Security Weaknesses of Copilot Generated Code in GitHub

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

Modern code generation tools, utilizing AI models like Large Language Models (LLMs), have gained popularity for producing functional code. However, their usage presents security challenges, often resulting in insecure code merging into the code base. 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 real-world scenarios. To address this gap, we conducted an empirical study, analyzing code snippets generated by GitHub Copilot from GitHub projects. Our analysis identified 452 snippets generated by Copilot, revealing a high likelihood of security issues, with 32.8% of Python and 24.5% of JavaScript snippets affected. These issues span 38 different Common Weakness Enumeration (CWE) categories, including significant ones like *CWE-330: Use of Insufficiently Random Values*, *CWE-78: OS Command Injection*, and *CWE-94: Improper Control of Generation of Code*. Notably, eight CWEs are among the 2023 CWE Top-25, highlighting their severity. Our findings confirm that developers should be careful when adding code generated by Copilot and should also run appropriate security checks as they accept the suggested code. It also shows that practitioners should cultivate corresponding security awareness and skills.

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,

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

etc.), or a combination of text and code [44]. After writing an initial code or comment, developers can rely on code generation tools to 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 [30, 47]. 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 [57]. 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 machine learning and 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 [33]. Currently, LLM-based code generation tools have also been widely applied, such as Codex by OpenAI [35], AlphaCode by DeepMind [29], and CodeWhisperer by Amazon [4].

These models are trained on billions of public open-source lines of code, which includes public code with unsafe coding patterns [22]. Therefore, code generation tools based on such models can pose code quality issues [31], and the code generated by these tools may also suffer from security weaknesses [45]. For example, GitHub Copilot may produce some insecure code, as its underlying Codex model is pre-trained on untrusted data from GitHub [5], which is known to contain buggy programs [41]. According to the developer security company Snyk, GitHub Copilot may also replicate existing security issues in code to suggest insecure code when the user's existing codebase contains security issues [37].

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 [9, 28, 40, 59], and relatively less attention has been paid to security aspects [38, 39, 46]. 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*" [17]. 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 for the code quality generated by Copilot [21]. GitHub also provides tools such as CodeQL to help developers scan for security issues in their code.

To this end, we conducted an empirical study on the security weaknesses of the generated code by GitHub Copilot, which is available on GitHub. We chose Copilot as our research subject because it is a commercial instance of AI-assisted 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 real-world systems [9]. We collected the code generated by Copilot that has been used in projects on GitHub and analyzed the security of the generated code through the lens of a real-world production 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) 29.6% of Copilot-generated code snippets have security weaknesses; (2) the security weaknesses are diverse and related to 38 different CWEs, in which *CWE-330: Use of Insufficiently Random Values*, *CWE-78: OS Command Injection* and *CWE-94: Improper Control of Generation of Code ('Code Injection')* 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.

The contributions of this work: (1) We curated a dataset of code snippets generated by Copilot that has been used in projects on GitHub (a curated data [15] 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.

The rest of this paper is structured as follows: Section 2 presents the 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 RELATED WORK

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

2.1 AI-assisted 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 real-world scenarios. Previous research mainly focused on whether the code generated by these tools can meet users' functional requirements. Yetistiren *et al.* [61] 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.* [48] 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 [34] 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.* [60] evaluated the code quality of AI-assisted code generation tools (GitHub Copilot, Amazon CodeWhisperer, and ChatGPT). 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-assisted code generation tools and how the tools can improve productivity by observing their behavior. Vaithilingam *et al.* [56] studied how programmers 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.* [3] presented the first theoretical analysis of how programmers interact with Copilot based on the observations of 20 participants. Sila *et al.* [28] 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.* [38] 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 [51]). Siddiq *et al.* [46] 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.* [27] 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.* [13] 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.* [43] conducted a security-driven user study, and their results showed that the rate at which AI-assisted 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.* [2] 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.* [55] proposed LLMSecEval, a dataset containing 150 natural language prompts that can be used to evaluate the security performance of LLMs. Siddiq *et al.* [47] provided a dataset, SecurityEval, for testing whether a code generation model has weaknesses. The dataset contains 130 Python code samples.

Different from prior work, we studied the security weaknesses exhibited by code generation tools in a real-world production 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 Pearce *et al.* [38] study, 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 Security Static Analysis

Vulnerabilities detection is critical to improve software security and ensure quality [24]. Vulnerability can be checked either statically or dynamically. Dynamic analysis techniques are more sound and precise but lack coverage [14]. On the other hand, static analysis is less precise but offers greater coverage and allows users to analyze programs without the need to execute them [49]. OWASP [36] provides a list of commonly used static analysis tools for security analysis. This includes tools like CodeQL, a general-purpose automatic scanning tool; FindBugs, a tool for Java programs; ESLint, a tool for JavaScript programs; Bandit, a tool for Python programs; and GoSec, a tool for Go programs. Such tools have been widely used in previous security analysis research [38, 46, 54].

Kaur *et al.* [26] compared static analysis tools for vulnerability detection in scanning C/C++ and Java source code. Tomasdottir *et al.* [54] conducted an empirical study on ESLint, the most commonly used JavaScript static analysis tool among developers. Pearce *et al.* [38] used CodeQL to scan security weaknesses in the generated Python and C++ code. Siddiq *et al.* [47] used Bandit to check Python code generated using a test dataset. Lisa *et al.* [10] 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 scan 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 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.

3.1 Research Goal and Questions

This study aims to understand the potential security weaknesses in Copilot-generated code produced in real-world GitHub projects. We first collect code snippets generated by Copilot from GitHub projects as our data source, which is the largest source of open-source projects. It should be noted that it is not possible to access all the code generated by Copilot in GitHub projects, as there is no direct way to identify if part of a file was generated by 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 chose to focus on Python and JavaScript code snippets as they are currently the two most popular programming languages used by developers according to a recent survey in 2022 by GitHub [1], and also the ones most frequently used with GitHub Copilot [62]. We first analyzed the functionality and application domains of the collected code snippets to get the demographic information of the Copilot-generated code. Next, we performed a security analysis on the identified

code. We selected static analysis because of its coverage and the ability to analyze programs without executing them. Dynamic analysis is more sound and precise (as it can reason about the program behaviour) but also lacks coverage [49]. Using static analysis will allow us to run the analysis on a segment of the program (smaller snippet) without needing the full program to be executed. Static security analysis tools have been widely used by developers and companies [14, 32, 38]. After obtaining the scan results, we manually checked the results to remove false positives reported by the static analysis tools. We finally used CWEs to classify the filtered results for further analysis to answer the RQs.

We conducted this empirical study by following the guidelines of Easterbrook *et al.* [12]. The RQs, their rationale, and the research process of this study (see Figure 1) are detailed in the subsections below.
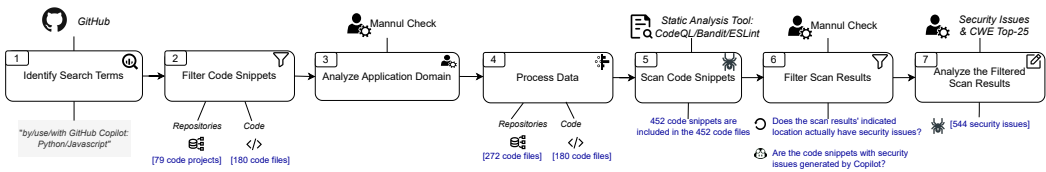

Fig. 1. Overview of the research process

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.

RQ2. What security weaknesses are present in the code snippets generated by Copilot?
Rationale: Copilot-generated code may contain a variety of security weaknesses [38], 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* [17]". 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, allowing them to be vigilant about the generated code before it is integrated into their programs.

RQ3. How many security weaknesses belong to the MITRE CWE Top-25 security weaknesses?
Rationale: The MITRE list contains the Top 25 most dangerous security weaknesses [50], offering a benchmark for gauging the severity of weaknesses in the Copilot-generated code [38]. The answer to RQ3 can help to understand whether the code generated by Copilot contains widely recognized types of security weakness and Copilot's ability to handle these recent and common weaknesses.

3.2 Data Collection and Filtering

We chose GitHub as the primary data source for answering our RQs. GitHub is widely used by developers. 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 [8]. We identify code snippets generated by Copilot from GitHub projects.

3.2.1 Code Snippets Collection. **Step 1.** To identify the code snippets generated by Copilot, we first conducted a pilot search to formulate our search keywords. First, we used "GitHub Copilot" and "Copilot" as our search keywords. As expected, we found that the term "Copilot" is a vague term, referring not only to the code generation tool launched by GitHub, but also to tools in the aviation or telemetry fields. Therefore, using the keyword solely may return irrelevant content that may not be related to the use of the tool. On the other hand, using "GitHub Copilot" can exclude content unrelated to the code generation tool Copilot and narrow the search scope, which is what we have used to locate the code snippets.

However, even with this basic search keyword, we still need to carefully filter the search results to ensure that they are truly related to GitHub Copilot. Although using "GitHub Copilot" increases the relevance of the results to Copilot, these results are not necessarily the code snippets generated by Copilot. It should be noted that many code snippets containing the "GitHub Copilot" keyword in the search results display GitHub Copilot as text. Developers may use them to describe their experience 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. Our target is code generated by Copilot, not code snippets containing the keyword "Copilot".

Our observations from the pilot search showed that using keywords such as "by GitHub Copilot", "use GitHub Copilot", and "with GitHub Copilot" can improve the accuracy of search results. These keywords enable us to focus more on the code generated using Copilot rather than code snippets that contain other content related to Copilot. In addition, since Python and JavaScript are the two most popular programming languages used by developers, and also the most frequently used with GitHub Copilot (see Section 3.1), we further limited the types of code snippets during the search to Python and JavaScript. We collected the search results under the *Code* label. Considering that some projects declare using GitHub Copilot generated code in their README files or project description provided in GitHub, we decided to retain the results from the *Repository* label in the search results. Figure 2 shows an example of our search process.

Table 1 reports the search terms we used and the number of search results obtained from GitHub. In this step, we collected a total of 8,157 results, of which 7,749 were from the *Code* label, and 408 were from the *Repository* label. The same search result may contain multiple keywords, which means that there are duplicate projects in the collected data. After removing duplicate projects, we obtained a total of 2023 search results, of which 1847 were from the *Code* label, and 176 were from the *Repository* label.

Table 1. Search results based on different terms used

#	Search Term	# Code	# Repositories
1	"By GitHub Copilot"	2549	94
2	"Use GitHub Copilot"	1822	123
3	"With GitHub Copilot"	3378	191
Total		**7749**	**408**

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. Additionally, since we aimed to obtain code snippets used in real-world projects, we excluded search results used to solve simple programming practice problems on platforms, such as bisection lookup and fast sorting tasks from LeetCode. We consider these programming exercises to be more closely related to correctness than security.
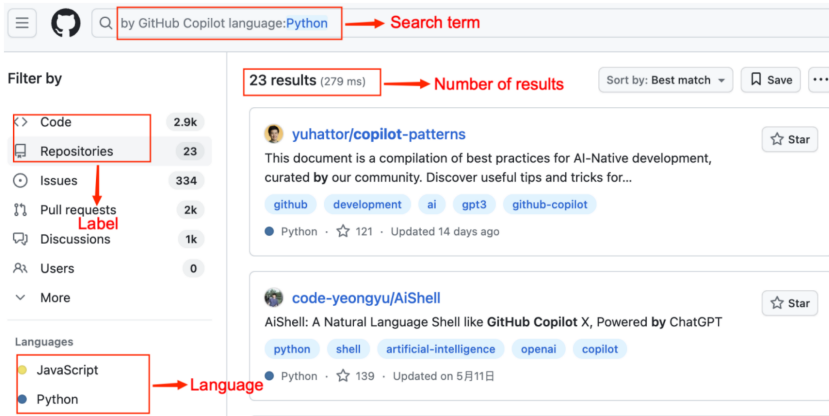
Fig. 2. Example of the search process

We first conducted a pilot data filtration to better filter the search results. 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 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. We provide examples for the three criteria in Figure 3, Figure 4, and Figure 5. As shown in the example in Figure 3 for the *Repository* label, we kept all the Python files. In the next example in Figure 4, we kept the entire file where the Copilot-generated code snippet was located. In Figure 5, the code snippet was removed as it was determined that the code just solved a simple algorithmic problem.

Specifically, the pilot data filtering process consisted of the following steps: (1) The first author randomly selected 40 projects and 400 code files from the search results from the *Repository* label and the *Code* label, respectively. (2) Two authors independently labelled whether Copilot generated these projects and code files. (3) The first author compared the labelling results by the two coders and calculated the level of agreement between them using the Cohen's Kappa coefficient [7]. (4) If there were any results they were unsure of or disagreed with, the two authors discussed until they reached an agreement. The Cohen's Kappa coefficients were 0.79 for the projects (from the *Repository* label) and 0.85 for the code files (from the *Code* label), which were both higher than 0.7, indicating a high level of agreement between the two coders and ensuring a good accuracy of the labelling results. After completing the pilot data labelling, the first author checked the rest of the search results (1,447 code files and 136 projects). After removing the duplicate results, we retained 79 projects under the *Repository* label and 180 code files under the *Code* label. **Step 3.** To further get the application domains represented in the dataset, we categorized the projects based on the project's description documents and the specific function of the code. Projects were classified into one of the following categories: Games, Web Applications, Utility Tools, AI Applications, Network Communication Applications, and Others. Figure 6 shows the classification and distribution of the

application domains of the Copilot-generated code in the dataset from the *Repository* and *Code* labels respectively.

For the code files retained under the *Repository* label, we consider the entire code file as code generated by Copilot. In other words, we assume that Copilot generates all the code in the file because it was stated in the README file that it was all generated by Copilot. 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. Instead of identifying the specific Copilot-generated code in this step, we combine the warning messages from the security scan and the code comments in the file to determine whether Copilot generates the code snippet with the security problem (this process is explained further in Section 3.3.2). As a result, we obtained 452 code files with different contents, 272 from the *Repository* label and 180 from the *Code* label. 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 [15].
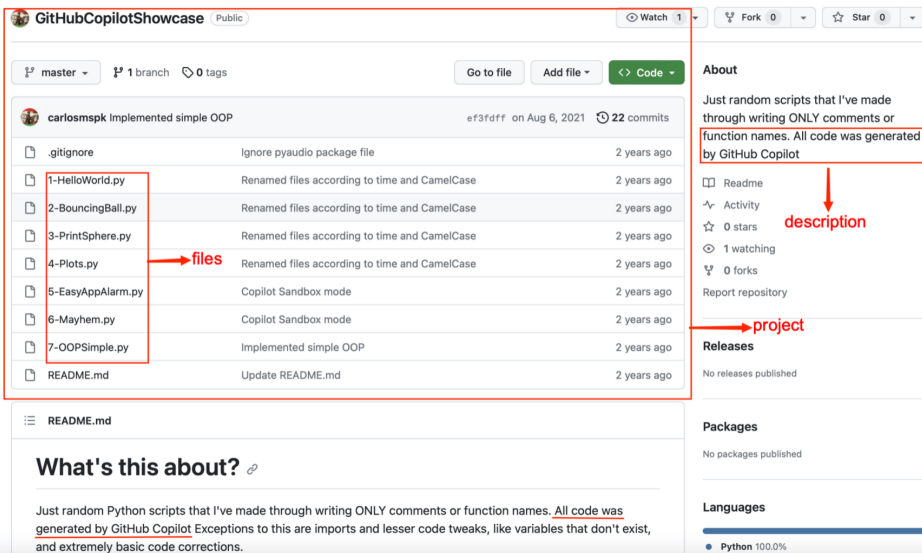

Fig. 3. Example of *Inclusion Criterion 1*: projects fully written by Copilot

Table 2. Code snippets from GitHub

#	Language	# Code Snippets: Repository	# Code Snippets: Code	Total
L1	Python	143	134	277
L2	JavaScript	129	46	175
Total		272	180	**452**

3.3 Data Processing and Analysis

3.3.1 Data Processing. **Step 4.** 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,

```
21      # Github Copilot wrote this class with just the class name as prompt!
22  ∨   class EarlyStopper:
23          def __init__(self, patience=10, decimal=5):
24              self.patience = patience
25              self.decimal = decimal
26              self.reset()
27
28  ∨       def track(self, loss):
29              if np.around(loss, self.decimal) >= np.around(self.best_loss, self.decimal):
30                  self.num_bad_epochs += 1
31              else:
32                  self.num_bad_epochs = 0
33                  self.best_loss = loss
34              if self.num_bad_epochs >= self.patience:
35                  return True
36              return False
```
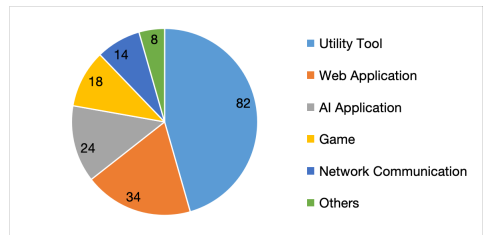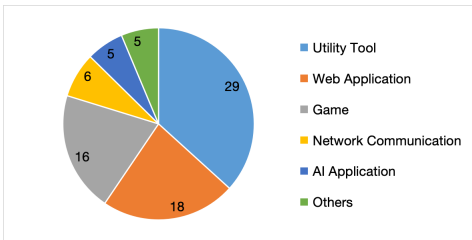
Prompt message

Fig. 4. Example of *Inclusion Criterion 2*: files with comments showing the code generated by Copilot

```
Code    Blame                                                      Raw  ⧉  ⬇

1    // BOJ 2438 [Printing Stars 1]      →   BOJ refers to an algorithm problem
2    // Supported by GitHub Copilot              on LeetCode platform
3
4    #include <bits/stdc++.h>
5    using namespace std;
6
7    int main() {
8        int N; cin >> N;
9        char s[100]; fill(s, s+100, '*');
10       for(int i=1; i<=N; i++) s[i] = 0, puts(s), s[i] = '*';
11   }
```

Fig. 5. Example of *Exclusion Criterion 1*: files used to solve simple algorithm problems

(a) Code from the *Repository* label

(b) Code from the *Code* label

Fig. 6. Application domains of the Copilot-generated code in our dataset

and Python [16]). Before using CodeQL to scan the identified code 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. To speed up the scan, we stored 20 files in each database using the command line to generate the database needed for CodeqQL queries. It should be noted that generating a database for an exceptionally large number of files would increase the database compilation and scanning time, which is much longer than partitioning them into small databases. In total, we obtained 25 code databases available for CodeQL scanning.

3.3.2 Data Analysis. **Step 5.** We used well-known automated static analysis tools listed by OWASP [36] to scan the collected code snippets. Static analysis has been widely used to detect security issues in code, for its ability to analyze programs without execution [11]. While dynamic analysis offers more precise insights by reasoning about program behavior, it suffers from limited coverage [49]. 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 issues 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).

We first used CodeQL to analyze the code in our dataset. The default query suite for the standard CodeQL query package is `codeql-suites/<lang>-code-scanning.qls`. Each package has several query suites in the *codeql-suite* directory. For example, the `codeql/python-queries` package contains the following query suites [19]:

- `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 issues, we manually matched the results in the query reports with the corresponding CWE IDs. We then selected one additional static security analysis tool for files in each programming language we analyzed: Bandit for Python and ESLint for JavaScript. 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 number of lines unspecified at this stage. **Step 6.** We scanned code snippets from the *Repository* and *Code* labels, and we filtered the scan 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* [10]. We first performed an initial filtering of the results based on the priority of the warnings. Specifically we removed the repeated scan results that were reported by the two tools, then removed the scan results that were not security issues, such as recommendations, to get the initial scan results. Subsequently, to confirm that the security issues were actually caused by the Copilot-generated code, the first author manually checked the scan results. 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 issue based on the line number information provided by the scan results. For the initial results obtained under the *Code* label, we confirmed whether the security issues truly existed and verified if the security issues 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 issue indicated in the scan results. We assessed whether it was generated by Copilot by checking the surrounding comments and determined whether a security

issue existed in that particular context. We further analyzed the filtered scan results in **Step 7**, detailed in Section 4, according to the specific RQs.

We provide the complete dataset (including code snippets, full scan results, and filtered results) in our replication package [15].

4 RESULTS

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

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

Approach. To answer this RQ, we collected 452 Python and JavaScript code snippets generated by Copilot from GitHub projects. We used two static analysis tools (CodeQL + another language-dedicated tool, Bandit for Python and ESLint for JavaScript) to scan and analyze the code snippets and then combine the results obtained from the two tools. The aim is to achieve better coverage of security issues. Therefore, as long as one of the tools detected the presence of a security issue, the code snippet was considered vulnerable.

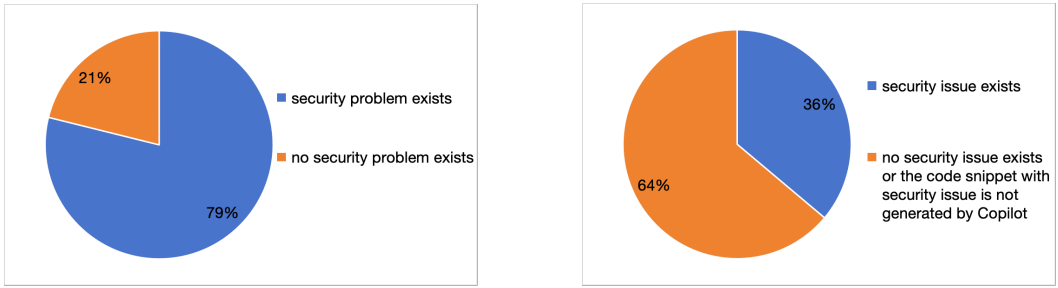We identified three types of warnings from CodeQL analysis:

- *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 the initial scan results related to security issues, we manually checked the scan results. For the scan results from the *Repository* label, we marked them as 1 (security issue exists) and 0 (no security issue exists). For the scan results from *Code* label, we marked them as 1 (security issue exists) and 0 (no security issue exists or the code snippet with a security issue was not generated by Copilot). Figure 7 shows the outcomes of the manual verification on the initial security scan results under the *Repository* and *Code* labels.

When manually checking the scan results from the *Code* label, we also needed to identify whether the security issues obtained from the scan were from Copilot-generated code snippets based on the comment (prompt message) that appears before the method. We provide a working example of the filtration of the scan results in Figure 8. In **Step 1**, we first went to the corresponding file to locate the specific code snippet based on the start line numbers of the scan results. In **Step 2**, we located the code at Line 149 in `aggregation.py`. We found that this code does have a security issue and determined that it was generated by Copilot based on the prompt messages above. Consequently, we marked the corresponding security scan result as "1". We also located the code at Line 145 in `_lytools.py`. We found that this code had a security issue, but there were no prompt messages that would indicate that it was generated by Copilot. Consequently, we marked the corresponding security scan result as "0" and discarded this scan result from further analysis.

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 issues detected.

Results. Table 3 shows the numbers of code snippets for different types and the numbers and percentages of code snippets with security weaknesses. From the statistical results, we found that

(a) Results from the *Repository* label

(b) Results from the *Code* label

Fig. 7. Manual verification of the initial security scan results for the *Repository* and *Code* label

Fig. 8. Example of filtering scan results from the *Code* label that are generated by Copilot

out of the 452 code snippets generated by Copilot, 25.9% of them have security weaknesses. Among the Python and JavaScript code snippets, Python exhibits a higher proportion of security issues compared to JavaScript. Out of the 277 Python code snippets, 91 (32.8%) have security weaknesses. Among the 175 JavaScript code snippets we collected, 43 (24.6%) have security weaknesses. Note that one related code snippet may contain multiple instances of a specific CWE.

Table 3. The number and percentage of code snippets with security weaknesses generated by Copilot

Language	# Snippets	# Snippets containing security weaknesses	%
Python	277	91	32.8%
JavaScript	175	43	24.6%
Total	**452**	**134**	**29.6%**

4.2 RQ2: What security weaknesses are present in the code snippets generated by Copilot?

Approach. To answer RQ2, we processed the scan results collected by RQ1 that were manually checked to contain vulnerabilities. In total, we found 544 security issues in 452 code snippets, and Table 4 shows the number of security weaknesses found in code files of different programming languages.

Table 4. The number of security weaknesses in code snippets generated by Copilot

Language	# Snippets containing security weaknesses	# Total security weaknesses
Python	91	277
JavaScript	43	175
Total	**134**	**544**

For each code snippet, we used CWEs to classify the security issues 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 [51]. To more accurately identify the type of security issue with the corresponding CWE, we did not directly accept the provided information of the static analysis tools (such as the CWE number reported by Bandit). Instead, we manually identified the type of security issue at the specific location reported in the warning message of the static analysis tool and matched it with the corresponding CWE number. Initially, two authors independently matched each description of the security issue with a CWE ID. 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. This process continued until all the descriptions of the security issues in the results were matched with CWE IDs. In the final stage, we performed a statistical analysis of CWE weaknesses in 134 code snippets that contained security weaknesses.

Results. Table 5 shows the distribution of CWEs in the code snippets and the total number of occurrences (Frequency) of the CWE in the code snippets (we put those CWEs whose Frequency = 1 in "Others"). In total, we found 544 CWEs in 452 code snippets. These security weaknesses were related to 38 CWEs, indicating that developers face a variety of security weaknesses when using Copilot. *CWE-330: Use of Insufficiently Random Values* is the most frequently occurring CWE, as it represents 23.3% of the security weaknesses, followed by *CWE-94: Improper Control of Generation of Code ('Code Injection')* , *CWE-78: OS Command Injection* and *CWE-95: Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval 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-22: Improper Limitation of a Pathname to a Restricted Directory* and *CWE-770: Allocation of Resources Without Limits or Throttling*. This indicates that the types of security issue are closely related to the specific scenarios in which developers use Copilot, emphasizing the importance of maintaining vigilance and caution when programming. Besides, Table 6 presents the top 5 CWEs that appear in Python and JavaScript code snippets.

Table 5. Distribution of CWEs in code snippets

CWE-ID	Frequency of Specific CWE	Percentage
CWE-330	127	23.3%

Table 5 – continued from previous page

CWE-ID	Frequency of Specific CWE	Percentage
CWE-94	115	21.1%
CWE-78	81	14.9%
CWE-95	30	5.5%
CWE-772	23	4.2%
CWE-89	15	2.8%
CWE-457	15	2.8%
CWE-259	14	2.6%
CWE-670	13	2.4%
CWE-396	11	2.0%
CWE-617	8	1.5%
CWE-628	7	1.3%
CWE-563	7	1.3%
CWE-561	7	1.3%
CWE-798	6	1.1%
CWE-703	6	1.1%
CWE-502	6	1.1%
CWE-20	6	1.1%
CWE-312	5	0.9%
CWE-605	4	0.7%
CWE-295	4	0.7%
CWE-252	4	0.7%
CWE-367	3	0.5%
CWE-327	3	0.5%
CWE-185	3	0.5%
CWE-79	2	0.4%
CWE-682	2	0.4%
CWE-665	2	0.4%
CWE-400	2	0.4%
CWE-215	2	0.4%
CWE-200	2	0.4%
CWE-117	2	0.4%
CWE-116	2	0.4%
Others	=1	<1%
38 Types	**Total: 544**	

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-95 (Improper Neutralization of Directives in Dynamically Evaluated Code)
3	CWE-772 (Missing Release of Resource after Effective Lifetime)	CWE-563 (Assignment to Variable without Use)
4	CWE-89 (Improper Neutralization of Special Elements used in an SQL Command)	CWE-20 (Improper Input Validation)
5	CWE-259 (Use of Hard-coded Password)	CWE-185 (Incorrect Regular Expression)

4.3 RQ3: How many security weaknesses belong to the CWE Top-25?

Approach. The code snippets in our collected dataset were mainly generated in 2022 and 2023. To compare whether the security issues in Copilot-generated code are widespread in this period, we chose MITRE 2023 CWE Top-25 list [50] as our baseline. In addition, MITRE maintains another list of "Stubborn Weaknesses in the CWE Top 25" [53], which includes those CWEs that have consistently appeared in the Top 25 most dangerous software weaknesses of the past five years (2019-2023). We compared the CWEs obtained in RQ2 with the 2023 CWE Top 25 and "Stubborn Weaknesses" in the CWE Top 25.

Results. The distribution of CWEs found compared to the MITER list is shown in Table 7. The results show that the CWE weaknesses present in the code generated by Copilot belong to eight CWEs. We highlight those CWEs that are "Stubborn Weaknesses" (SW) in bold. This means these are present issues and are currently among the most common and serious security weaknesses in practice. It is worth noting that the 218 security issues present in the code snippet correspond to these eight CWEs, while another 30 CWEs cover the remaining 326 security issues. This indicates that the CWE Top-25 weaknesses are also prevalent in the code generated by Copilot. At the same time, we can see that *CWE-78: OS Command Injection* occurs more frequently in Copilot-generated code. *CWE-94: Improper Control of Generation of Code* is in the list of the CWE Top-25 [50], and meanwhile it is also one of the weaknesses with a high occurrence frequency in our results. Some CWEs with a higher ranking in the Top-25 list do not frequently appear in Copilot-generated code, such as *CWE-79: Cross-site Scripting* and *CWE-89: SQL Injection*.

Table 7. The CWEs that belong to the 2023 CWE Top-25 list

CWE-ID	Description	Is SW	# Related Snippets	Frequency
CWE-94	Improper Control of Generation of Code	No	34	115
CWE-78	Improper Neutralization of Special Elements used in an OS Command	Yes	18	81
CWE-798	Use of Hard-coded Credentials	Yes	3	6
CWE-502	Deserialization of Untrusted Data	Yes	2	6
CWE-20	Improper Input Validation	Yes	3	6
CWE-79	Improper Neutralization of Input During Web Page Generation	Yes	2	2
CWE-276	Incorrect Default Permissions	No	1	1
CWE-22	Improper Limitation of a Pathname to a Restricted Directory	Yes	1	1
Total				**218**

5 DISCUSSION

In this section, we explain the study results in Section 5.1 and then discuss their implications in Section 5.2.

5.1 Interpretation of Results

RQ1: How secure is the code generated by Copilot?

Among the 452 code snippets generated by Copilot, we found that 29.6% of these code snippets contain security weaknesses. Although in our results, the proportion of security issues in Python code snippets is higher than in JavaScript, there is no significant difference. Using Copilot to write code in Python or JavaScript can generally lead to security issues. This could be attributed to features that made their code more flexible, such as dynamic typing and interpretation. From

Table 8. CWEs and security issue types in Copilot-generated code

Type of Security Issue	Relevant CWEs
Data Neutralization Issues	CWE-94, CWE-78, CWE-117, CWE-89
Resource Management Errors	CWE-772, CWE-502
Error Conditions, Return Values	CWE-396, CWE-617
Bad Coding Practices	CWE-628, CWe-563
Credentials Management Errors	CWE-798
Information Management Errors	CWE-312, CWE-215
Authentication Errors	CWE-295
Concurrency Issues	CWE-367
File Handling Issues	CWE-22

a Copilot point of view, the generated code does not need to reason about the whole program because it is dynamically typed, consequently, it does not need to read the flow of the whole program to generate working suggestions. In summary, developers should pay special attention to the security of Copilot-generated code, taking appropriate measures to validate input data and manage resources effectively to minimize security risks. The results of RQ1 suggest that in practical production, Copilot can help developers write code faster and increase productivity [20, 39]. Still, additional security assessments and fixes are also required to ensure that the generated code does not introduce potential security risks.

RQ2: What security weaknesses are present in the code snippets generated by GitHub Copilot?

After conducting a security evaluation of 452 code snippets generated by Copilot, a total of 544 security weaknesses were identified, involving 38 CWEs, which is around 10% of the CWEs (439 CWEs) [52]. This may be because Copilot generates code in different programming languages and application scenarios. In addition, since the Copilot base model (Codex) is 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 by Copilot. This set of 38 CWEs covers many security issues, and Table 8 shows the types of security issues to which these CWEs are relevant.

The diversity of security weaknesses indicates that developers using Copilot face various security risks. These risks are diverse, covering different development environments and application scenarios. In addition, we find that CWEs in Python are mainly related to data processing and system calls. In contrast, CWEs in JavaScript are often associated with dynamic code generation problems and security issues in web development. This may be because the two languages are designed differently. JavaScript is more commonly used for web development, while Python is more widely used in areas such as data processing and scientific computing. 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. For example, developers should perform adequate validation of user inputs. It is also necessary to restrict the program's permissions so that they only access essential resources.

The results of RQ2 reveal the security weaknesses that developers may encounter in an actual production environment and their frequency of occurrence, which can help developers to be aware of the security aspects of the code generated by Copilot and to take appropriate measures to address the security weaknesses in an informed manner.

RQ3: How many security weaknesses belong to the CWE Top-25?

As shown in Table 7, eight of the CWEs in the Copilot-generated code can be found in the 2023 CWE Top-25 list, covering more than 218 security issues (40.1% of 544 identified CWEs) in our dataset. This indicates that the commonly acknowledged CWE Top-25 weaknesses in software development,

which are considered the most prevalent and dangerous, are also prevalent in the code generated by Copilot. 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. Meanwhile, we found that some higher-ranked CWEs in the CWE Top-25 list, such as *CWE-89: SQL Injection*, did not frequently appear in our scan results. This could be because the generated code does not establish connections with external resources (e.g., databases). Besides, we also observed that some vulnerabilities from the CWE Top-25 list were not detected in our scans, indicating that Copilot may sanitize and prevent specific weaknesses from being suggested to developers. GitHub is gradually improving the security of Copilot and its underlying model (Codex) [18]. 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 [32].

We also identified 30 security weaknesses in the code that do not belong to the CWE Top-25 list. Although these less common security weaknesses 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. Developers should also be aware of these less common security weaknesses to fully protect their code from attacks.

5.2 Implications

Importance of continuous security analysis of Copilot-generated code. We conjecture practitioners using Copilot will likely encounter security weaknesses, regardless of the programming language used. Our study results reflect the inevitability of security weaknesses in Copilot-generated code, and there are a variety of security weaknesses in Copilot-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. Maintaining a rigorous process of security checks concurrently with code generation can identify potential vulnerability risks and rectify the security issues 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 for checking and preventing security issues when committing code generated by Copilot. Initially, we can turn to automated tools to continuously scan the Copilot-generated code for known security issues, such as CWEs. Recognizing that these automated tools may not always detect all security issues, especially newly emerging ones or those that are context-specific, it is necessary to conduct a subsequent manual assessment, including manual security code review for Copilot-generated code. By embracing this combined strategy for continuous security analysis, we can ensure a robust security shield for the code committing process with Copilot-generated code.

Prevention of security issues in Copilot-generated code. According to our study results, we provide the following suggestions on how to prevent potential security issues in Copilot-generated code: (1) Targeted security countermeasures: Based on the frequency of related CWEs in Copilot-generated code, practitioners can proactively prevent and address security issues in a targeted manner. When using Copilot to generate Python or JavaScript code, developers should focus on dedicated security weaknesses (CWEs) in Copilot-generated code of different programming languages (see Table 6). Furthermore, a recent study shows that security weaknesses appear in certain code suggestions, but not all [32]; consequently, developers should carefully select potentially more secure suggestions, with the assistance of tools, that do not expose the code to vulnerabilities. (2) Standardized security assessment: Common security weaknesses in software development are

also prevalent in the code generated by Copilot. 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 and follow the mitigation measures for related CWEs provided by the MITRE [51]. 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 code generated by Copilot. (3) Enhancing prompt engineering to generate security code: The instruction tuning schemes in code generation not only impacts 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 [23]. We should incorporate security considerations from the initial stages of code generation. For specific security scenarios that are prone to CWEs, we can improve the security of code by enhancing prompt engineering, e.g., with the prompt patterns proposed in [58].

6 THREATS TO VALIDITY

The validity threats are discussed according to the guidelines in [42]. 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 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. Since this process was manually done, it may have been influenced by personal bias. In this regard, two authors conducted the pilot 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 more specific definition. Second, to mitigate the personal bias, two authors independently assigned each security issue description a CWE ID. 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 Copilot-generated code snippets collected from open-source projects on GitHub. During the filtering process, we excluded code that utilized Copilot to solve algorithmic problems, aiming to ensure that the collected data genuinely reflected real-world production environments. 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 scan results (as the case with static analysis, in general, [25, 49]). 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 scan results remain consistent when used multiple times. In addition, we performed two rounds of scanning 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. 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 452 code snippets generated by Copilot from GitHub projects and analyzed those snippets for security weaknesses using static analysis tools. 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) 29.8% of the 452 Copilot-generated code snippets contain security weaknesses. Developers have a high risk of raising security issues when using Copilot, regardless of the programming language, so security checks are necessary. (2) The detected security weaknesses are diverse in nature and are associated with 38 different CWEs. Developers face a variety of development scenarios and application environments in production and need the appropriate security awareness and skills. (3) Among these CWEs, eight appear in the MITRE CWE Top-25 list, and six belong to the Stubborn Weaknesses, demonstrating their severity.

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.

DATA AVAILABILITY

We have shared the link to our dataset in the reference [15].

ACKNOWLEDGMENTS

REFERENCES

[1] [n. d.]. *Octoverse 2022 Top Programming Languages*. https://octoverse.github.com/2022/top-programming-languages
[2] 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.
[3] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2022. Grounded copilot: How programmers interact with code-generating models. *arXiv preprint arXiv:2206.15000* (2022).
[4] Brett A Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2022. Programming Is Hard–Or at Least It Used to Be: Educational Opportunities And Challenges of AI Code Generation. *arXiv preprint arXiv:2212.01020* (2022).
[5] 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. *Advances in Neural Information Processing Systems* 33 (2020), 1877–1901.

[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).

[7] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46.

[8] 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.

[9] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, Zhen Ming, et al. 2022. GitHub Copilot AI pair programmer: Asset or Liability? *arXiv preprint arXiv:2206.15331* (2022).

[10] 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.

[11] 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.

[12] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. 2008. Selecting Empirical Methods for Software Engineering Research. Springer, 285–311.

[13] 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).

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

[15] 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"*.

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

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

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

[19] 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.

[20] GitHub. 2024. *A developer's second brain: Reducing complexity through partnership with AI*. https://github.blog/2024-01-17-a-developers-second-brain-reducing-complexity-through-partnership-with-ai.

[21] 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.

[22] Jingxuan He and Martin Vechev. 2023. Controlling Large Language Models to Generate Secure and Vulnerable Code. *arXiv preprint arXiv:2302.05319* (2023).

[23] Jingxuan He, Mark Vero, Gabriela Krasnopolska, and Martin Vechev. 2024. Instruction Tuning for Secure Code Generation. *arXiv preprint arXiv:2402.09497* (2024).

[24] 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.

[25] 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.

[26] 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.

[27] Raphaël Khoury, Anderson R Avila, Jacob Brunelle, and Baba Mamadou Camara. 2023. How Secure is Code Generated by ChatGPT? *arXiv preprint arXiv:2304.09655* (2023).

[28] Sila Lertbanjongngam, Bodin Chinthanet, Takashi Ishio, Raula Gaikovina Kula, Pattara Leelaprute, Bundit Manaskasem-sak, 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.

[29] 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.

[30] Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Shuai Wang, and Cuiyun Gao. 2022. CCTEST: Testing and Repairing Code Completion Systems. *arXiv preprint arXiv:2208.08289* (2022).

[31] 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* (2024).

[32] 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.

[33] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2022. Reading Between the Lines: Modeling User Behavior and Costs in AI-Assisted Programming. *arXiv preprint arXiv:2210.14306* (2022).

[34] 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.

[35] OpenAI. [n. d.]. *Codex*. https://openai.com/blog/openai-codex.

[36] OWASP. [n. d.]. *Source Code Analysis Tools*. https://owasp.org/www-community/Source_Code_Analysis_Tools.

[37] 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

[38] 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.

[39] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2022. Do Users Write More Insecure Code with AI Assistants? *arXiv preprint arXiv:2211.03622* (2022).

[40] Rohith Pudari and Neil A Ernst. 2023. From Copilot to Pilot: Towards AI Supported Software Development. *arXiv preprint arXiv:2303.04142* (2023).

[41] 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.

[42] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14 (2009), 131–164.

[43] 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).

[44] 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).

[45] 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).

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] The MITRE Corporation. 2023. *CWE - 2023 CWE Top 25*. https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html.

[51] The MITRE Corporation. 2023. *CWE - Common Weakness Enumeration*. https://cwe.mitre.org/data/index.html.

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

[53] The MITRE Corporation. 2023. *Stubborn Weaknesses in the CWE Top 25*. https://cwe.mitre.org/top25/archive/2023/2023_stubborn_weaknesses.html.

[54] 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.

[55] Catherine Tony, Markus Mutas, Nicolás E Díaz Ferreyra, and Riccardo Scandariato. 2023. LLMSecEval: A Dataset of Natural Language Prompts for Security Evaluations. *arXiv preprint arXiv:2303.09384* (2023).

[56] 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.

[57] 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.

[58] 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).

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

[60] 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).

[61] 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.

[62] 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.