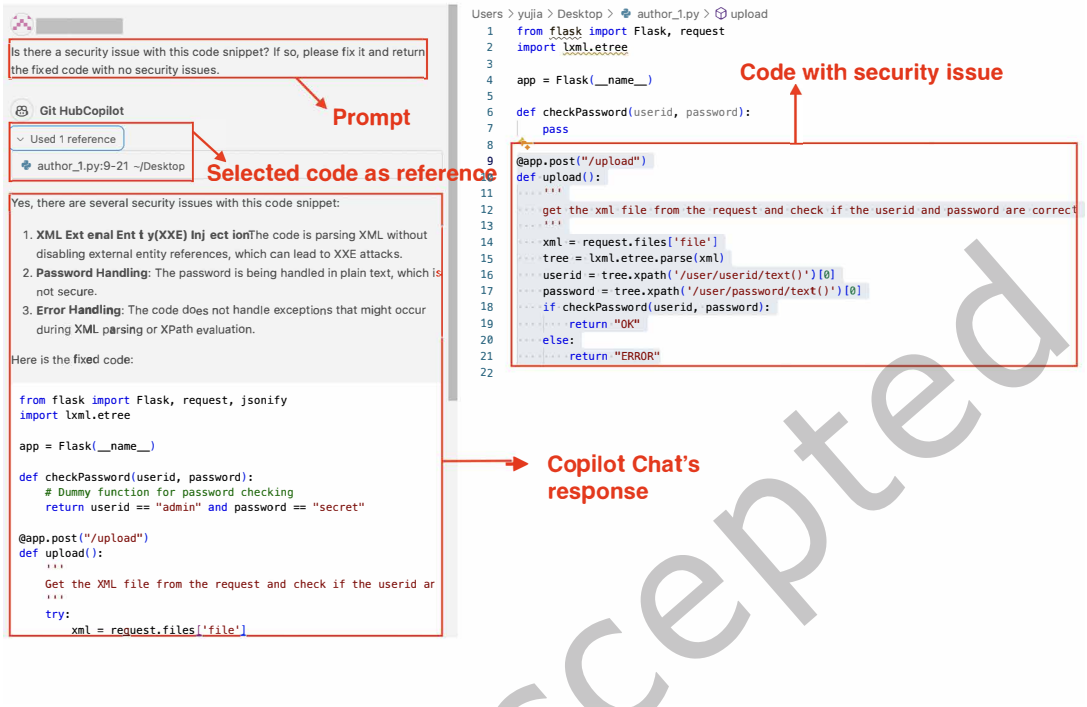


Fig. 10. An example of using *Copilot Chat* to fix security weaknesses in Visual Studio Code

Subsequently, we filtered the analysis results as described in Section 3.3.2. For the analysis results from the *Repository* label, we marked them as 1 (security weakness exists) and 0 (no security weakness exists). For the analysis results from *Code* label, we marked them as 1 (security weakness exists) and 0 (no security weakness exists or the code snippet with a security weakness was not generated by code generation tools). When manually checking the analysis results from the *Code* label, we also needed to identify whether the security weaknesses obtained from the scan were from automatically generated code snippets based on the comment (prompt message) that appears before the method. We provide a working example of the filtration of the analysis results in Figure 11. In **Step 1**, we first went to the corresponding file to locate the specific code snippet based on the start line numbers of the analysis results. In **Step 2**, we located the code at Line 149 in `aggregation.py`. We found that this code does have a security weakness and determined that it was generated by Copilot based on the prompt messages above. Consequently, we marked the corresponding security analysis result as "1". We also located the code at Line 145 in `_lytools.py`. We found that this code had a security weakness, but there were no prompt messages that would indicate that it was generated by Copilot. Consequently, we marked the corresponding security analysis result as "0" and discarded this analysis result from further analysis.

Two authors independently filtered the analysis results within a period of two weeks, and if there were any results that they were unsure of or disagreed with, the two authors discussed the results until they reached an agreement. Cohen's Kappa coefficients [9] were 0.85 for the analysis results from the *Repository* label, and 0.82 for the analysis results from the *Code* label, which were both higher than 0.8, ensuring a good accuracy of

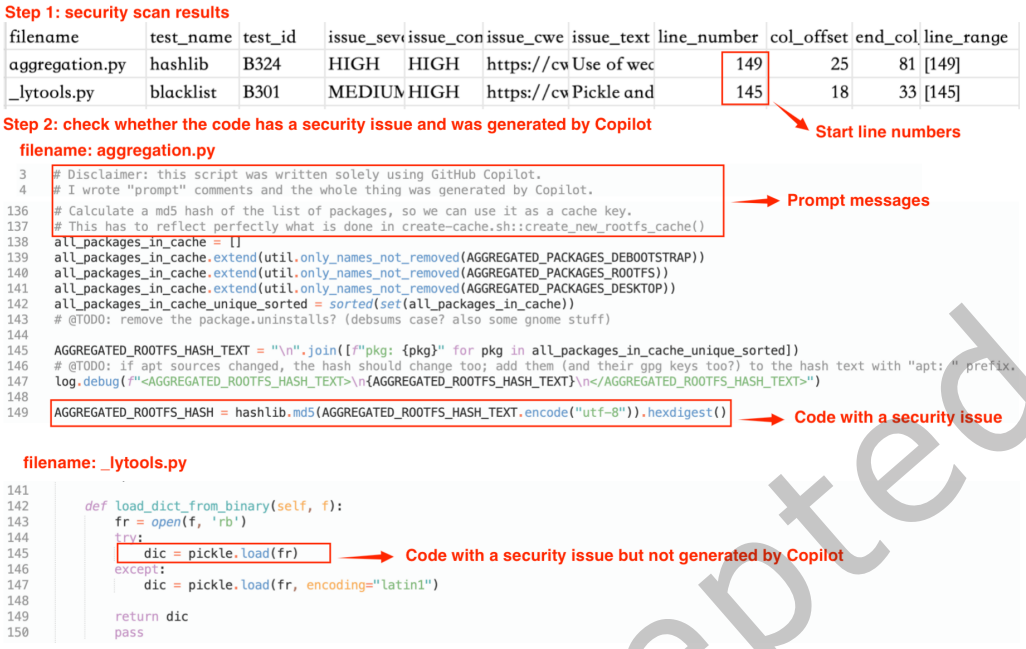


Fig. 11. Example of filtering analysis results from the *Code* label that are generated by Copilot

the filtering results. Finally, we kept the results marked as 1 and aggregated the filtered results obtained using multiple analysis tools to calculate the number of code snippets with security weaknesses detected. All labeling results can be found in our online replication package [21].

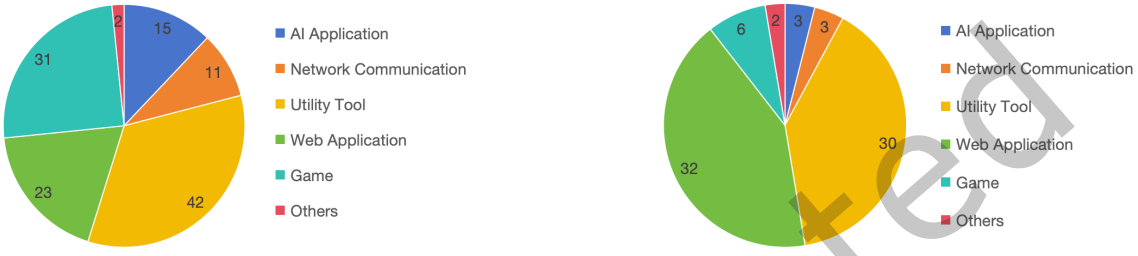
Results. Table 4 shows the number of code snippets containing security weaknesses and the number of security weaknesses identified in the code snippets. From the statistical results, we found that out of the 733 generated code snippets, 27.3% of them contained security weaknesses. Around 30% of Python code snippets (124 of the 419) and around 25% of JavaScript code snippets (76 of the 314) contained security weaknesses. At the same time, we can see that more than half of the code snippets containing security weaknesses (102 of 200) have more than one security issue. In Figure 12, we present the distribution of application domains for the 200 code snippets (27.3% of all) identified with security weaknesses. We also found 628 security weaknesses in the 200 code snippets. Note that one code snippet may contain multiple security weaknesses attributed to multiple CWEs. Figure 13 shows the distribution of security weaknesses per code snippet. Our analysis reveals an average incidence of 3 security weaknesses per code snippet, irrespective of the programming language. The maximum count of security weaknesses within a single code snippet, which consisted of 1,595 LoC, reached 22.

4.2 RQ2: What security weaknesses are present in the code generated by Copilot across different application domains?

Approach. To answer RQ2, we processed the analysis results collected by RQ1, which were manually checked to see if they contained security weaknesses. For each code snippet, we used CWEs to classify the security weaknesses identified by the static analysis tools. Each CWE has a unique ID and a set of related descriptions, including its potential impact and how to detect and fix the CWE [46]. To accurately identify the type of security weakness with the corresponding CWE, we did not directly accept the information provided by the static analysis

Table 4. Statistics of the code snippets with security weaknesses

Language	# Snippets	# Security weaknesses	# Snippets containing security weaknesses	# containing more than one security weakness
Python	419	387	124 (29.5%)	65 (15.5%)
JavaScript	314	241	76 (24.2%)	37 (11.8%)
Total	**733**	**628**	**200 (27.3%)**	**102 (13.9%)**



(a) application domains of Python code

(b) application domains of JavaScript code

Fig. 12. Application domains of the code snippets with security weaknesses

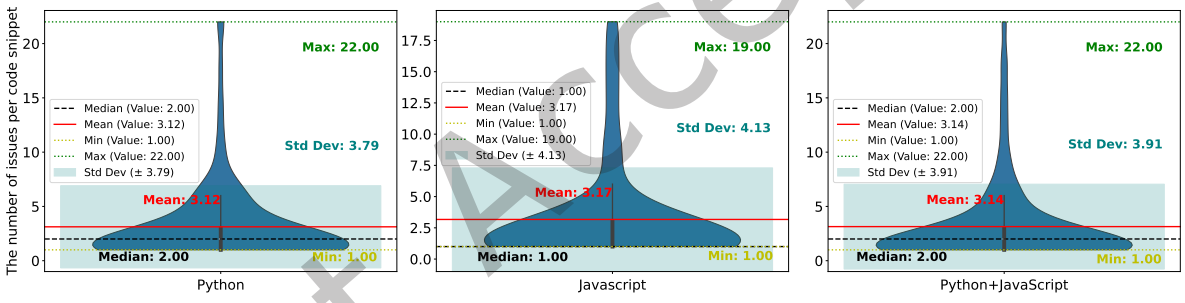


Fig. 13. Distribution of the number of security weaknesses per code snippet

tools. Instead, we manually identified the type of security weakness at the specific location reported in the warning message of the static analysis tool. Only Bandit provided the CWE IDs of the security issues identified, and we did not follow the CWE IDs of 55 out of 235 security issues reported by Bandit. For example, Bandit reported "*Starting a process with a partial executable path*" as belonging to *CWE-78: OS Command Injection*. After our manual assessment, we concluded that it was more appropriate to map this warning message to *CWE-427: Uncontrolled Search Path Element*, as the security issue was primarily related to path control rather than command construction. We utilized the warning information provided by static analysis tools, such as the test_name and message fields indicated by CodeQL, in conjunction with relevant descriptions from the CWE, to perform manual correspondences. Initially, two authors independently matched each description of the security issue with a CWE ID within a period of ten days. Cohen's Kappa coefficient [9] was 0.82, which was higher than 0.8 and indicated a high level of agreement between the two authors. In case of disagreement, a discussion was initiated between the two authors, and one other author (a security expert) was then involved to provide his assessment. We provided

correspondences between the warning prompt messages and CWEs in our online replication package [21]. In the final stage, we performed a statistical analysis of CWE weaknesses in 200 code snippets that contained security weaknesses. Additionally, we analyzed the distribution of different CWEs across different application domains of the code snippets.

We also discussed whether the security weaknesses present in the code generated by Copilot are widely prevalent in the code produced during software development. The code snippets in our collected dataset were mainly generated in 2022 and 2023. To compare whether the security weaknesses in in our dataset are widespread in this period, we chose MITRE 2023 CWE Top-25 list [45] as our baseline. We compared the CWEs obtained in RQ2 with the 2023 CWE Top 25.

Table 5. Distribution of CWEs in code snippets

CWE-ID	Name	Frequency	Percentage
CWE-330	Use of Insufficiently Random Values Weakness	114	18.15%
CWE-94	Improper Control of Generation of Code ('Code Injection')	62	9.87%
CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	60	9.55%
CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	39	6.21%
CWE-427	Uncontrolled Search Path Element	35	5.57%
CWE-457	Use of Uninitialized Variable	30	4.78%
CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	29	4.62%
CWE-772	Missing Release of Resource after Effective Lifetime	29	4.62%
CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	27	4.30%
CWE-259	Use of Hard-coded Password	16	2.55%
CWE-685	Function Call with Incorrect Number of Arguments	14	2.23%
CWE-284	Improper Access Control	13	2.07%
CWE-312	Cleartext Storage of Sensitive Information	12	1.91%
CWE-390	Detection of Error Condition Without Action	11	1.75%
CWE-456	Missing Initialization of Variable	11	1.75%
CWE-607	Use of Wrong Operator in String Comparison	11	1.75%
CWE-770	Allocation of Resources Without Limits or Throttling	11	1.75%
CWE-20	Improper Input Validation	8	1.27%
CWE-665	Improper Initialization	8	1.27%
CWE-117	Improper Output Neutralization for Logs	7	1.11%
CWE-400	Uncontrolled Resource Consumption	7	1.11%
CWE-561	Dead Code	7	1.11%
CWE-732	Incorrect Permission Assignment for Critical Resource	6	0.96%
CWE-798	Use of Hard-coded Credentials	6	0.96%
CWE-215	Information Exposure Through Debug Information	5	0.80%
CWE-290	Authentication Bypass by Spoofing	5	0.80%
CWE-295	Improper Certificate Validation	5	0.80%
CWE-209	Information Exposure Through an Error Message	4	0.64%
CWE-252	Unchecked Return Value	4	0.64%
CWE-571	Expression is Always True	4	0.64%
CWE-605	Multiple Binds to the Same Port	4	0.64%
CWE-200	Information Exposure	3	0.48%
CWE-327	Use of a Broken or Risky Cryptographic Algorithm	3	0.48%
CWE-367	Time-of-check Time-of-use Race Condition	3	0.48%
CWE-563	Assignment of a Fixed Address to a Pointer	3	0.48%

Continued on next page

Table 5 – continued from previous page

CWE-ID	Description	Frequency of Specific CWE	Percentage
CWE-116	Improper Encoding or Escaping of Output	2	0.32%
CWE-480	Use of Incorrect Operator	2	0.32%
CWE-502	Deserialization of Untrusted Data	2	0.32%
CWE-601	URL Redirection to Untrusted Site ('Open Redirect')	2	0.32%
CWE-208	Observable Timing Discrepancy	1	0.16%
CWE-570	Expression is Always False	1	0.16%
CWE-628	Function Not Implemented Correctly	1	0.16%
CWE-682	Incorrect Calculation	1	0.16%
43 Types		**Total: 628**	

Results. Table 5 shows the distribution of CWEs in the code snippets and the total number of occurrences (frequency) of the CWEs in the code snippets. In total, we found 628 CWEs in 200 code snippets with security weaknesses. These security weaknesses were related to 43 CWE types, indicating that developers face a variety of security weaknesses when using Copilot and other code generation tools. *CWE-330: Use of Insufficiently Random Values* is the most frequently occurring CWE, as it represents 18.15% of the security weaknesses, followed by *CWE-94: Code Injection* , *CWE-79: Cross-site Scripting*, and *CWE-78: OS Command Injection*. Some CWEs appear less frequently. For example, *CWE-117: Improper Output Neutralization for Logs* only occurred twice. Furthermore, many CWEs occur with a frequency of less than 1%, for example, *CWE-117: Improper Output Neutralization for Logs*and *CWE-290: Authentication Bypass by Spoofing*. This indicates that the types of security weaknesses are closely related to the specific scenarios in which developers use code generation tools, emphasizing the importance of maintaining vigilance and caution when programming. In addition, we highlighted those CWEs that are the "Top-25 CWEs" in **bold**. It is worth noting that 233 security weaknesses in the code snippet correspond to these eight CWEs, which belong to the Top-25 CWE list, indicating that the CWE Top-25 weaknesses are also prevalent in the code generated by Copilot and other tools. At the same time, we can see that *CWE-79: Cross-site Scripting* and *CWE-78: OS Command Injection* are among the most frequently occurring security weaknesses in our results and rank high on the Top-25 CWE list.

Table 6 presents the top 5 CWEs that appear in Python and JavaScript code snippets. We found that the detected CWEs vary across programming languages. CWEs in Python are mainly related to data processing and system calls, such as *CWE-78: OS Command Injection* and *CWE-427: Uncontrolled Search Path Element*. In contrast, CWEs in JavaScript are primarily associated with dynamic code generation problems and security issues in Web development, such as *CWE-94: Code Injection* and *CWE-79: Cross-site Scripting*.

Furthermore, Figure 14 and Figure 15 show the distribution of CWEs across different application domains. We can observe that the number and type of instances of CWE vary from one application domain to another. In addition to this, the distribution of CWEs within the same application domain varies by programming language. For example, in Python, the application domain with the most CWEs is Utility Tool, while in JavaScript, Web Applications have the most CWEs. Meanwhile, we can see that in the Web Application category, *CWE-79: Cross-site Scripting* is the most frequent in JavaScript code snippets, while *CWE-89: SQL Injection* is the most present in Python code snippets.

4.3 RQ3: Can Copilot Chat fix the security weaknesses of code generated by Copilot?

Approach. To answer RQ3, we conducted experiments using *Copilot Chat* on the randomly selected code snippets containing security weaknesses and asked *Copilot Chat* to provide fix suggestions based on the three prompts (see Section 3.4). We then fixed the code according to the suggestion provided by *Copilot Chat*. Specifically, we

Table 6. Top 5 CWEs in Python and JavaScript

Rank	CWE Type	
	Python	**JavaScript**
1	CWE-330 (Use of Insufficiently Random Values Weakness)	CWE-94 (Improper Control of Generation of Code)
2	CWE-78 (Improper Neutralization of Special Elements used in an OS Command)	CWE-79 (Improper Neutralization of Directives in Dynamically Evaluated Code)
3	CWE-427(Uncontrolled Search Path Element)	CWE-22 (Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal'))
4	CWE-457 (Use of Uninitialized Variable)	CWE-685 (Function Call with Incorrect Number of Arguments)
5	CWE-772 (Missing Release of Resource after Effective Lifetime)	CWE-770 (Allocation of Resources Without Limits or Throttling)

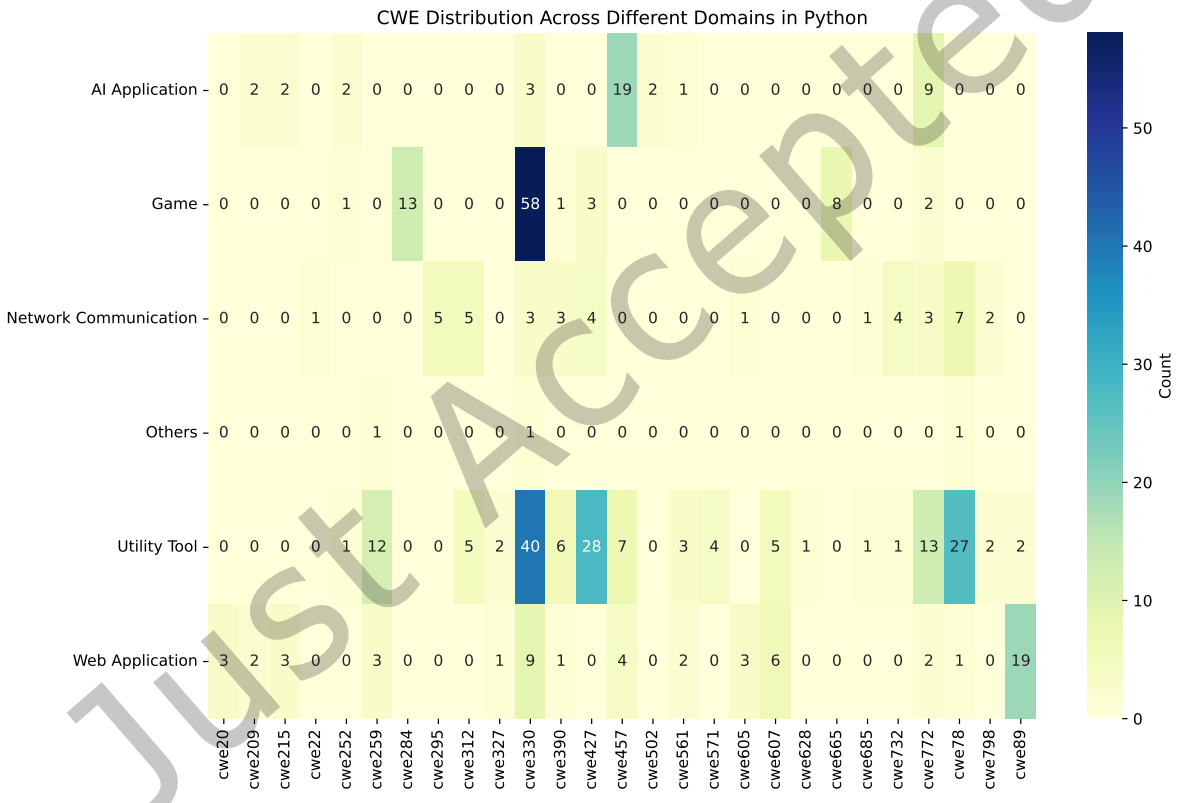


Fig. 14. Distribution of CWEs in Python code snippets across different application domains

used *Copilot Chat* to repair snippets with security weaknesses. The repaired code snippets are then checked for security weaknesses using the same static analysis tools. After getting the analysis results, The first author used the same filtering steps described in Section 3.3.2 to filter the results. We first removed the results that were repetitively reported by two of the tools. Then, we removed the analysis results that were not security

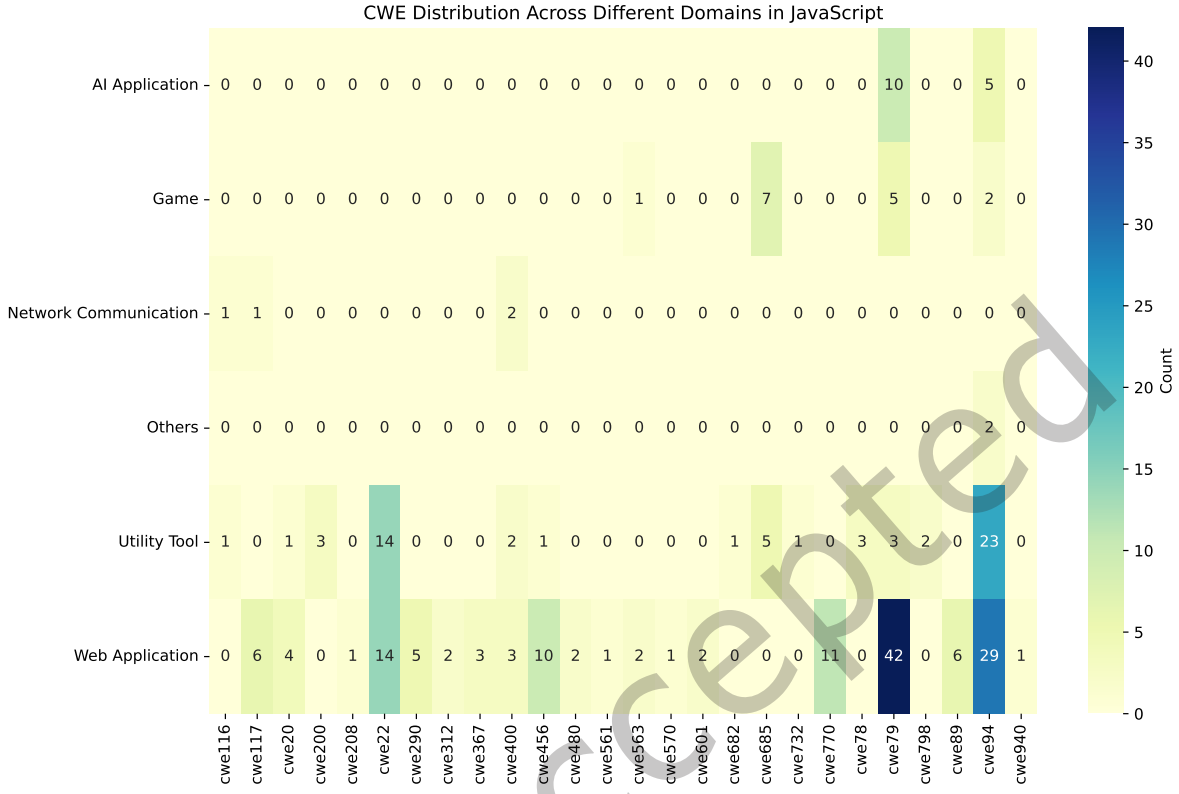


Fig. 15. Distribution of CWEs in JavaScript code snippets across different application domains

weaknesses. Among the three types of warnings from CodeQL analysis, we only counted code snippets that had *warnings* and *errors*, and ignored *recommendations* on code quality. Since the code snippets we selected all come from the *Repo* label, it is easier to filter the results related to security weaknesses, as we did not need to locate the specific line numbers of the code generated by Copilot in each file. Based on the results of the two successive security analysis, we determine whether the security weakness was successfully removed and the code is fixed.

Results. Table 7 shows the proportion of CWEs that were fixed under the three prompts. We can see that using the slash command fix can fix 19.3% of security weaknesses, while our designed basic and enhanced prompts can fix 31.8% and 55.5% of security weaknesses, respectively. *Copilot Chat* demonstrates promising capability in fixing security weaknesses regardless of the prompt used. At the same time, providing enhanced information in the prompts helps improve the proportion of fixes. Figure 16 shows the number of security weaknesses fixed in code snippets of different languages when using various prompts. With basic prompts, security weaknesses were fixed in 29.4% (48 of 163) of Python code and 34.8% (46 of 132) in JavaScript. When using the enhanced prompt, the percentage of security weaknesses fixed in both languages rose to 58.3% (95 of 163) and 51.5% (68 of 132), respectively. We can see that *Copilot Chat*'s ability to fix security weaknesses in Python and JavaScript snippets does not differ significantly.

Furthermore, Figure 17 shows a heatmap with the number of different CWEs that were fixed when using different prompts. We can visually observe that *Copilot Chat*'s effectiveness in fixing different CWEs varies.

Table 7. CWE issue fix percentages by the three prompts

Prompt	Total CWE Before Fix	Total CWE After Fix	Fix Rate (%)
/fix	295	238	19.3%
basic	295	201	31.8%
enhanced	295	132	55.5%

For certain CWEs, *Copilot Chat* can perform well in repairing those snippets and provide a complete fix (i.e., remove the CWE from the code) such as *CWE-259: Use of Hard-coded Password* and *CWE-685: Function Call with Incorrect Number of Arguments*, which can be 100% fixed with the enhanced prompt. However, for some security weaknesses, the repair performance of *Copilot Chat* is considerably poor as it fails to aviod certain CWEs. For example, for *CWE-94: Code Injection*, Copilot Chat's fixes were successful in less than 20% of cases, and Copilot Chat was unable to fix *CWE-78: OS Command Injection*. In addition, we also observe that as the information provided in the prompts increases, the percentages of certain CWEs being fixed improve, such as *CWE-79: Cross-site Scripting* and *CWE-330: Use of Insufficiently Random Values Weakness*. The percentage of successful fixes for *CWE-79: Cross-site Scripting* increases from 0% to 70%, while the percentage of successful fixes for *CWE-330: Use of Insufficiently Random Values Weakness* increases from less than 10% to 98%.

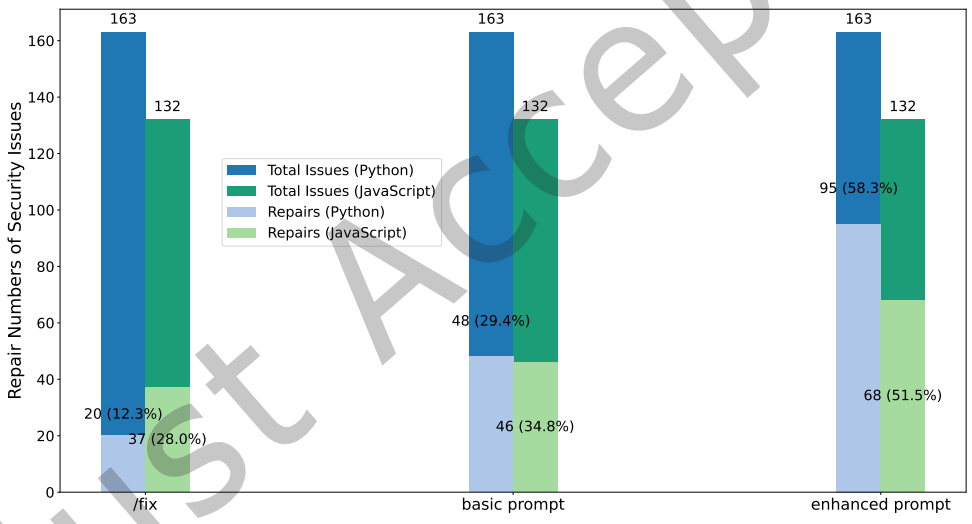


Fig. 16. Fix numbers and percentages for security weaknesses by prompt and language

5 DISCUSSION

5.1 Interpretation of Results

RQ1: How secure is the code generated by Copilot in GitHub projects?
Around quarter of the code snippets in our dataset contained security weaknesses. We found that the code with security weaknesses in Python primarily involves Utility Tools and Game applications, whereas, in JavaScript, Web Application code constitutes the majority of security weaknesses followed by Utility Tools. This disparity

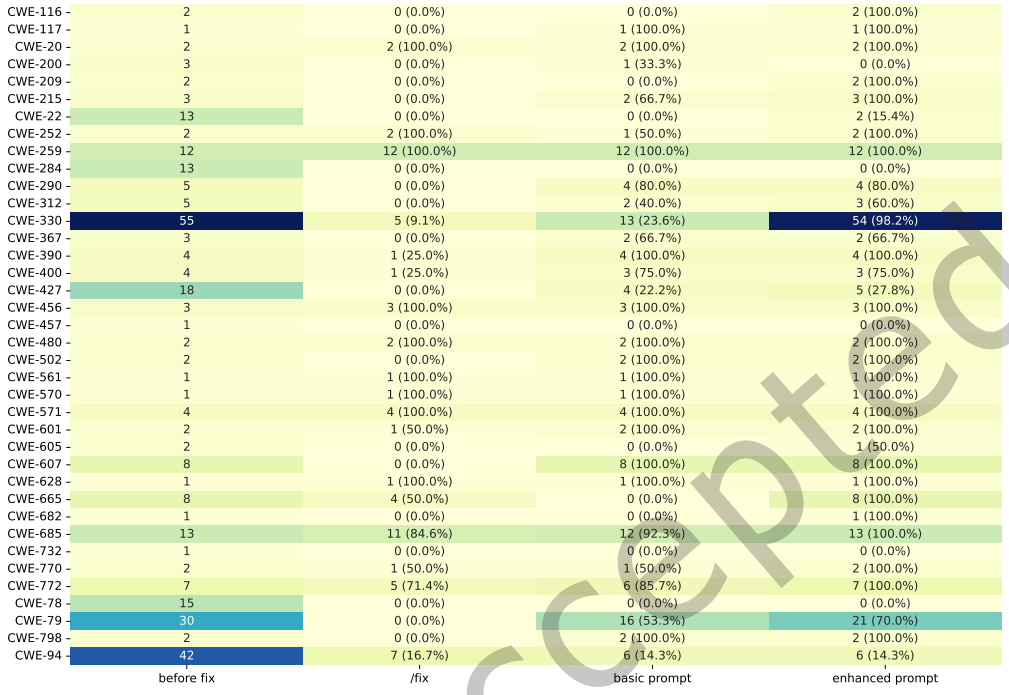


Fig. 17. Distribution of the numbers of fixed CWEs under different prompts

may arise from the two languages' differing purposes and design philosophies [29, 60]. JavaScript is mainly used for front-end development and Web applications [57], leading to its security weaknesses being concentrated in scenarios related to the web and user interaction. In contrast, Python is more broadly applied in fields such as data processing and scientific computing [48, 69], resulting in security weaknesses occurring across a wider range of application domains. Although our results show that the proportion of security weaknesses in Python code snippets is higher than in JavaScript, there is no significant difference between the two languages. Using AI code generation tools to write code in Python or JavaScript can generally lead to security weaknesses. This could be attributed to features that made their code more flexible, such as dynamic typing and interpretation. From a tool point of view, the generated code does not need to reason about the whole program because it is dynamically typed, and consequently, it does not need to read the flow of the whole program to generate working suggestions. In summary, developers should pay careful attention to the potential security weaknesses present in automatically generated code, taking appropriate measures to validate input data and manage resources effectively to minimize security risks. RQ1 results suggest that using Copilot is prone to security issues in production and additional security assessments are required to ensure the generated code does not introduce potential security risks.

RQ2: What security weaknesses are present in the code generated by Copilot across different application domains?

After conducting a security analysis of 733 code snippets generated by Copilot and other tools, a total of 628 security weaknesses were identified, involving 43 CWEs, which is around 10% of the CWEs (439 CWEs) [74]. This may be because Copilot and the other two tools generate code in different programming languages and application scenarios. In addition, since the base models of AI code generation tools (such as Codex) are trained on publicly available data that potentially contain various types of security weaknesses, this can lead to the presence of multiple CWEs in the generated code. This set of 43 CWEs covers many security weaknesses.

The diversity of security weaknesses indicates that developers using Copilot face various security risks. These risks are diverse, covering different development environments and scenarios. At the same time, eight of the CWEs identified in our dataset can be found in the 2023 CWE Top-25 list, covering more than 200 security weaknesses (37.1% of 628 identified). This indicates that the commonly acknowledged Top-25 CWEs, which are considered the most prevalent and dangerous weaknesses, are also prevalent in the AI-generated code. Therefore, developers using Copilot must pay close attention to these weaknesses and take appropriate measures to prevent them before they are integrated into their code base. Meanwhile, we observed that some vulnerabilities from the CWE Top-25 list were not detected in our analysis, indicating that Copilot may sanitize and prevent specific weaknesses from being suggested to developers.

We also identified 30 security weaknesses in the generated code that do not belong to the CWE Top-25 list. Although these are less common security weaknesses and may not be as widespread as CWE Top-25, attackers can still exploit them. For example, we only detected one instance of *CWE-732: Incorrect Permission Assignment for Critical Resource* in our dataset. This security weakness is not commonly found in code and only occurs when specific users have certain permissions. However, it can lead to significant security risks when it does occur.

Besides, we observed that the types of CWEs in generated code are closely related to the application domains. Code snippets used in Game development are prone to issues such as *CWE-330: Use of Insufficiently Random Values Weakness* and *CWE-284: Improper Access Control*. This may be due to the complex logic and player inputs typically involved in game applications, which can lead to security problems related to memory management and input validation. In the Utility Tool category, the proportion of security weaknesses is relatively high, which may be attributed to the diversity of user inputs. Furthermore, we found that Python code snippets often exhibit security weaknesses such as *CWE-330: Use of Insufficiently Random Values Weakness*, *CWE-427: Uncontrolled Search Path Element*, and *CWE-78: OS Command Injection*, whereas JavaScript code snippets primarily encounter issues like *CWE-22: Path Traversal* and *CWE-94: Code Injection*. This may be related to the differences in the application characteristics of the two languages: JavaScript, particularly through environments like Node.js, is use for file operations and dynamic code execution [7], while Python tends to focus more on system-level operations and automation scripts. In addition, the measures we need to take are different: Utility Tools in JavaScript need to emphasize path validation and the security of dynamic execution to prevent attacks such as path traversal and code injection, while those in Python should focus on ensuring safe library loading paths and validating user input for command execution to prevent system-level vulnerabilities caused by malicious input. In Web development, Python is commonly used for frequent interactions between the back-end and the database [69], which makes it prone to *CWE-89: SQL Injection*. *CWE-89: SQL Injection* typically occurs when user input is directly embedded into SQL query statements without parameterized queries or input validation. On the other hand, JavaScript, as a front-end language that interacts directly with users, often faces security weaknesses such as *CWE-79: Cross-site Scripting*, *CWE-94: Code Injection*, and *CWE-22: Path Traversal*. This may be attributed to the frequent interactions between users and data in Web development. Additionally, the complexity of dependencies and the handling of sensitive data further increase the likelihood of security weaknesses.

Regarding the Network Communication category, the proportion of security weaknesses is relatively lower than other categories. This may be attributed to the fact that code in the Network Communication domain typically

demands a high level of security, benefiting from mature protocols, stringent industry standards. Additionally, the protocols involved in network communication inherently incorporate certain security mechanisms to safeguard the data transmission process, further mitigating common attack risks.

Overall, while Python and JavaScript differ in some common types of security weaknesses, they require developers to be aware of and take timely and targeted security measures to mitigate these risks. RQ2 findings note the security weaknesses that developers may encounter in an actual production environment and their occurrence frequency. These findings can help developers be aware of the security aspects of the code generated by AI code generation tools and take appropriate measures to address the security weaknesses in an informed manner.

RQ3: Can *Copilot Chat* help fix the security weaknesses found in the code generated by Copilot?
As shown in Table 7, we can see that regardless of the prompts used, *Copilot Chat* can fix a certain proportion of security weaknesses in the code we analyzed. Even the slash command `/fix` can resolve nearly 20% of security problems, indicating that *Copilot Chat* has some capability to fix certain weaknesses. Copilot Chat is an enhanced feature that extends the capabilities of the original Copilot [23]. In fact, the security of Copilot and its underlying model (Codex) is continually improving [25]. However, it has also been found that Copilot's security layer can potentially handle some CWEs better than others, leaving applications vulnerable to some critical vulnerabilities [43]. It is believed that similar programming-based chatbots, such as Amazon Q (which works with CodeWhisperer), also offer similar repair capabilities. Furthermore, we can observe that using prompts with richer information helps improve the repair effectiveness, which guides us to leverage static analysis tools in combination with *Copilot Chat* to enhance the security of automatically generated code. When using the fix command, security weaknesses in JavaScript code are resolved more frequently than those in Python, possibly because JavaScript code is easier to trace in context, allowing *Copilot Chat* to more readily identify and fix these issues. In contrast, Python often involves data structures, libraries, and system calls, which may make security weakness resolution less intuitive compared to JavaScript. However, when enhanced information is provided, the repair rates for security weaknesses in Python and JavaScript code segments do not differ significantly.

From the heatmap in Figure 17, we can see that the effectiveness of *Copilot Chat* in fixing different CWEs varies. *Copilot Chat* is particularly effective at fixing *CWE-259: Use of Hard-coded Password* and *CWE-685: Function Call with Incorrect Number of Arguments*, which pertain to access control issues and environment configuration. These issues are relatively straightforward and have standard repair methods, making them easier for automated tools to identify and fix. On the other hand, other weaknesses such as *CWE-78: command injection* and *CWE-94: code injection* involve executing commands or code provided by users, which typically require better complex contextual understanding and detailed input validation. Due to the complexity of these issues, *Copilot Chat* may encounter difficulties in automatically providing a fix. Additionally, we found that using more detailed instructions significantly improves repair outcomes for certain CWEs, such as *CWE-330: Use of Insufficiently Random Values Weakness* and *CWE-79: Cross-site Scripting*. This may be because our designed prompts provide richer information and context (the warning messages from static analysis tools), reducing ambiguity. This allows Copilot to effectively identify issues and generate correct fixes.

5.2 Implications

5.2.1 Importance of continuous security analysis of automatically-generated code. We conjecture that practitioners using Copilot and other AI code generation tools are likely to encounter security weaknesses in their generated code, regardless of the programming language used. Our study results reflect the inevitability of security weaknesses in automatically generated code. Practitioners must be aware of the diverse security scenarios in

production and adopt multiple security prevention measures to address security risks before accepting vulnerable code suggestions.

Some CWEs, such as *CWE-78: OS Command Injection* and *CWE-94: Code Injection*, involve direct control of code execution by user inputs and may lead to severe security risks. Developers should be particularly vigilant about these high-risk operations and manually review these parts of the code. Input validation is the foundation of code security. Many security weaknesses (such as *CWE-22: Path Traversal*) arise from inadequate input validation. When using AI code generation tools, developers should avoid directly using user input as the basis for code execution or file path operations and manually add input validation or adopt a more secure coding approach. Moreover, Copilot performs relatively well in handling permission control (such as *CWE-259: Use of Hard-coded Password*) and environment configuration issues. Nevertheless, developers should be familiar with best practices related to permissions and verify that the configurations generated by AI code generation tools meet the project's security requirements.

Maintaining a rigorous process of security analysis concurrently with code generation can help identify potential vulnerability issues and rectify the security weaknesses in time. Developers should follow the best practices for using code generation tools and always check the code suggestions generated by Copilot (or any code generation tools). For example, developers can establish a gated check-in build process to check and prevent security weaknesses when committing code generated by AI code generation tools. Initially, we can turn to automated tools to continuously scan the Copilot-generated code for known security weaknesses, such as CWEs. Then, *Copilot Chat* or other LLMs can be used to fix identified code with security weaknesses, followed by a code re-analysis with security analysis tools. At the same time, the severity of security weaknesses and the limitations of static analysis tools remind us that it is essential to conduct a subsequent manual assessment, including manual security code review for auto-generated code. By embracing this combined strategy for continuous security analysis, we can ensure a robust security shield for the code-committing process with AI-generated code.

5.2.2 Prevention of security weaknesses in AI-generated code. We offer the following suggestions on how to prevent potential security weaknesses in generated code:

Targeted security countermeasures: Based on the frequency of related CWEs in our dataset, practitioners can proactively prevent and address security weaknesses in a targeted manner. Developers should focus on dedicated CWEs in AI-generated code of different programming languages when using Copilot to generate e.g., Python or JavaScript code. Furthermore, a recent study shows that security weaknesses appear in certain code suggestions, but not all [43]; consequently, developers should carefully select potentially more secure suggestions, with the assistance of tools, that do not expose the code to vulnerabilities. Security weaknesses can vary across programming languages and application domains, and developers should pay special attention to these differences when using these automated code generation tools. For example, the security weaknesses path validation (such as *CWE-22: Path Traversal*) and dynamic execution (such as *CWE-94: Code Injection*) are more likely to occur in JavaScript code. At the same time, system calls and library loading in Python code are more likely to raise security concerns.

Standardized security assessment: Common security weaknesses in software development are also prevalent in the code generated by AI code generation tools. As a good practice, developers can use the CWE Top-25 list as a guide to understand which security weaknesses are most common and dangerous in the generated code. Additionally, the CWE Top-25 provides a standardized approach for security assessment, and developers can also use it as a guide to perform security audits of the AI-generated code. Using a standardized remediation strategy for common security weaknesses (such as privilege control and environment configuration) can effectively reduce the likelihood of generating insecure code. Developers can refer to the fixes provided in the static analysis tool documentation [8] or the mitigation measures for related CWEs provided by the MITRE [46].

Enhancing prompt engineering to generate more secure code: The instruction tuning schemes in code generation not only impact the utility of the code but also its security. By combining the security fine-tuning with standard instruction tuning, joint optimization of security and utility can be promoted [32]. It is recommended to incorporate security considerations from the initial stages of code generation. For specific security scenarios prone to CWEs, we can improve code security by enhancing prompt engineering, e.g., with the prompt patterns proposed in [79].

Distinguishing the complexity of security weaknesses: Some security issues, such as access control and environment configuration, have standard solutions and are well suited for quick fixes using tools like *Copilot Chat*. However, for complex logic controls (such as logic processing related to multi-layered inputs), *Copilot Chat*'s repair capabilities may be limited, requiring developers' review and design. When using *Copilot Chat*, developers should assess the complexity of issues to determine which code can be automatically fixed and which requires manual intervention.

Static analysis tools combined with LLM-based chatbots for fixing security weaknesses: When using *Copilot Chat* to address security weaknesses, developers should provide more specific context and clearer instructions. More detailed information can significantly enhance *Copilot Chat*'s ability to fix security problems. Meanwhile, integrating static analysis tools allows for a more comprehensive identification of potential risks in the code. Developers can leverage static analysis tools to identify specific security risks and provide detailed instructions to Copilot Chat, enabling it to effectively fix the issues in Copilot-generated code.

6 THREATS TO VALIDITY

The validity threats are discussed according to the guidelines in [62]. Note that we did not consider internal validity threats since we did not investigate any relationships between variables and results.

Construct Validity: This study has three threats to construct validity: *(1) Using keyword-based search* – We used a keyword-based search to collect relevant code snippets from GitHub. The results obtained through the keyword-based search may not cover all the code snippets generated by Copilot and other tools on GitHub. We tried to mitigate this threat by constantly and iteratively refining the keywords and using synonyms. *(2) Manual data filtering* – We manually screened the results obtained from the keyword-based search by analyzing the comments, tags, and other metadata of the code snippets to determine whether they were generated by Copilot or other tools. Since this process was manually done, it may have been influenced by personal bias. In this regard, two authors conducted the experiment independently to minimize the impact on the construct validity. *(3) Manual association of CWEs* – We manually associated the warning messages reported by the static analysis tools with a particular CWE, which may introduce personal subjective bias, threatening the construct validity. We employed two measures to mitigate this threat. First, since the list of CWEs is a tree structure with interconnections between them, we first matched the warning messages to a higher-level CWE and further checked whether we can match the warning messages to a lower-level CWE with a more specific definition. We refer to the description of the test suite and CWE coverage in the static analysis tool documentation. Second, to mitigate the personal bias, two authors independently assigned each security weakness description a CWE ID within a period of ten days. In case of disagreement, the two authors discussed it with the assessment by a third author (a security expert).

External Validity: Our dataset consists of the code snippets generated by Copilot and other code generation tools collected from open-source projects on GitHub. During the filtering process, we excluded code that utilized AI code generation tools to solve simple programming practice problems, aiming to ensure that the collected data genuinely reflected open source development on GitHub. Since the data from GitHub are not diversified enough, we had a higher number of code snippets originating from the Game projects. This could result in a lack of comprehensiveness in the security scenarios involved. The peculiarity of the data source may make the dataset incomplete, thereby threatening the external validity of the results. Furthermore, we acknowledge the need to

collect more diverse code snippets from different platforms to increase the generalizability of the results. We will consider adopting more diversified ways or platforms to collect code.Additionally, due to the limitations of static analysis tools themselves, these tools could not scan all CWEs, and there is a degree of false positives in the analysis results (as the case with static analysis, in general, [35, 72]). Although we used two widely used static analysis tools to increase the comprehensiveness of the scans and manually checked the results of the tool scans, the results may suffer from incompleteness.

Reliability: We used multiple automated static analysis tools to analyze the Copilot-generated code snippets to improve security weaknesses detection. Developers have widely used these automated tools. The querying mechanism of these tools ensures that the analysis results remain consistent when used multiple times. In addition, we performed two rounds of analyzing with two tools for security checks on each code snippet, intending to complement the results of one tool with the other. By implementing these measures, we believe that our research results are reliable and these threats to reliability are mitigated.

7 CONCLUSIONS

Automatic code generation and recommendation have been an active research area due to the advancement of AI and, specifically, LLMs. AI code generation tools, such as Copilot, can significantly improve developers' development efficiency but can also introduce vulnerabilities and security risks. Existing studies on the security of AI-generated code mainly focused on the security issues in the generated code using crafted scenarios, and potential security weaknesses of AI-generated code in practical scenarios in open source development environment (e.g., GitHub) have not been fully considered. In this paper, we present the results of an empirical study to analyze security weaknesses in Copilot-generated code found in public GitHub projects. We identified 733 code snippets generated by Copilot and other tools (i.e., CodeWhisperer and Codeium) from GitHub projects and analyzed those snippets for security weaknesses using static analysis tools. We also examined using *Copilot Chat* to fix security issues in Copilot-generated code. This study aims to help developers understand the security risks of weaknesses introduced in the code generated by Copilot (and potentially similar code generation tools). Our results show that (1) around 30% of the 733 generated code snippets contain security weaknesses. Developers have a high risk of raising security weaknesses when using Copilot or other AI code generation tools, regardless of the programming language, and therefore, applying appropriate security checks is necessary. (2) The detected security weaknesses are diverse in nature and are associated with 43 different CWEs. Developers encounter a variety of development scenarios and production environments, requiring appropriate security awareness and skills. (3) Among these CWEs, eight appear in the 2023 CWE Top-25 list, and six belong to the Stubborn Weaknesses, indicating high severity. (4) *Copilot Chat* can help fix many of the security issues in Copilot-generated code, but its success rate varies across different CWEs. (5) Providing *Copilot Chat* with a warning message from the static analysis tool leads to a better fix.

In the future, we plan to: (1) collect additional code snippets from other open source repositories and industrial projects and code snippets generated by newer releases of Copilot; (2) analyze and summarize the application scenarios of these code snippets, studying how practitioners use Copilot and fix the issues in development; and (3) compare the results with other emerging Generative AI code generation tools such as CodeWhisperer, aiXcoder, and Code Llama.

ACKNOWLEDGMENTS

This work is funded by the National Natural Science Foundation of China (NSFC) under Grant No. 62172311 and the Major Science and Technology Project of Hubei Province under Grant No. 2024BAA008.

REFERENCES

[1] Owura Asare, Meiyappan Nagappan, and N. Asokan. 2023. Is GitHub's Copilot as Bad as Humans at Introducing Vulnerabilities in Code? *Empirical Software Engineering* 28, 6 (2023), Article No. 129. https://doi.org/10.1007/s10664-023-10380-1

[2] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 85–111. https://doi.org/10.1145/3586030

[3] Brett A Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2023. Programming is hard-or at least it used to be: Educational opportunities and challenges of ai code generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 500–506. https://doi.org/10.1145/3545945.3569759

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Proceedings of the 34th Annual Conference on Neural Information Processing Systems (NeurIPS)*. 1877–1901.

[5] John L Campbell, Charles Quincy, Jordan Osserman, and Ove K Pedersen. 2013. Coding in-depth semistructured interviews: Problems of unitization and intercoder reliability and agreement. *Sociological Methods & Research* 42, 3 (2013), 294–320. https://doi.org/10.1177/0049124113500475

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021). https://doi.org/10.48550/arXiv.2107.03374

[7] Nimesh Chhetri. 2016. A comparative analysis of node. js (server-side javascript). *Culminating Projects in Computer Science and Information Technology* 5 (2016), 1–69.

[8] CodeQL. 2024. *CodeQL Query Help.* https://codeql.github.com/codeql-query-help/

[9] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46. https://doi.org/10.1177/001316446002000104

[10] Copilot. 2024. *best-practices-for-using-github-copilot.* https://docs.github.com/zh/copilot/using-github-copilot/best-practices-for-using-github-copilot

[11] Valerio Cosentino, Javier L Cánovas Izquierdo, and Jordi Cabot. 2017. A systematic mapping study of software development with GitHub. *IEEE Access* 5 (2017), 7173–7192. https://doi.org/10.1109/ACCESS.2017.2682323

[12] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Zhen Ming Jack Jiang. 2023. Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software* 203 (2023), 111734. https://doi.org/10.1016/j.jss.2023.111734

[13] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70. https://doi.org/10.1145/3338112

[14] Lisa Nguyen Quang Do, James R Wright, and Karim Ali. 2020. Why do software developers use static analysis tools? a user-centered study of developer needs and motivations. *IEEE Transactions on Software Engineering* 48, 3 (2020), 835–847. https://doi.org/10.1109/TSE.2020.3004525

[15] Alex Dow. 2021. *GitHub's Copilot Security Issues.* https://miraisecurity.com/blog/githubs-copilot-security-issues

[16] Trevor Dunlap, Seaver Thorn, William Enck, and Bradley Reaves. 2023. Finding Fixed Vulnerabilities with Off-the-Shelf Static Analysis. In *Proceedings of the 8th IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 489–505. https://doi.org/10.1109/EuroSP57164.2023.00036

[17] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. 2008. Selecting Empirical Methods for Software Engineering Research. Springer, 285–311. https://doi.org/10.1007/978-1-84800-044-5_11

[18] Ran Elgedawy, John Sadik, Senjuti Dutta, Anuj Gautam, Konstantinos Georgiou, Farzin Gholamrezae, Fujiao Ji, Kyungchan Lim, Qian Liu, and Scott Ruoti. 2024. Ocassionally Secure: A Comparative Analysis of Code Generation Assistants. *arXiv preprint arXiv:2402.00689* (2024). https://doi.org/10.48550/arXiv.2402.00689

[19] Michael D Ernst. 2003. Static and dynamic analysis: Synergy and duality. In *Proceedings of the ICSE Workshop on Dynamic Analysis (WODA)*. ACM, 24–27.

[20] ESLint Contributors. 2024. *ESLint - Pluggable JavaScript linter.* GitHub. https://eslint.org/

[21] Yujia Fu, Peng Liang, Amjed Tahir, Zengyang Li, Mojtaba Shahin, Jiaxin Yu, and Jinfu Chen. 2024. *Dataset of the Paper "Security Weaknesses of Copilot-Generated Code in GitHub Projects: An Empirical Study".* https://doi.org/10.5281/zenodo.10802054

[22] GitHub. 2023. *CodeQL* (1.6 ed.). GitHub. https://securitylab.github.com/tools/codeql

[23] GitHub. 2023. *GitHub Copilot Chat Beta Now Available for Every Organization.* https://github.blog/news-insights/product-news/github-copilot-chat-beta-now-available-for-every-organization/

[24] GitHub. 2023. *GitHub Copilot for Individuals.* https://docs.github.com/en/copilot/overview-of-github-copilot/about-github-copilot-for-individuals

[25] GitHub. 2023. *GitHub CopilotX Preview.* https://github.com/features/preview/copilot-x

[26] GitHub. 2023. *Using the CodeQL CLI.* GitHub. https://docs.github.com/zh/code-security/codeql-cli/using-the-codeql-cli/analyzing-databases-with-the-codeql-cli
[27] GitHub. 2024. *GitHub Copilot Chat - Visual Studio Marketplace.* https://marketplace.visualstudio.com/items?itemName=GitHub.copilot-chat
[28] GitHub. 2024. *Octoverse 2024 Top Programming Languages.* https://github.blog/news-insights/octoverse/octoverse-2024/#the-most-popular-programming-languages
[29] Halfnine. 2024. *JavaScript vs Python: Breaking Down Performance, Ease of Use, and More.* https://www.halfnine.com/blog/post/javascript-vs-python
[30] William Harding and Matthew Kloster. 2024. *Coding on Copilot: 2023 Data Suggests Downward Pressure on Code Quality.* https://www.gitclear.com/coding_on_copilot_data_shows_ais_downward_pressure_on_code_quality
[31] Jingxuan He and Martin Vechev. 2023. Controlling Large Language Models to Generate Secure and Vulnerable Code. *arXiv preprint arXiv:2302.05319* (2023). https://doi.org/10.48550/arXiv.2302.05319
[32] Jingxuan He, Mark Vero, Gabriela Krasnopolska, and Martin Vechev. 2024. Instruction Tuning for Secure Code Generation. *arXiv preprint arXiv:2402.09497* (2024). https://doi.org/10.48550/arXiv.2402.09497
[33] Michael Huth and Flemming Nielson. 2019. Static analysis for proactive security. , 374–392 pages. https://doi.org/10.1007/978-3-319-91908-9_19
[34] Emanuele Iannone, Roberta Guadagni, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. 2022. The secret life of software vulnerabilities: A large-scale empirical study. *IEEE Transactions on Software Engineering* 49, 1 (2022), 44–63. https://doi.org/10.1109/TSE.2022.3140868
[35] Hong Jin Kang, Khai Loong Aw, and David Lo. 2022. Detecting false alarms from automatic static analysis tools: How far are we?. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. ACM, 698–709. https://doi.org/10.1145/3510003.3510214
[36] Arvinder Kaur and Ruchikaa Nayyar. 2020. A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code. *Procedia Computer Science* 171 (2020), 2023–2029. https://doi.org/10.1016/j.procs.2020.04.217
[37] Raphaël Khoury, Anderson R Avila, Jacob Brunelle, and Baba Mamadou Camara. 2023. How secure is code generated by chatgpt?. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE, 2445–2451. https://doi.org/10.1109/SMC53992.2023.10394237
[38] Paul Krill. 2024. *GitHub Copilot makes insecure code even less secure, Snyk says.* https://www.infoworld.com/article/3713141/github-copilot-makes-insecure-code-even-less-secure-snyk-says.amp.html
[39] Sila Lertbanjongngam, Bodin Chinthanet, Takashi Ishio, Raula Gaikovina Kula, Pattara Leelaprute, Bundit Manaskasemsak, Arnon Rungsawang, and Kenichi Matsumoto. 2022. An Empirical Evaluation of Competitive Programming AI: A Case Study of AlphaCode. In *Proceedings of the 16th IEEE International Workshop on Software Clones (IWSC)*. IEEE, 10–15. https://doi.org/10.48550/arXiv.2208.08603
[40] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097. https://doi.org/10.1126/science.abq1158
[41] Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Dong Chen, Shuai Wang, and Cuiyun Gao. 2023. Cctest: Testing and repairing code completion systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1238–1250. https://doi.org/10.1109/ICSE48619.2023.00110
[42] Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D Le, and David Lo. 2024. Refining chatgpt-generated code: Characterizing and mitigating code quality issues. *ACM Transactions on Software Engineering and Methodology* 33, 5 (2024), 1–26. https://doi.org/10.1145/3643674
[43] Vahid Majdinasab, Michael Joshua Bishop, Shawn Rasheed, Arghavan Moradidakhel, Amjed Tahir, and Foutse Khomh. 2024. Assessing the Security of GitHub Copilot Generated Code - A Targeted Replication Study. In *Proceedings of the 31st IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 435–444. https://doi.org/10.1109/SANER60148.2024.00051
[44] Dwayne McDaniel. 2021. *GitHub Copilot: Security and Privacy.* https://blog.gitguardian.com/github-copilot-security-and-privacy/
[45] MITRE. 2023. *CWE Top 25 Most Dangerous Software Errors 2023.* https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html
[46] MITRE. 2024. *Common Weakness Enumeration (CWE) Data.* https://cwe.mitre.org/data/index.html
[47] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2024. Reading between the lines: Modeling user behavior and costs in AI-assisted programming. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–16. https://doi.org/10.1145/3613904.3641936
[48] Abhinav Nagpal and Goldie Gabrani. 2019. Python for data analytics, scientific and technical applications. In *Proceedings of the Amity international conference on artificial intelligence (AICAI)*. IEEE, 140–145. https://doi.org/10.1109/AICAI.2019.8701341
[49] Roberto Natella, Pietro Liguori, Cristina Improta, Bojan Cukic, and Domenico Cotroneo. 2024. AI Code Generators for Security: Friend or Foe? *IEEE Security & Privacy* 22, 5 (2024), 73–81. https://doi.org/10.1109/MSEC.2024.3355713
[50] Nhan Nguyen and Sarah Nadi. 2022. An empirical evaluation of GitHub copilot's code suggestions. In *Proceedings of the 19th IEEE/ACM International Conference on Mining Software Repositories (MSR)*. IEEE, 1–5. https://doi.org/10.1145/3524842.3528470

[51] Joao Pedro Nunes, Alex S Ribeiro, Brad J Schoenfeld, and Edilson S Cyrino. 2018. Comment on: "Comparison of periodized and non-periodized resistance training on maximal strength: A meta-analysis". *Sports Medicine* 48 (2018), 491–494. https://doi.org/10.1007/s40279-017-0824-x

[52] OpenAI. 2021. *Introducing Codex: The AI Behind GitHub Copilot.* https://openai.com/blog/openai-codex

[53] Stack Overflow. 2024. *Stack Overflow Developer Survey 2024: Most Popular Technologies (AI, Search, Dev).* https://survey.stackoverflow.co/2024/technology#most-popular-technologies-ai-search-dev

[54] OWASP. 2024. *Source Code Analysis Tools.* https://owasp.org/www-community/Source_Code_Analysis_Tools

[55] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768. https://doi.org/10.1109/SP46214.2022.9833571

[56] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2023. Do users write more insecure code with AI assistants?. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2785–2799. https://doi.org/10.1145/3576915.3623157

[57] PubNub. 2024. *JavaScript: The Complete Guide for Front-End Development.* https://www.pubnub.com/guides/javascript/

[58] Rohith Pudari and Neil A Ernst. 2023. From Copilot to Pilot: Towards AI Supported Software Development. *arXiv preprint arXiv:2303.04142* (2023). https://doi.org/10.48550/arXiv.2303.04142

[59] PyCQA. 2024. *Bandit: A Security Linter for Python.* https://github.com/PyCQA/bandit

[60] Meghan Reichenbach. 2022. *Python vs. JavaScript: Which Should You Learn?* https://blog.boot.dev/python/python-vs-javascript/

[61] Md Omar Faruk Rokon, Risul Islam, Ahmad Darki, Evangelos E Papalexakis, and Michalis Faloutsos. 2020. SourceFinder: Finding Malware Source-Code from Publicly Available Repositories in GitHub. In *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. USENIX, 149–163.

[62] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14 (2009), 131–164. https://doi.org/10.1007/s10664-008-9102-8

[63] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at google. *Commun. ACM* 61, 4 (2018), 58–66. https://doi.org/10.1145/3188720

[64] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2022. Security Implications of Large Language Model Code Assistants: A User Study. *arXiv preprint arXiv:2208.09727* (2022). https://doi.org/10.48550/arXiv.2208.09727

[65] Advait Sarkar, Andrew D Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, and Ben Zorn. 2022. What is it like to program with artificial intelligence? *arXiv preprint arXiv:2208.06213* (2022). https://doi.org/10.48550/arXiv.2208.06213

[66] Xinyu She, Yue Liu, Yanjie Zhao, Yiling He, Li Li, Chakkrit Tantithamthavorn, Zhan Qin, and Haoyu Wang. 2023. Pitfalls in Language Models for Code Intelligence: A Taxonomy and Survey. *arXiv preprint arXiv:2310.17903* (2023). https://doi.org/10.48550/arXiv.2310.17903

[67] Mohammed Latif Siddiq, Shafayat H Majumder, Maisha R Mim, Sourov Jajodia, and Joanna CS Santos. 2022. An Empirical Study of Code Smells in Transformer-based Code Generation Techniques. In *Proceedings of the 22nd IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 71–82. https://doi.org/10.1109/SCAM55253.2022.00014

[68] Mohammed Latif Siddiq and Joanna CS Santos. 2022. SecurityEval dataset: mining vulnerability examples to evaluate machine learning-based code generation techniques. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security (MSR4P&S)*. ACM, 29–33. https://doi.org/10.1145/3549035.3561184

[69] Simplilearn. 2024. *Applications of Python (Explained with Examples).* https://www.simplilearn.com/what-is-python-used-for-article

[70] Snyk. 2024. *Snyk Code: Secure Your Code as You Develop.* https://snyk.io/product/snyk-code

[71] Dominik Sobania, Martin Briesch, and Franz Rothlauf. 2022. Choose your programming copilot: a comparison of the program synthesis performance of github copilot and genetic programming. In *Proceedings of the 24th Annual Conference on Genetic and Evolutionary Computation Conference (GECCO)*. ACM, 1019–1027. https://doi.org/10.1145/3512290.3528700

[72] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. 2020. On the recall of static call graph construction in practice. In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, 1049–1060. https://doi.org/10.1145/3377811.3380441

[73] Amjed Tahir and Stephen G MacDonell. 2012. A systematic mapping study on dynamic metrics and software quality. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 326–335. https://doi.org/10.1109/ICSM.2012.6405289

[74] The MITRE Corporation. 2023. *CWE VIEW: Software Development.* https://cwe.mitre.org/data/definitions/699.html

[75] Kristín Fjóla Tómasdóttir, Mauricio Aniche, and Arie Van Deursen. 2018. The adoption of javascript linters in practice: A case study on eslint. *IEEE Transactions on Software Engineering* 46, 8 (2018), 863–891. https://doi.org/10.1109/TSE.2018.2871058

[76] Catherine Tony, Markus Mutas, Nicolás E Díaz Ferreyra, and Riccardo Scandariato. 2023. Llmseceval: A dataset of natural language prompts for security evaluations. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 588–592. https://doi.org/10.1109/MSR59073.2023.00084

[77] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Proceedings of the 40th ACM Conference on Human Factors in Computing Systems (CHI)*. ACM, 1–7. https://doi.org/10.1145/3491101.3519665

[78] Richard J Waldinger and Richard CT Lee. 1969. PROW: A step toward automatic program writing. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence (IJCAI)*. ACM, 241–252.

[79] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382* (2023). https://doi.org/10.48550/arXiv.2302.11382

[80] Dakota Wong, Austin Kothig, and Patrick Lam. 2022. Exploring the Verifiability of Code Generated by GitHub Copilot. *arXiv preprint arXiv:2209.01766* (2022). https://doi.org/10.48550/arXiv.2209.01766

[81] Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. 2023. Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. *arXiv preprint arXiv:2304.10778* (2023). https://doi.org/10.48550/arXiv.2304.10778

[82] Burak Yetistiren, Isik Ozsoy, and Eray Tuzun. 2022. Assessing the quality of GitHub copilot's code generation. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*. ACM, 62–71. https://doi.org/10.1145/3558489.355907

[83] Xiao Yu, Lei Liu, Xing Hu, Jacky Wai Keung, Jin Liu, and Xin Xia. 2024. Where Are Large Language Models for Code Generation on GitHub? *arXiv preprint arXiv:2406.19544* (2024). https://doi.org/10.48550/arXiv.2406.19544

[84] Beiqi Zhang, Peng Liang, Xiyu Zhou, Aakash Ahmad, and Muhammad Waseem. 2023. Demystifying Practices, Challenges and Expected Features of Using GitHub Copilot. *International Journal of Software Engineering and Knowledge Engineering* 33, 11&12 (2023), 1653–1672. https://doi.org/10.1142/S0218194023410048