Security Weaknesses of Copilot Generated Code in GitHub

Yujia Fu
School of Computer Science
Wuhan University
Wuhan, China
yujia_fu@whu.edu.cn

Peng Liang
School of Computer Science
Wuhan University
Wuhan, China
liangp@whu.edu.cn

Amjed Tahir
School of Mathematical and Computational Sciences
Massey University
Palmerston North, New Zealand
a.tahir@massey.ac.nz

Zengyang Li
School of Computer Science
Central China Normal University
Wuhan, China
zengyangli@ccnu.edu.cn

Mojtaba Shahin
School of Computing Technologies
RMIT University
Melbourne, Australia
mojtaba.shahin@rmit.edu.au

Jiaxin Yu
School of Computer Science
Wuhan University
Wuhan, China
jiaxinyu@whu.edu.cn

ABSTRACT

Modern code generation tools use AI models, particularly Large Language Models (LLMs), to generate functional and complete code. While such tools are becoming popular and widely available for developers, using these tools is often accompanied by security challenges, leading to insecure code merging into the code base. Therefore, it is important to assess the quality of the generated code, especially in terms of its security. Researchers have recently explored various aspects of code generation tools, including security. However, many open questions about the security of the generated code require further investigation, especially the security issues of automatically generated code in the wild. To this end, we conducted an empirical study by analyzing the security weaknesses in code snippets generated by GitHub Copilot that are found as part of publicly available projects hosted on GitHub. The goal is to investigate the types of security issues and their scale in real-world scenarios (rather than crafted scenarios). To this end, we identified 435 code snippets generated by GitHub Copilot from publicly available projects. We then conducted extensive security analysis to identify Common Weakness Enumeration (CWE) instances in these code snippets. The results show that (1) 35.8% of Copilot generated code snippets contain CWEs, and those issues are spread across multiple languages, (2) the security weaknesses are diverse and related to 42 different CWEs, in which *CWE-78: OS Command Injection*, *CWE-330: Use of Insufficiently Random Values*, and *CWE-703: Improper Check or Handling of Exceptional Conditions* occurred the most frequently, and (3) among the 42 CWEs identified, 11 of those belong to the currently recognized 2022 CWE Top-25. Our findings confirm that developers should be careful when adding code generated by Copilot (and similar AI code generation tools) 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.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

CCS CONCEPTS

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

KEYWORDS

Code Generation, Security Weaknesses, CWEs, GitHub Copilot

ACM Reference Format:
Yujia Fu, Peng Liang, Amjed Tahir, Zengyang Li, Mojtaba Shahin, and Jiaxin Yu. 2023. Security Weaknesses of Copilot Generated Code in GitHub. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [34]. 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 an active research discussion topic [22, 36]. 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 predicate calculus [46]. 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 in the form of 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-answering systems [5]. 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 [24]. Currently, code generation tools based on LLMs have also been widely applied, such as Codex by OpenAI [26], AlphaCode by DeepMind [21], and CodeWhisperer by Amazon [3].

These models are trained on billions of public open-source lines of code, which includes public code with unsafe coding patterns [15]. Therefore, code generation tools based on such models can pose security risks, and the code they generate may also have security weaknesses. For example, GitHub Copilot may produce some insecure code, as its underlying model Codex is pre-trained on untrusted data from GitHub [4], which is known to contain buggy programs [31]. 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 [6, 20, 30, 47], and relatively less attention has been paid to security aspects [28, 29, 35]. 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* [11]". GitHub also provides tools such as CodeQL to help developers scan the security issues in their code.

To this end, we conducted an empirical study on the security weaknesses of 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 [6]. We collected 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) 35.8% of Copilot generated code snippets have security weaknesses, and security weaknesses arise regardless of the programming language used; (2) the security weaknesses are diverse and related to 42 different CWEs, in which*CWE-78: OS Command Injection*, *CWE-330: Use of Insufficiently Random Values* and *CWE-703: Improper Check or Handling of Exceptional Conditions* are the most frequently occurred; and (3) among the 42 CWEs identified, 11 CWEs belong to the currently recognized 2022 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 [10] is made available online for future research in this area.) 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; (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 accept the code suggestions provided by Copilot safely.

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 our study results, 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

2.1 AI-assisted Code Generation Tools

With the rise of code generation tools integrated with IDEs, many studies have evaluated these code generation 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.* [49] 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.* [37] 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 [25] 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.* [48] 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.* [45] studied how programmers use and perceive Copilot, and they found that while Copilot may not necessarily improve task completion time or success rate, it often provides a useful starting point. They also noted that participants faced difficulties in understanding, editing, and debugging the code snippets generated by Copilot. Barke *et al.* [2] presented the first theoretical analysis of how programmers interact with Copilot based on the observations of 20 participants. Sila *et al.* [20] 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.

These works above conducted relatively extensive evaluations of code generation tools in terms of correctness, effectiveness, and robustness. However, there is still room for improvement regarding its security, as detailed in the following section.

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 that these tools are based on.

Pearce *et al.* [28] 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 [40]). Siddiq *et al.* [35] 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.* [19] studied the security of the source code generated by the ChatGPT chatbot based on LLMs, and they found that ChatGPT was aware of potential weaknesses but still frequently generated some non-robust code.

Several researchers also compared the situation where code generation tools produce insecure code with that of human developers. Sandoval *et al.* [33] 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.* [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 it comes to introducing vulnerabilities in code. In addition, researchers have also constructed datasets to test the security of these tools. Tony *et al.* [44] proposed LLMSecEval, a dataset containing 150 natural language prompts that can be used to evaluate the security performance of LLMs. Siddiq *et al.* [36] provided a dataset, SecurityEval, for testing whether a code generation model has weaknesses. The dataset contains 130 Python code samples.

Unlike the works above, 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.* [28] study, the research data came from code generated by the authors using Copilot based on the natural language prompts related to high-risk network security weaknesses. In addition to this, Pearce et al. configured CodeQL only to examine CWEs targeted by security weaknesses associated with the prompted 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 [16]. There are two used methods for vulnerability detection in source code: via static and dynamic code analysis. Dynamic analysis techniques are more sound and precise but lack coverage [9]. On the other hand, static analysis is less precise but offers greater coverage and allows to analyze programs without the need to execute them [39]. Static analysis has been widely used to find security issues in code, given it is cheaper to run and can conduct whole program analyses without the need to execute the program [7]. OWASP [27] provides a list of commonly used static analysis tools. 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 [28, 35, 43].

Kaur *et al.* [18] compared static analysis tools for vulnerability detection in scanning C/C++ and Java source code. Tomasdottir *et al.* [43] conducted an empirical study on ESLint, the most commonly used JavaScript static analysis tool among developers. Pearce *et al.* [28] used CodeQL for security weakness scanning of generated Python and C++ code. Siddiq *et al.* [36] used Bandit to check Python code generated using a test dataset.

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 a set of 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 empirical study aims to understand the potential security weaknesses in Copilot generated code. We first collected code snippets generated by Copilot from GitHub projects as the data source for our research. 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 through the repository description and the comments provided in the code (details provided in Section 3.2.2). We then analyzed the identified snippets for security weaknesses. We aim to help developers and researchers understand common security weaknesses when using Copilot without focusing on whether the code generation aspects are used correctly.

We conducted this empirical study following the guidelines proposed in [8]. The RQs, their rationale, and the research process of this study (see Fig. 1) are detailed in the subsections below.

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 potentially make the program vulnerable. The answer to RQ1 helps to 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 security weaknesses [28], and developers should conduct a rigorous security review before accepting the code generated by Copilot. As clarified in the documentation of GitHub Copilot "*the users of Copilot are responsible for ensuring the security and quality of their code* [11]". The answer to RQ2 can help developers better understand possible security weaknesses in the code generated by Copilot, thereby enabling them to prevent and fix these weaknesses more effectively.

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

Rationale: The MITRE list contains the top 25 most dangerous security weaknesses. The answer to RQ3 can help developers understand whether the code generated by Copilot contains widely recognized types of security weaknesses 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. We used code snippets generated by Copilot on GitHub as our research object to analyze the relevant security weaknesses of Copilot. Our scripts and dataset are provided online in our replication package [10].

3.2.1 Code Snippets Collection. **Step 1.** To collect 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" not only refers to the code generation tool launched by GitHub but also to tools in the aviation or telemetry fields. Therefore, using the keyword "Copilot" solely may return irrelevant content that may not be related to the use of the tool. On the other hand, using "GitHub Copilot" as a search keyword 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 they are truly related to GitHub Copilot. Although using "GitHub Copilot" as a search keyword 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 need because 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 the 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 our goal is to use automated analysis tools to perform security scans on the collected code snippets, we further limited the types of code snippets during the search to Python, JavaScript, Java, C++, C#, and Go. These are the mainstream languages supported by Copilot, and also the languages supported by CodeQL. We collected the *Code* parts from these search results. 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. Fig. 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 8,004 results, of which 7,749 were from the *Code* label, and 255 were from the *Repository* label. The same search result may contain multiple keywords, meaning there are duplicate projects in the collected data. After removing duplicate projects, we obtained a total of 4249 search results, of which 4081 were from the *Code* label, and 168 were from the *Repository* label. Table 2 shows the number of different language types of search results we obtained from the *Code* label.

Table 1: Search results based on different terms used

#	Search Term	# Code	# Repositories
1	"By GitHub Copilot"	2549	54
2	"Use GitHub Copilot"	1822	77
3	"With GitHub Copilot"	3378	127
Total		**7749**	**255**

3.2.2 Filtering Code Snippets. **Step 2.** After obtaining the results from the keyword searches, we further filtered them by not only considering the accuracy of the keywords but also 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 wanted to obtain code snippets used in real-world projects, we excluded search results used to solve simple algorithmic problems on platforms, such as

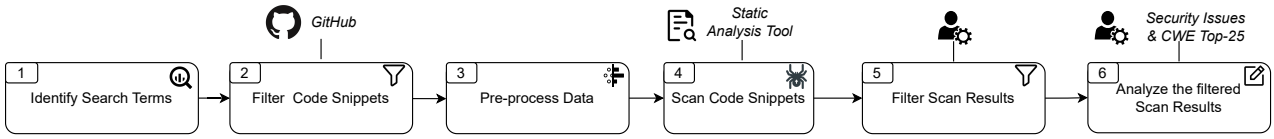

Figure 1: Overview of the research process

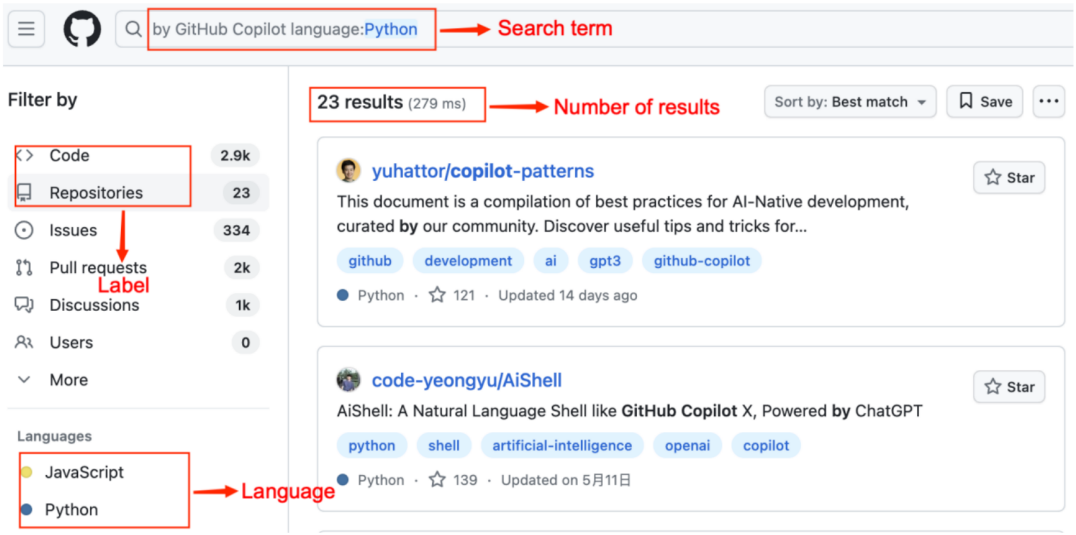

Figure 2: Example of the search process

Table 2: Search Results from GitHub

#	Language	# Code results
L1	Python	784
L2	JavaScript	863
L3	Java	641
L4	C++	437
L5	C#	386
L6	Go	970
Total		**4081**

LeetCode, which generally involve simple code and may not involve security weaknesses.

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 Copilot generated code snippets. In filtering the projects, we followed three rules: (1) for search results under the *Repository* label, we identified projects that are fully generated by Copilot, as declared in the projects description or the associated README file(s). We retained code files for Python, JavaScript, Java, C++, C#, and Go, which are the main languages supported by Copilot. (2) For search results under the *Code* label, we retained code files with comments showing the code generated by Copilot. (3) As we mentioned above, we then excluded code files used to solve simple algorithm problems. We provide examples for the three rules in Figs. 3, 4 and 5. As shown in the example in Fig. 3 for the *Repository* label, we kept all the Python files. In the next example in Fig. 4, we kept the entire file where the Copilot generated code snippet was located. In Fig. 5, the code snippet was removed as it was determined the code just solved a simple algorithmic problem. Meanwhile, 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 all code in the file is generated by Copilot 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).

After completing the pilot data labeling, the first author checked the rest of the search results, and obtained a total of 465 code snippets. After removing duplicate results, we finally obtained 435 different code snippets. Among them, 249 are from the *Repository*

label, and 186 are from the *Code* label. Table 3 shows the types and numbers of code snippets obtained.

Table 3: Code snippets from GitHub

#	Language	# Code Snippets: Repository	# Code Snippets: Code	Total
L1	Python	132	119	251
L2	JavaScript	51	28	79
L3	Java	25	18	43
L4	C++	14	12	26
L6	Go	19	1	20
L5	C#	8	8	16
Total		249	186	**435**

3.3 Data Pre-processing and Analysis

3.3.1 Data Pre-process. **Step 3.** CodeQL is a scalable static security analysis tool that is widely used in practice, and enables 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 [12]). Before using CodeQL to scan the identified code snippets for security weaknesses, we needed to create a CodeQL database for the source code. For interpreted languages like Python and JavaScript, the source code can be directly analyzed, while for compiled languages such as Java, the source code will need to be compiled first and then imported into the CodeQL database. Therefore, we first compiled the code snippets of all compiled languages (i.e., C#, Java, C++, and Go). We removed any code snippets that could not be compiled. For successfully compiled files, we generated the CodeQL database required for queries. At the same time, for interpreted languages Python and JavaScript files, we stored 20 files in each database to improve efficiency, because if we generate a database for an exceptionally large number of files, this would increase the database compilation and scanning time, which is much longer than partitioning them into small databases. In total, we obtained 80 code databases available for CodeQL scanning. Table 4 shows the types and numbers of files in each database.

Table 4: Databases for CodeQL scanning

#	Language	# Databases: Repository	# Databases: Code
L1	Python	7	6
L2	JavaScript	3	2
L3	Java	8	18
L4	C++	5	12
L5	C#	3	7
L6	Go	13	1
Total		**39**	**46**

3.3.2 Data Analysis. **Step 4.** We used well-known automated static analysis tools listed by OWASP [27] to scan the collected code snippets. 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 a dedicated tool for the specific language).

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`. There are several useful query suites in the *codeql-suite* directory of each package. For example, the `codeql/cpp-queries` package contains the following query suites [13]:

- `Cpp-code-scanning.qls`, which is the standard code scanning query for C++. It covers various features and syntax of C++ and aims to discover some common weaknesses in the code.
- `Cpp-security-extended.qls`, which includes some more advanced queries than *cpp-code-scanning.qls* and can detect more security weaknesses.
- `Cpp-security-and-quality.qls`, which combines queries related to security and quality, covering various aspects of C++ 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 related to security weaknesses. 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, the JavaScript test suite provides 203 security checks, and the C++ test suite provides 163 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 other popular static security analysis tools for files in each program languages we analyzed. We used the following popular security analysis tools: Bandit for Python, ESLint for JavaScript, Cppcheck for C ++, Findbugs for Java, Roslyn for C# and Gosec for Go. In cases where we could not directly obtain the CWE ID related to the security issue from the scan results, we manually mapped the security attributes to the corresponding CWE for later analysis. We explain the specific correspondences in detail in Section 4.2.

Step 5. We scanned code snippets from the *Repository* and *Code* labels, and we filtered the scan results before analyzing them. We first removed the scan results that were repeatedly prompted by two of the tools, then removed the results that were unrelated to the security issue, and finally confirmed that the Copilot generated code indeed caused the results related to the security issue. As we explained in Section 3.2.2, we considered the code snippet from the *Repository* label to be the entire code file. Therefore, we kept the entire scan results from the *Repository* label and counted all the security issues they suggested. For the code snippet obtained from the *Code* label, we started by scanning the code file. If the static analysis tool found a security issue in the code, we located the code snippet in the file according to the line number of the security issue indicated by the scanning result. We determined whether it was generated by Copilot based on the comments before and after the code snippet. If Copilot indeed generated the code snippet with a security issue, we kept the scan result for our subsequent statistics. We further analyzed the filtered scan results in **Step 6**, detailed in

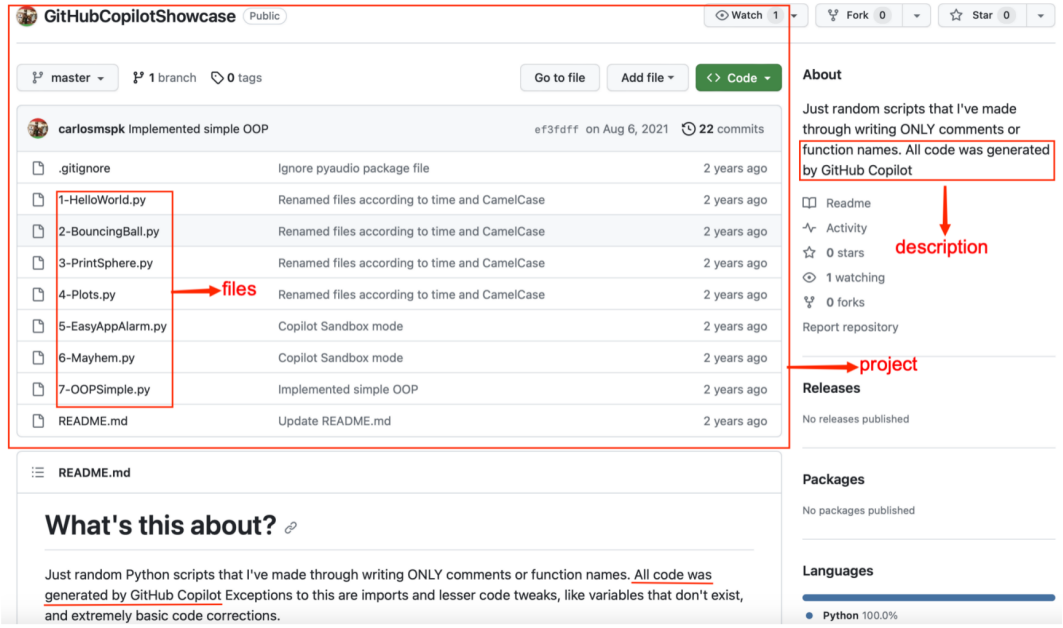

Figure 3: Example of rule 1: project is fully written by Copilot

```
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
```

Figure 4: Example of rule 2: files with comments showing the code generated by Copilot

Section 4 according to the specific RQs. We provide our full dataset (including code snippets, full scan results, and filtered results) in our replication package [10].

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 provide a detailed presentation of the final results for each RQ.

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

Approach. To answer this RQ, we collected 435 code snippets generated by Copilot from GitHub projects. These snippets cover six common programming languages. We used two static analysis tools (CodeQL + another language-dedicated tool) to scan and analyze the code snippets and then combine the results obtained from the two tools. The aim is to achieve a 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.

In the analysis results obtained from the CodeQL tool, three types of warnings were used to describe the detected weaknesses:

Code Blame Raw

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

Figure 5: Example of rule 3: files used to solve simple algorithm problems

Recommendation, which provides suggestions for improving code quality; *Warning*, which alerts to potential weaknesses that could cause code to run abnormally or unsafely; and *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 the other code quality *recommendations*. For the scanning 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 that appears before the method. We provide an example of the scan results filtration in Fig. 6. In **Step 1**, we first went to the corresponding file to locate the specific code snippet based on the start and end lines of the scan results that suggested a security issue. In **Step 2**, we located the code at Line 103 and found no comment indicating that Copilot generated it. In **Step 3**, we also found that the code snippet generated by Copilot in the file was located at Lines 226 to 239, and we then determined that the code snippet generated by Copilot did not cause this security issue and discarded this scan result from further analysis. Finally, we aggregated the filtered results obtained using multiple analysis tools to calculate the number of code snippets with security issues detected.

Results. Table 5 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 out of the 435 code snippets generated by Copilot, 35.8% of them have security weaknesses, regardless of the programming language. There is a higher proportion of security weaknesses in Python and JavaScript code, which are the most popular Copilot languages developers use [38]. Out of the 251 Python code snippets, 39.4% have security weaknesses. Among the 79 JavaScript code snippets we collected, 27.8% have security weaknesses. Among all programming languages, C++ code snippets have the highest proportion of security weaknesses, reaching 46.1%. Go also has a relatively high proportion of security weaknesses, at 45.0%. In comparison, the proportion of files with security issues is lower for C# and Java code, at 25% and 23.2%, respectively.

Table 5: The number and percentage of code snippets with security weaknesses generated by Copilot

Language	# Snippets	# Snippets containing security weaknesses	%
Python	251	99	39.4%
JavaScript	79	22	27.8%
Java	43	10	23.2%
C++	26	12	46.1%
Go	20	9	45.0%
C#	16	4	25%
Total	**435**	**156**	**35.8%**

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

Approach. To answer RQ2, we processed the results of the scans conducted for RQ1, eliminating duplicate security issues detected at the same code snippet location. In total, we identified 600 security weaknesses across 435 code snippets. Table 6 shows the number of security weaknesses found in code files of different programming languages.

Table 6: The number of security weaknesses in code snippets generated by Copilot

Language	# Snippets containing security weaknesses	# Total security weaknesses
Python	99	352
JavaScript	22	93
Java	10	56
C++	12	70
Go	9	18
C#	4	11
Total	**156**	**600**

For each code snippet, we used CWEs to classify the security issues identified by the static analysis. Each CWE has a unique ID and a set of related descriptions, including its potential impact and how to detect and fix the CWE [40]. Some static analysis tools we used, such as *Bandit* and *Gosec*, provide a CWE ID corresponding to the detected security issues in their scan results. For other scan

Step 1

Clear-text logging of sensiti	Logging s	error	This expr	/database_interactions.p	103	69	103	76

Step 2

```
99      def register_new_user(self, user: str, password: str) -> bool:
100         role = 'default'
101         if not self.first_user_registered:
102             role = 'admin'
103             print("first user registered, role set to admin", user, password)
104             self.first_user_registered = True
105         err, res = self.backend.query(
106             f"INSERT INTO users (name, password, role) VALUES (%s, %s, %s);", user, password, role)
107         # print(err, res)
108         if err:
109             # fixme
110             return False
111         else:
112             return True
```

Step 3

```
223         # TODO github copilot suggested this. figure out if it is better
224         # err, res = self.backend.query(
225         #     f"SELECT name, tags, path_to FROM projects WHERE parent_project=%s AND owner=%s;", proje
226         if err:
227             # fixme
228             return {}
229         i = 0
230         for child in res[1]:
231             result['items']['children'].append({
232                 'name': child[0],
233                 'tags': child[1],
234                 'classes': self.get_project_classes(user, child[2]),
235                 'key': i
236             })
237             i += 1
238         pprint.pprint(result)
239         return result
```

Figure 6: Example of filtering scan results from the *Code* label that are generated by Copilot

tools that do not directly give a CWE ID, such as Codeql, ESLint, and FindBugs, we manually associated the provided security issue information with a CWE ID, which is detailed below. The QL queries used in CodeQL often point to a specific CWE, and the scanning process typically displays the CWE-ID associated with the QL query. Although only the name of the QL query is shown in the scan results, we can manually correlate the name field with the CWE ID. For example, the result shows "*Hard-coded credential in API call*", and we know that `Hardcodedcredential.ql` query belongs to CWE-798, this security issue can be mapped to the same CWE (798). In addition, while *FindBugs* and *ESLint* also identify security issues, their scan results only provide descriptions of the security issues. We manually associated the descriptions of the security issues in the results with relevant CWE descriptions to determine the specific CWE category to which these security issues belong. 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. Table 7 shows the list of manually matched CWE IDs and warning messages from CodeQL, ESLint, and FindBugs results. In the final stage, we performed a statistical analysis of CWE weaknesses in 156 code snippets that contained security weaknesses.

Results. Table 8 shows the distribution of CWEs in the code snippets, including the number of code snippets that contain a certain CWE (Related Snippets) and the total number of occurrences (Frequency) of the CWE in the code snippets (we put those CWEs whose Frequency = 1 in "Others"). Note that one related code snippets may contain multiple instances of specific CWE. In total, we found 600 CWEs in 435 code snippets. These security weaknesses were related to 42 types of CWE, indicating that developers face a variety of security weaknesses when using Copilot. *CWE-78: OS Command Injection* is the most frequently occurred CWE, as it was detected in 15 code snippets (representing 14% of the security weaknesses), followed by *CWE-330: Use of Insufficiently Random Values*, *CWE-703: Improper Check or Handling of Exceptional Conditions* , *CWE-400: Uncontrolled Resource Consumption* and *CWE-502: Deserialization of Untrusted Data*. Some CWEs appeared less frequently, such as *CWE-95: Eval Injection*, and *CWE-22: Improper Limitation of a Pathname to a Restricted Directory*.

Additionally, many CWEs occur with a probability of less than 1%, for example, *CWE-176: Improper Handling of Unicode Encoding*, *CWE-312: Cleartext Storage of Sensitive Information*, and *CWE-326: Inadequate Encryption Strength*. This indicates that the types of security issues are closely related to the specific scenarios in which developers use Copilot and make the security issues become apparent, emphasizing the importance of maintaining vigilance and caution when programming.

Table 7: The warning messages of the scan results by CodeQL, ESLint, and FindBugs manually matched with corresponding CWE IDs

Tool	Warning Message	CWE-ID
CodeQL	Reflected server-side cross-site scripting	CWE-79
CodeQL	Flask app is run in debug mode	CWE-215
CodeQL	Clear-text logging of sensitive information	CWE-532
CodeQL	Clear-text storage of sensitive information	CWE-312
CodeQL	Information exposure through an exception	CWE-209
CodeQL	Request without certificate validation	CWE-295
CodeQL	Assignment to constant	CWE-682
CodeQL	Log injection	CWE-117
CodeQL	Identical operands	CWE-570
CodeQL	Incomplete string escaping or encoding	CWE-176
CodeQL	DOM text reinterpreted as HTML	CWE-79
CodeQL	Arbitrary file write during archive extraction ("Zip Slip")	CWE-22
CodeQL	Hard-coded credential in API call	CWE-798
CodeQL	Hard-coded credentials	CWE-798
CodeQL	Uncontrolled data used in path expression	CWE-22
CodeQL	Resource not released in destructor	CWE-416
CodeQL	Missing Dispose call on local IDisposable	CWE-690
CodeQL	Use of the return value of a procedure	CWE-252
CodeQL	Dereferenced variable may be null	CWE-476
ESLint	Generic Object Injection Sink	CWE-502
ESLint	Function Call Object Injection Sink	CWE-20
ESLint	Unsafe Regular Expression	CWE-20
ESLint	Variable Assigned to Object Injection Sink	CWE-95
FindBugs	M S Dm: Hardcoded constant database password	CWE-798
FindBugs	H I Dm: Found reliance on default encoding	CWE-116
FindBugs	M D ICAST: Integral division result	CWE-682
FindBugs	H C IL: There is an apparent infinite loop	CWE-835
FindBugs	M B RV:Exceptional return value of java.io.File.mkdirs()ignored	CWE-252
FindBugs	M D NP: Possible null pointer dereference	CWE-476
FindBugs	M B ODR: DatabaseOperate.readDatabase() may fail to close Connection	CWE-404

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

Approach. The code in our collected dataset was generated between June 2021 and June 2023. To compare whether the security issues in Copilot generated code are widespread in this period, we chose MITRE 2022 CWE Top-25 list [41] as our baseline. Then, we compared the CWEs obtained in RQ2 with the CWE Top-25.

Results. The distribution of CWEs found compared to the MITER list is shown in Table 9. The results show that the CWE weaknesses present in the code generated by Copilot belong to eleven CWE types included in the MITER CWE Top-25 list. 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 237 security issues present in the code snippet correspond to these 11 CWEs, while another 31 CWEs cover the remaining 363 security issues. This indicates that the CWE Top-25 weaknesses 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 codebase. At the same time, we can see that *CWE-78: OS Command Injection* is the most frequently occurring weakness from the Top-25 security weaknesses, ranking sixth in the Top-25 list and first in our RQ2 results. Although *CWE-400: Uncontrolled Resource Consumption* is ranked towards the end of the Top-25 list, it is one of the weaknesses with a high occurrence frequency. Some CWEs with a higher ranking in the Top-25 list do not appear frequently in Copilot generated code, such as *CWE-79: Cross-site Scripting* and *CWE-89: SQL Injection*.

Table 8: Distribution of CWEs in code snippets

CWE-ID	# Related Code Snippets	Frequency of Specific CWE	Percentage
CWE-78	15	84	14.0%
CWE-330	34	81	13.5%
CWE-703	20	78	13.0%
CWE-398	11	60	10.0%
CWE-502	15	56	9.3%
CWE-400	22	50	8.3%
CWE-20	10	19	3.1%
CWE-252	2	13	2.1%
CWE-259	6	13	2.1%
CWE-404	3	13	2.1%
CWE-451	3	13	2.1%
CWE-682	2	13	2.1%
CWE-116	3	11	1.8%
CWE-690	3	9	1.5%
CWE-798	5	9	1.5%
CWE-561	4	8	1.3%
CWE-95	4	8	1.3%
CWE-22	4	6	1.0%
CWE-327	5	5	<1%
CWE-532	3	5	<1%
CWE-563	4	4	<1%
CWE-605	4	4	<1%
CWE-89	3	4	<1%
CWE-295	1	3	<1%
CWE-476	1	3	<1%
CWE-775	2	3	<1%
CWE-117	1	2	<1%
CWE-209	1	2	<1%
CWE-215	2	2	<1%
CWE-416	1	2	<1%
CWE-570	2	2	<1%
CWE-664	1	2	<1%
CWE-676	2	2	<1%
CWE-79	1	2	<1%
CWE-94	2	2	<1%
Others		=1	<1%
42 Types		**Total: 600**	

Table 9: The CWEs that belong to the 2022 CWE Top-25 list

CWE-ID	Description	# Related Snippets	Frequency
CWE-78	OS Command Injection	15	84
CWE-502	Deserialization of Untrusted Data	15	56
CWE-400	Uncontrolled Resource Consumption	22	50
CWE-20	Improper Input Validation	10	19
CWE-798	Use of Hard-coded Credentials	5	9
CWE-22	Improper Limitation of a Pathname to a Restricted Directory	4	6
CWE-89	SQL Injection	3	4
CWE-476	NULL Pointer Dereference	1	3
CWE-94	Code Injection	2	2
CWE-416	Use After Free	1	2
CWE-79	Cross-site Scripting	1	2
Total			**237**

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 435 code snippets generated by Copilot, we found that 35.8% of these code snippets contain security weaknesses. Those weaknesses appear in all six top-used programming languages supported by Copilot. Furthermore, when it comes to the occurrence of security issues in code snippets of different programming languages, it is important to analyze them in conjunction with the popularity of the languages [23]. In code snippets written in languages like Python and JavaScript, which are frequently used with Copilot, there may be a slightly higher number of security issues. However, overall the proportion of security issues across these six languages ranges from 25% to 45%, showing no significant difference.

Besides, we also found that *CWE-502: Deserialization of Untrusted Data* and *CWE-400: Uncontrolled Resource Consumption* problems mainly appeared in code snippets written in Python and JavaScript. This could be attributed to certain features that made their code more flexible, such as dynamic typing and dynamic interpretation. Therefore, developers should pay special attention to the security of their JavaScript and Python-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, although Copilot can help developers write code faster and increase productivity, 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 425 code snippets generated by Copilot, a total of 600 security weaknesses were identified, involving 42 CWE types, which is around 10% of the CWEs (439 CWE types) in software development [42]. This may be due to the reason that Copilot generates code in different programming languages and application scenarios, and a wide variety of application scenarios may lead to various types of security issues. 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 42 CWE types covers many types of security issues, and Table 10 shows the types of security issues that these 42 CWEs are relevant to.

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. At the same time, it also reflects the inevitability of security weaknesses in Copilot generated code. Developers need to have corresponding security awareness and skills and take appropriate security measures to avoid these risks in a timely and targeted manner. In addition, we can see that developers often encounter *CWE-78: OS Command Injection*, *CWE-330: Use of Insufficiently Random Values*, and*CWE-703: Improper Check or Handling of Exceptional Conditions*, *CWE-502: Deserialization of Untrusted Data*, and *CWE-400:Uncontrolled Resource Consumption*, which appear in multiple code snippets and have a high frequency of occurrence. This can remind developers to take timely and targeted security measures to mitigate these risks. For example, developers should perform adequate validation of user inputs. In addition to this, 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 be aware of security aspects of code generated by Copilot and take appropriate measures to address the security weaknesses in an informed manner.

Table 10: The CWEs and Types of Security issues

Type of Security Issue	Relevant CWEs
Web security issue	CWE-79, CWE-94, CWE-690, CWE-732
Access control issue	CWE-252, CWE-259, CWE-327, CWE-338
Input validation and representation issue	CWE-20, CWE-89, CWE-116
Command injection issue	CWE-78, CWE-563, CWE-835
SQL injection issue	CWE-89, CWE-95
File handling issue	CWE-22, CWE-570
Insecure storage	CWE-502, CWE-775
Improper error handling	CWE-398, CWE-400
Encryption issue	CWE-327
Memory management issue	CWE-416
Buffer errors	CWE-476
Insecure random number	CWE-330
Incorrect type conversion	CWE-703

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

As shown in Table 9, eleven of the CWEs in Copilot generated code can be found in the 2022 CWE Top-25 list, covering more than 237 security issues (39.5% of 600 identfied CWEs) in our dataset. This indicates that the commonly acknowledged 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 need to pay special attention to these frequently occurring weaknesses and take corresponding measures to avoid and fix them. 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 enhancing the security of Copilot and its underlying model (Codex) [14]. We also identified 31 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

Code Snippets with Security Weaknesses: In practical production, practitioners often use Copilot to generate code in six languages: Python, JavaScript, Java, C++, Go, and C#. These languages

all inevitably produce security weaknesses. We conjecture that practitioners using Copilot will likely encounter security weaknesses, regardless of the programming language used, and security checks are mandatory. When using Copilot, practitioners should conduct their own assessment of the generated code with the support of security analysis tools. They should exercise extreme caution when attempting to rely entirely on Copilot's behavior, especially for the most commonly used languages with Copilot: Python and JavaScript.

Types of Security Weaknesses in Copilot Generated Code: Practitioners using Copilot may encounter a variety of security weaknesses. Our results show that these weaknesses are related to over 40 CWEs. This finding indicates that there are diverse security scenarios in production, and practitioners must have the corresponding security awareness and skills and adopt multiple security prevention measures to address security risks so that they do not simply accept vulnerable code suggestions. At the same time, our study reveals the frequency of related CWEs. When using Copilot to generate code, practitioners should pay particular attention to specific weaknesses, such as *CWE-78: OS Command Injection*, *CWE-330: Use of Insufficiently Random Values*, *CWE-502: Deserialization of Untrusted Data*, *CWE-703: Improper Check or Handling of Exceptional Conditions*, and *CWE-400: Uncontrolled Resource Consumption*. Our findings can assist practitioners in proactively preventing and addressing security issues in a targeted manner.

The CWEs in Copilot Generated Code from the Top-25 CWE List: Common security weaknesses in software development are also prevalent in 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 take appropriate measures to improve the code security. Additionally, the CWE Top-25 provides a standardized approach for security assessment, and developers can also use it to conduct security audits of the code generated by Copilot. Developers should also follow best practices and use code analysis tools (static, dynamic, or hybrid) to check the suggested code by Copilot (or any code generation tools) before integrating any code suggestions. Such tools can safeguard the code and help in discovering weaknesses early.

6 THREATS TO VALIDITY

The validity threats are discussed according to the guidelines in [32]. Note that we did not consider internal validity threats since we did not investigate any relationships between variables and results.

Construct Validity is the degree to which a measurement can explain the theoretical structure and characteristics of the measurement, reflecting the extent to which the studied operational measures truly represent the researcher's ideas and the content investigated based on the research questions. This study has three threats to the construct validity: *(1) Using the 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 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. At the same time, we assumed that all code files contained in projects declared to be written by Copilot in markdown files were generated by Copilot, and we included them in the research data. This has an impact on the construct validity, as we could not have excluded the possibility of human-written code files in these projects. *(3) Manual association of CWEs* – We manually associated some warning messages prompted by static analysis tools with CWEs. Some warning messages could be associated with multiple CWEs, while we only focused on assigning one most suitable CWE for each warning message. Personal bias may occur with this step, impacting the final association, and we mitigated the bias by two authors conducting the association with the assessment by a security expert.

External Validity refers to the extent to which research results can be generalized and the degree to which people outside the investigated cases are interested in the research results. It indicates whether the research results are representative and can be validated in similar contexts. 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. Due to the reason that the data from GitHub is not diversified enough, there are many code snippets from Game projects. The peculiarity of the data source may make the dataset incomplete, thereby threatening the external validity of the results. However, GitHub is one of the largest code hosting platforms in the world, with hundreds of millions of public code repositories, and is popular among developers and the technology community. The code snippets obtained from GitHub are diverse, which mitigates this threat. 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 scanning results (as the case with static analysis tools [17, 39]). Although we attempted to use two static analysis tools to increase the coverage of weaknesses, these tools may have limited abilities in analyzing some CWEs with specific error rates, which may impact the completeness and correctness of the results.

Reliability refers to the extent to which a specific research method can produce consistent results. 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 one. 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 has been an active research area due to the advancement of AI and specifically LLMs. AI code generation tools such as Copilot can greatly improve the development efficiency of programmers, but they 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 435 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: (1) 35.8% of the 435 Copilot generated code snippets contain security weaknesses, spreading across six programming languages. (2) The detected security weaknesses are diverse in nature and are associated with 42 different CWEs. The CWEs that occurred most frequently are *CWE-78: OS Command Injection*, *CWE-330: Use of Insufficiently Random Values*, and *CWE-703: Improper Check or Handling of Exceptional Conditions* (3) Among these CWEs, 11 appear in the MITRE CWE Top-25 list, demonstrating their commonality, current and severity. These are: *CWE-78: OS Command Injection*, *CWE-502: Deserialization of Untrusted Data*, *CWE-400: Uncontrolled Resource Consumption*, *CWE-89: SQL Injection*, *CWE-20: Improper Input Validation*, *CWE-22: Improper Limitation of a Pathname to a Restricted Directory*, *CWE-94: Code Injection*, *CWE-476: NULL Pointer Dereference*, *CWE-798: Use of Hard-coded Credentials*, *CWE-79: Cross-site Scripting*, and *CWE-416: Use After Free*.

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.

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.

[2] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2022. Grounded copilot: How programmers interact with code-generating models. *arXiv preprint arXiv:2206.15000* (2022).

[3] 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).

[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. *Advances in Neural Information Processing Systems* 33 (2020), 1877–1901.

[5] 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).

[6] 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).

[7] Trevor Dunlap, Seaver Thorn, William Enck, and Bradley Reaves. 2023. Finding Fixed Vulnerabilities with Off-the-Shelf Static Analysis. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 489–505.

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

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

[10] Yujia Fu, Peng Liang, Amjed Tahir, Zengyang Li, Mojtaba Shahin, and Jiaxin Yu. 2023. *Dataset of the Paper "Security Weaknesses of Copilot Generated Code in GitHub"*.

[11] GitHub. [n. d.]. GitHub Copilot for Individuals. https://docs.github.com/en/copilot/overview-of-github-copilot/about-github-copilot-for-individuals

[12] GitHub. 2021. *CodeQL* (1.6 ed.). GitHub. https://securitylab.github.com/tools/codeql.

[13] GitHub. 2021. *Using the CodeQL CLI*. GitHub. https://docs.github.com/zh/code-security/codeql-cli/using-the-codeql-cli/analyzing-databases-with-the-codeql-cli.

[14] GitHub. 2023. GitHub CopilotX Preview. https://github.com/features/preview/copilot-x Accessed: 2023-07-28.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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).

[20] 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.

[21] 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.

[22] 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).

[23] mend.io. 2023. The Most Secure Programming Languages. https://www.mend.io/most-secure-programming-languages/

[24] 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).

[25] 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.

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

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

[28] 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.

[29] 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).

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

[31] 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.

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

[33] 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).

[34] 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).

[35] 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.

[36] 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.

[37] 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 Genetic and Evolutionary Computation Conference (GECCO)*. ACM, 1019–1027.

[38] Stackscale. 2021. The 9 most popular programming languages to learn in 2021. https://www.stackscale.com/blog/most-popular-programming-languages-to-learn-in-2021/ Accessed on 2023-07-27.

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

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

[41] The MITRE Corporation. 2022. *CWE - 2022 CWE Top 25*. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html#cwe_top_25.

[42] The MITRE Corporation. 2023. CWE VIEW: Software Development. https://cwe.mitre.org/data/definitions/699.html Accessed: 2023-07-28.

[43] 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.

[44] 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).

[45] 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.

[46] 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.

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

[48] 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).

[49] 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.