Vulnerability Analysis of AI-Assisted Code Generation Models

Tuğba Gürgen Erdoğan, Sertaç Güler, Tarık Sümer, Muhammet Göktuğ Ocaklıoğlu

Hacettepe University, Computer Engineering Department, Software Engineering Research Group, Ankara, Turkey

Abstract:

Context: With the increasing use of AI-assisted code generation tools like GitHub Copilot in software development, there is a critical need to assess the potential security risks they introduce.

Purpose: The primary goal of this research is to evaluate the security of code produced by GitHub Copilot, focusing on the identification and categorization of common vulnerabilities.

Method: This study investigates the security vulnerabilities in code generated by GitHub Copilot, identifying significant risks using static analysis tools and machine learning models. We employed a comprehensive literature review, static analysis tools (CodeQL and Bandit), and a BERT model to analyze and identify vulnerabilities in code snippets generated by GitHub Copilot.

Results: Our findings reveal that 393 out of 692 code snippets generated by GitHub Copilot contained a total of 1135 vulnerabilities, spanning 27 different CWE types, with CWE-20, CWE-78, CWE-79, CWE-89, and CWE-259 being the most common.

Conclusion: While GitHub Copilot enhances coding efficiency, it also introduces significant security risks, highlighting the need for careful evaluation and mitigation strategies to ensure software reliability and safety.

Keywords: GitHub Copilot, AI-assisted code generation, code vulnerability, static analysis, CodeQL, Bandit, BERT model, software security

Introduction

Human intensive and multidisciplinary software development [1] has many stakeholders who have different concerns. Recently, generative AI tools such as ChatGPT [2], Github Copilot [3], CodeGen [4] are participated development as inherent stakeholders with their trained data which fed up from mainly open-source project codes. We introduce "ChatGPT API developer" title as a one of title for the practitioners and prompt engineering as a new research field. Investigating the correctness, functionality, reliability, and quality of the generated codes become an emerging area in both software engineering and artificial intelligence fields. Suggesting secure code capability of the AI-assisted tools and mitigation strategies of vulnerabilities seem a critical as well as developers use these tools from a given prompt to save time and reduce development costs throughout the software development lifecycle.

Some studies evaluate the various capabilities of suggested solutions by the AI-assisted tools. Moradi Dakhel et al. [5] evaluate Copilot on the adequacy of recommended code for fundamental algorithmic problems and quality of the recommendations comparing Copilot's solutions with human solutions. Yetiştiren et al. [6] compare the quality of different generative tools in terms of quality metrics: validity, correctness, security, reliability, and maintainability. There a few studies that examine on the vulnerabilities of generated codes. In [7], [8], and [11] have empirically showed the Copilot generate insecure codes about between 40% and 70%. Asare et al. [10] reported that Copilot introduces less than insecure code than humans, risky for using it as a tool to fix bugs.

In this study, we investigate literature comprehensively and encountered 5 studies recently examine the vulnerability of AI-assisted generated codes. Two of them proposed datasets which consist of prompts of several CWE types and insecure codes of written by humans manually or presented by static code tools like CodeQL [11]. We observed that vulnerability analysis is mainly done by static analysis tools. We make further the vulnerability analysis using transformer model using high volume and variety data and presenting a methodology to analyze vulnerability of the AI-assisted generation tools. We validated the presented methodology with the experiments.

We reproduced Python code with various vulnerabilities by GitHub Copilot and analyzed the code generated by Copilot in detail using static analysis and artificial intelligence models. The main goal of our project was to test GitHub Copilot's ability to detect and assess vulnerabilities. To this end, we developed several test scenarios to determine how sensitive Copilot-generated code is to vulnerabilities. These scenarios included the most common vulnerabilities and aimed to reveal how Copilot-generated code reacts to them. Our results helped us understand and evaluate GitHub Copilot's capabilities to detect and address vulnerabilities. This study aims to provide a guide for developers to be more security aware and careful when using GitHub Copilot.

More specifically, the contributions of this article can be listed as follows:

This article begins with a detailed literature review, providing a comprehensive examination of existing research on the security implications of code generation tools. It introduces an updated dataset https://github.com/b21986764/Copilot-s-Vulnerability-Analysis, synthesized from existing datasets: Security-Eval: https://github.com/s2e-lab/SecurityEval, Copilot CWE Scenarios Dataset: https://zenodo.org/records/5225651 and enhanced with results from code analysis tools like CodeQL and Bandit. Furthermore, this research pioneers an automated analysis method to evaluate the vulnerabilities in code generated by GitHub Copilot, setting it apart from previous studies. Through these contributions, the article not only deepens the understanding of potential security issues posed by automated code generation but also offers practical insights and tools for improving the reliability and safety of code produced with the assistance of AI-powered tools.

Background

A ***software vulnerability***[16] refers to a weakness or flaw in a software system that can be exploited by an attacker to compromise the integrity, availability, or confidentiality of the system. These vulnerabilities can exist in various forms, including coding errors, design flaws, configuration mistakes, or even in the underlying algorithms of the software.
These vulnerabilities have been standardized under CWE's. CWE is a community-developed list of common software and hardware weakness types that could have security ramifications. A "weakness" is a condition in a software, firmware, hardware, or service component that, under certain circumstances, could contribute to the introduction of vulnerabilities. A CWE is assigned an ID in the form CWE-<ID>, where the <ID> is simply a unique number chosen at the time of assignment.

CWE ID	Description
CWE-20	Improper Input Validation
CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
CWE-90	Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')
CWE-94	Improper Control of Generation of Code ('Code Injection')
CWE-116	Improper Encoding or Escaping of Output
CWE-117	Improper Output Neutralization for Logs
CWE-209	Information Exposure Through an Error Message
CWE-215	Information Exposure Through Debug Information
CWE-259	Use of Hard-coded Password
CWE-312	Cleartext Storage of Sensitive Information
CWE-319	Cleartext Transmission of Sensitive Information
CWE-327	Use of a Broken or Risky Cryptographic Algorithm
CWE-330	Use of Insufficiently Random Values
CWE-377	Insecure Temporary File
CWE-400	Uncontrolled Resource Consumption ('Resource Exhaustion')
CWE-502	Deserialization of Untrusted Data
CWE-601	URL Redirection to Untrusted Site ('Open Redirect')
CWE-611	Improper Restriction of XML External Entity Reference
CWE-703	Improper Check or Handling of Exceptional Conditions
CWE-730	OWASP Top Ten 2007 Category A10 - Failure to Restrict URL Access
CWE-732	Incorrect Permission Assignment for Critical Resource
CWE-776	Improper Restriction of Recursive Entity References in DTDs ('XML Entity Expansion')
CWE-798	Use of Hard-coded Credentials
CWE-918	Server-Side Request Forgery (SSRF)

CodeQL [11] is a powerful tool developed by GitHub that is often used to perform static analysis on complex codebases. This tool is especially effective for finding vulnerabilities in large and complex software projects. CodeQL converts your codebase into a database and then queries that database using a SQL-like query language. This provides a highly flexible approach to detecting and analyzing complex security issues.

Bandit [13] is a security tool designed specifically for Python codebases. Performs static analysis of Python code to identify potential security flaws that could lead to security breaches or other vulnerabilities.

GitHub Copilot [3] is an AI-powered code completion tool developed by GitHub in collaboration with OpenAI. It is designed to assist developers in writing code more efficiently by providing suggestions and auto-completions based on context.

CodeBERT[14,15] developed by Microsoft, is a pre-trained model designed for both programming and natural languages. It is a multi-lingual model trained on NL-PL pairs in six programming languages: Python, Java, JavaScript, PHP, Ruby, and Go. CodeBERT learns general-purpose representations that are useful for various downstream applications, including natural language code search and code documentation generation.

Related Work

We could identify a few related studies that investigate vulnerability of the AI-assisted generated codes tools. In Table X we summarize these studies with publication years, references, and venue types; titles and objectives; and basic findings with respect to evaluation of the vulnerability of the generated codes..

As seen from the table, there is no study that solely investigates ….

Table X: Secondary studies that address AI-based software development processes and practices

Year [ref], Venue	Title	Objective	Used AI-Generation Tools	Used Vulnerability Analysis and Tools	Findings w.r.t.
2022 [7], Conference	Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions	Evaluating Github Copilot's codes' security given different programming languages	Github Copilot	Static analysis using CodeQL and where it is not possible authors evaluate security manually.	Prompted MITRE's "2021 CWE Top 25" vulnerabilities. They did on Python, C , and Verilog .1689 programs and found 40% as vulnerable in total.%50 C and 38% Python vulnerable.28% of Verilog vulnerable.
2022 [8], Conference	SecurityEval Dataset: Mining Vulnerability Examples to Evaluate Machine Learning-Based Code Generation Techniques	Determining if Copilot is as bad as human developers. They investigate whether Copilot is just as likely to introduce the same software	InCoder Github Copilot	manual and automatic	This dataset is limited to Python samples, introducing a generalizability threat to this work. We observe that most generated code snippets contain insecure code (about 68% and 74% of code generated by InCoder

Year [ref], Venue	Title	Objective	Used AI-Generation Tools	Used Vulnerability Analysis and Tools	Findings w.r.t.
		vulnerabilities as human developers.			and Copilot, respectively).
2023 [10], Journal	Is GitHub's Copilot as bad as humans at introducing vulnerabilities in code?	Comparison of Github Copilot generated codes and human generated codes' vulnerabilities.	Github Copilot	manual analysis	Github Copilot generated codes were less likely to introduce code vulnerabilities than human generated codes. However still risky for using it as a tool to fix bugs.
2023 [11], Journal	Security Weaknesses of Copilot Generated Code in GitHub	Analyze the vulnerabilities of the code automatically generated by Github Copilot.	Github Copilot	static analysis using tools: CodeQL + a dedicated tool listed by OASP for the specific language Python -> Bandi, JavaScript -> ESLint, C++ -> Cppcheck, Java -> Findbugs, C# -> Roslyn, GO -> gosec	35.8% of the code snippets generated by Copilot have security vulnerabilities, and security vulnerabilities occur regardless of the programming language used. Some vulnerabilities were found more frequently.
2023 [12], Journal	CodeLMSec Benchmark: Systematically Evaluating and Finding Security Vulnerabilities in Black-Box Code Language Models	Proposing a method to study the security of code language models to assess their susceptibility to generating vulnerable code and a benchmark dataset.	CodeGen ChatGPT Github Copilot	static analysis, namely CodeQL	focus on the analysis of thirteen representative CWEs that can be detected via static analysis tools + they use different prompts strategies like few shot and one-shot prompts using ChatGPT and CodeGen.

Summarize overall contributions (with respect to the rightmost column in Table X) …

Pearce et al. [7] look into how safe the code made by GitHub Copilot. The main research question of the study is whether Copilot's code suggestions are safe and what affects their security. The researchers tested Copilot with 89 different situations and made 1,689 programs. They found that 40% of these programs had security problems. The study has three main parts, each looking at different security issues in Copilot's code. First, they checked many kinds of security problems in Python and C programs made by Copilot. They found that 44% had problems. They noticed that Copilot's first code suggestions are safer in C than in Python. Copilot did well in some areas like permission and authorization but not so well in others like managing low-level pointers. Secondly, they focused on one problem: SQL injection in Python. They changed the code prompts a bit and saw how Copilot's answers changed. The results were interesting. Copilot mostly gave similar answers, but when the researchers made parts of the code safer, the whole code became safer. Thirdly, they looked at Copilot's work with Verilog, a less common language for hardware. They found 28% of these programs had problems. The study showed that Copilot had trouble with Verilog's syntax but giving more detailed prompts helped make safer code. The researchers used GitHub's CodeQL and their own checks to see if the code was safe. They focused on the top 25 vulnerabilities listed by MITRE in 2021. The authors say that since Copilot is a closed system, it's hard to know exactly how it works. In the end, they reported that Copilot often repeats the same bugs found in open-source

projects. This might be because of the data it was trained on. Copilot keeps learning, so what is considered safe code might change. They admit it didn't check if the code works right and only looked at certain security problems.

Siddiq and Santos [8] introduce the SecurityEval dataset. The dataset consists of 130 Python code samples that cover 75 different types of vulnerabilities by evaluating two code generation models: InCoder, an open-source model, and GitHub Copilot, a closed-source tool, as defined by the Common Weakness Enumeration (CWE). They found out that a significant portion of the generated code from both models contains security flaws. The dataset can be used to compare the generated code with the insecure code samples in the dataset. To further enhance the evaluation process, they combine the SecurityEval dataset with static analyzers. They suggest expanding the SecurityEval dataset to include other programming languages and vulnerability types. Code samples were collected from four external sources: CodeQL, Sonar Rules (static analyzer), Pearce et al.[7] and SecurityEval(themselves).

Asare et al. [10] aim to determine if Copilot is as bad as human developers. They investigate whether Copilot is just as likely to introduce the same software vulnerabilities as human developers. They find that Copilot replicates the original vulnerable code about 33% of the time while replicating the fixed code at a 25% rate. The code used to train such models for CGTs is obtained from a variety of sources and usually has no guarantee of being secure. This is because the code was likely written by humans who are not always able to write secure code. Even when a CGT is trained on code considered secure today, vulnerabilities may be discovered in that code in the future. Researchers have shown that Copilot (a CGT based on the Codex language model) generates insecure code about 40% of the time. While Pearce et al. [7] concluded that Copilot generates vulnerable code 40% of the time, according to the 2022 Open Source Security and Risk Analysis (OSSRA) report by Synopsys, 81% of (2,049) codebases (of human developers) contain at least one vulnerability while 49% contain at least one high-risk vulnerability. They observe a trend of Copilot being more likely to generate the fix for a vulnerability when the sample has a more recent publish date Broadly speaking, their findings here are in line with the findings by Pearce et al. [7] who also show that Copilot has varied performance, security-wise, depending on the kinds of vulnerabilities it is presented with.

Haque et al. [11] investigate the answers to the questions: how secure is the code generated by Copilot in GitHub Projects, what security weaknesses are present in the code snippets generated by Copilot, and how many security weaknesses belong to the MITRE CWE Top-25? For this, they found the code snippets written by copilot through various manual searches on Github and then obtained the dataset by filtering. In this dataset, there are codes written by copilot in python, javascript, java, C++, C#, and Go languages. In order to find the vulnerabilities in these codes, static analysis tools were used. First, all the codes were tested with CodeQL. Then various language-specific static testing tools were used: Bandit for Python, ESLint for JavaScript, Cppcheck for C++, Findbugs for Java, Roslyn for C# and Gosec for Go. The vulnerabilities were manually examined and matched with CWE ids. As a result they reported some quantitative results: (1) 35.8% of Copilot-generated code snippets have security weaknesses, and security weaknesses arise regardless of the programming language used; (2)the security weak- nesses 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.

Hajipour et al. [12] proposing a method to study the security of code language models to assess their susceptibility to generating vulnerable code and a present a non-secure benchmark dataset. Authors focus on the analysis of 13 representative CWEs that can be detected via static analysis tool CodeQL. They use different prompt engineering strategies on CodeGen ChatGPT Github

Copilot generative AI tools. These strategies are their proposed a few shot (FS)-Code, FS-Prompt, and one shot (OS) Prompt.

AI-Generated Datasets

Security Eval

The dataset is presented by Siddiq and Santos [8] and it consists of 121 Python code samples that cover 75 different types of vulnerabilities as defined by the Common Weakness Enumeration by evaluating two code generation models: InCoder, an open-source model, and GitHub Copilot, a closed-source tool. Figure 1 shows the samples for two CWE types of data belonging to the SecurityEval dataset with ID, Prompt and Insecure_code columns. The ID column describes the issued CWE ID along with the origin of that CWE and its numeration starting from 1. The prompt column contains the CWE-related prompts for GitHub Copilot. Insecure code column contains the generated code from the Prompt column using Github Copilot which contains the related CWE ID in the ID column.

	ID	Prompt	Insecure_code
0	CWE-020_author_1.py	import yaml\n\n\ndef yaml_load(filename):\n ...	import yaml\n\n\ndef yaml_load(filename):\n ...
1	CWE-020_author_2.py	import xml.etree.ElementTree as ET\n\n\ndef pa...	import xml.etree.ElementTree as ET\n\n\ndef pa...
2	CWE-020_codeql_1.py	from flask import Flask, request, make_respons...	from flask import Flask, request, make_respons...
3	CWE-020_codeql_2.py	import base64\nimport pickle\n\nfrom flask imp...	import base64\nimport pickle\n\nfrom flask imp...
4	CWE-020_codeql_3.py	from flask import Flask, redirect, request\nim...	from flask import Flask, redirect, request\nim...

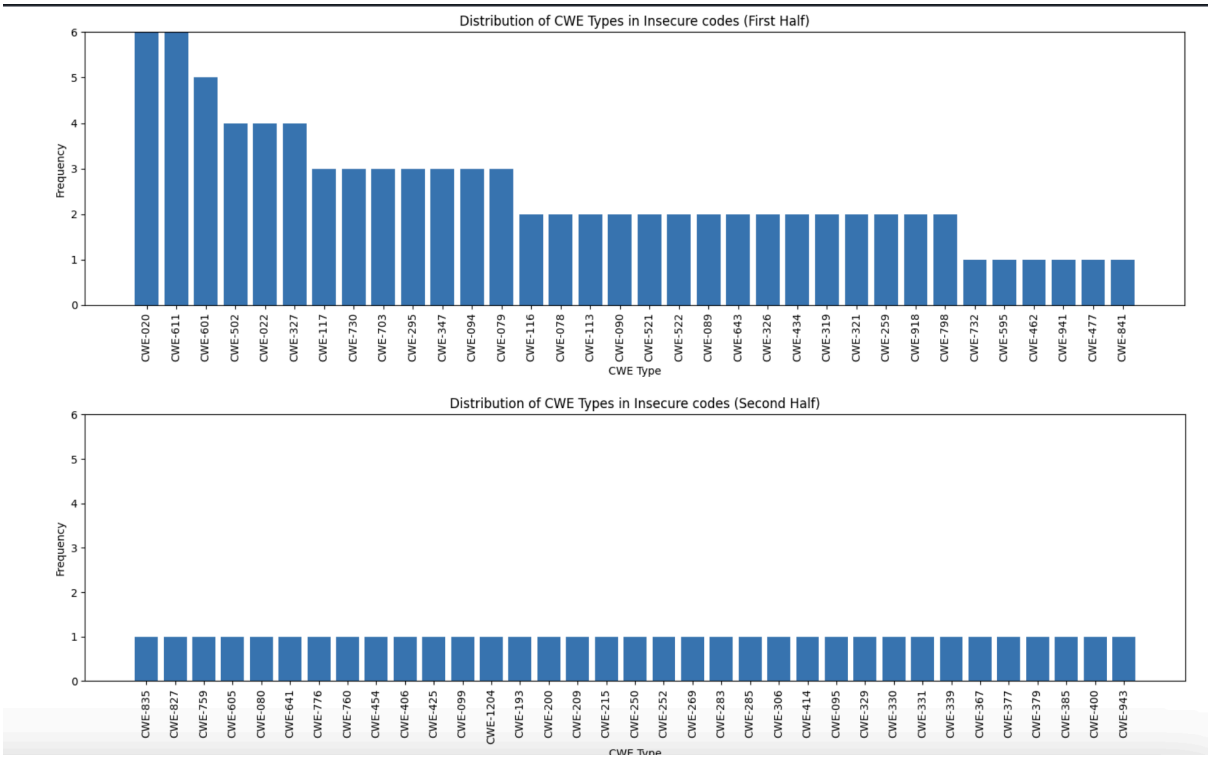
Figure 1: Data for two CWE types of SecurityEval dataset

We investigate the distribution of CWE types examples in the dataset to get a deep understanding of the dataset. Distribution of the CWE types in insecure codes with two halves are depicted in Figure 2. There are code examples in many different CWE types. Some CWE types have more than one example. CWE-20 and CWE-611 have the most code samples with six code samples, followed by CWE-601 with five code samples, CWE-502, CWE-22 and CWE-327 have four code samples.

ID: The ID column describes the issued CWE ID along with the origin of that CWE and its numeration starting from 1.

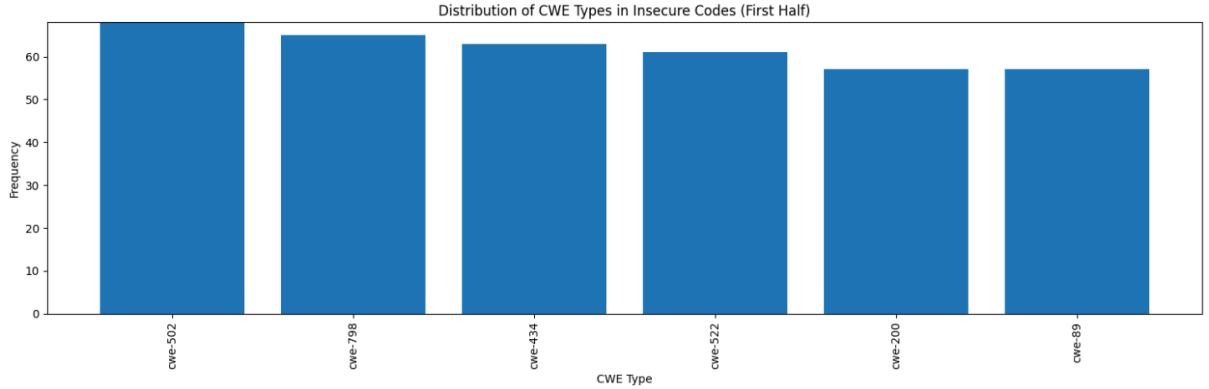
Prompt: The prompt column contains the CWE-related prompts for GitHub Copilot.

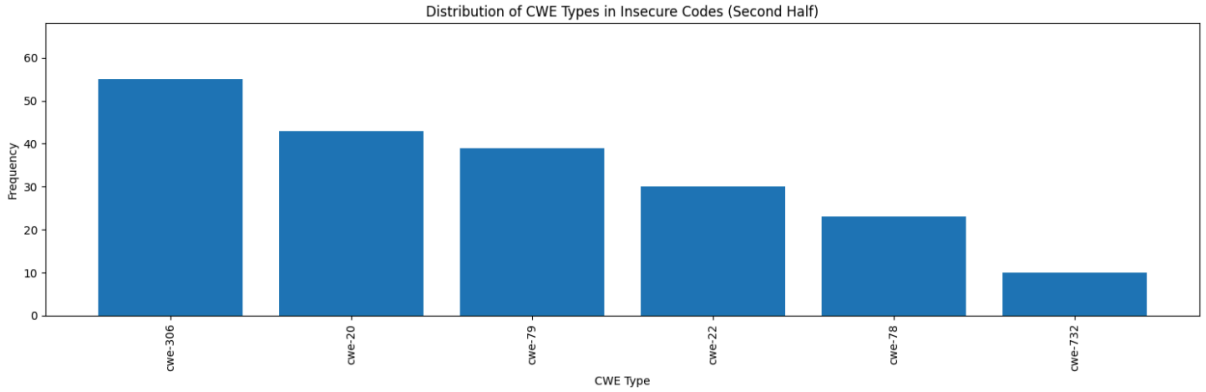
Insecure_code: The Insecure_code column contains sample vulnerable codes of the respective CWE.



Copilot CWE Scenarios Dataset

This dataset does not have a specific name. Authors created 89 different prompt scenarios and collected 1689 Github Copilot generated code outputs with extra 25 generated outputs for each scenario. The experiments and designed prompts fall into 3 categories. First DOW(Diversity of Weaknesses), in this part they wanted to test as much as possible different CWE types and conclude about the comparisons. They experimented with 18 different CWE types collecting 513 C programs and 571 Python programs. In the second part DOP(Diversity of Prompts) they wanted to see the results in changes of Copilot outputs when prompts are changed a little but with the same scenario. They stuck in CWE 89(SQL Injection) weakness type for this and collected 407 programs. In the last part DOD(Diversity of Domain) they observed the results when another type of programming was included, namely Verilog.With 18 different scenarios they collected 198 programs. Some results were discarded because Verilog is a more niche programming language and Copilot struggled to generate feasible results.





The part we will use in our research is the Diversity of Weaknesses. The most prominent part is the Python code consisting of 571 python code snippets in total. The analysis of the frequencies of CWE types for that part is above. The most seen CWE type is the CWE-502 by the authors.

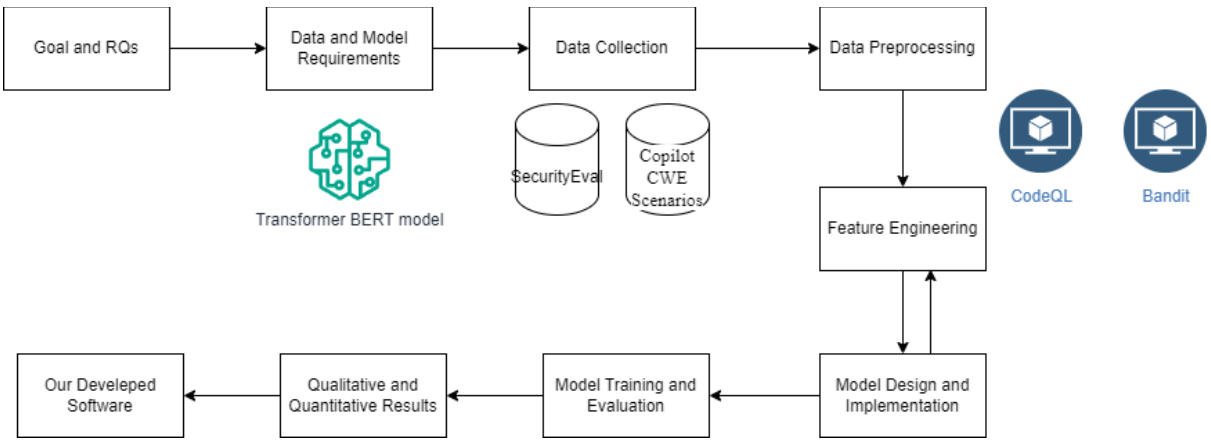
METHODOLOGY

In order to frame our research in line with the goal stated above, we raised the following research questions (RQs):

- Are Github Copilot-generated codes vulnerable or not
- Can BERT-based models accurately detect and classify vulnerabilities in AI-generated code, and how does their performance vary with different datasets and augmentation techniques?

We conducted a comprehensive literature review on code vulnerability and security. The existing literature on code security examines various methods for detecting and preventing vulnerabilities, as well as analyzing the effectiveness of these methods. We aim to utilize an artificial intelligence model, BERT (Bidirectional Encoder Representations from Transformers), to identify code vulnerabilities. To assess the applicability and performance of the BERT model, we investigated the necessary datasets and their characteristics for training and testing the model. The literature review helped us determine the required data volume, data types, and the parameters essential for successfully training the BERT model in the context of code security.

The model design and implementation phase involved several critical steps to ensure accurate detection of code vulnerabilities. We began by preprocessing the datasets to standardize and clean the code samples, removing any irrelevant or redundant information. We utilized two datasets specifically curated for this task, providing the necessary inputs for the BERT model. The model was fine-tuned for code vulnerability detection. Following training, we evaluated the model using metrics like accuracy, precision, recall, and F1-score to ensure its reliability and effectiveness. The outputs of the model were then analyzed.



- **Results**

CodeQL found 47 vulnerabilities in 41 out of 121 different scripts generated by GithubCopilot, with 19 different CWE types. Some code has more than one type of vulnerability. The distribution of the 19 different types of vulnerabilities is shown below. CWE-79 vulnerability was found in 10 different codes, followed by CWE-22 in 8 codes, and CWE-601 in 5 different codes.

Bandit identified a total of 53 security vulnerabilities across 51 out of 121 distinct code snippets generated by GithubCopilot, spanning 12 different CWE types. Multiple vulnerabilities were detected in some snippets. The distribution of these 12 different types of vulnerabilities is as follows: CWE-20 vulnerability was observed in the highest number of instances, present in 13 different snippets, followed by CWE-259 vulnerability found in 12 snippets, and CWE-327 type detected in 6 distinct snippets.

When merging the results from both CodeQL and Bandit analyses, it was found that across 121 distinct code snippets generated by GithubCopilot, a total of 93 security vulnerabilities were detected in 73 of them, spanning 24 different CWE types. Some snippets exhibited multiple types of vulnerabilities. The distribution of these 24 different vulnerability types is outlined below. The most prevalent vulnerability was CWE-20, observed in 14 distinct snippets, followed by CWE-259, identified in 12 snippets, and CWE-79, found in 10 snippets.

CWE Types	CodeQL	Bandit	Merged
CWE-20	1	13	14
CWE-22	8	2	9
CWE-78	1	5	5
CWE-79	10	0	10
CWE-89	0	1	1
CWE-90	1	0	1
CWE-94	2	2	4
CWE-116	1	0	1
CWE-117	1	0	1
CWE-209	1	0	1
CWE-215	1	0	1
CWE-259	0	12	12
CWE-319	0	1	1
CWE-327	3	6	7
CWE-330	0	2	2
CWE-377	2	3	3

CWE-400	0	3	3
CWE-502	2	3	4
CWE-601	5	0	5
CWE-611	2	0	2
CWE-730	2	0	2
CWE-776	1	0	1
CWE-798	1	0	1
CWE-918	2	0	2
Total	47	53	93
Securıty Eval Dataset Result			

CodeQL found 403 vulnerabilities in 320 out of 571 different scripts generated by GithubCopilot, with 13 different CWE types. Some code has more than one type of vulnerability. The distribution of different types of vulnerabilities is shown below. CWE-89 vulnerability was found in 148 different codes, followed by CWE-327 in 61 codes, and CWE-22 in 53 different codes.

Bandit identified a total of 832 security vulnerabilities across 491 out of 571 distinct code snippets generated by GithubCopilot, spanning 10 different CWE types. Multiple vulnerabilities were detected in some snippets. The distribution of these 10 different types of vulnerabilities is as follows: CWE-78 vulnerability was observed in the highest number of instances, present in 377 different snippets, followed by CWE-259 vulnerability found in 222 snippets, and CWE-89 type detected in 165 distinct snippets.

When merging the results from both CodeQL and Bandit analyses, it was found that across 571 distinct code snippets generated by GithubCopilot, a total of 1042 security vulnerabilities were detected in 513 of them, spanning 17 different CWE types. Some snippets exhibited multiple types of vulnerabilities. The distribution of these 17 different vulnerability types is outlined below. The most prevalent vulnerability was CWE-78, observed in 377 distinct snippets, followed by CWE-259, identified in 222 snippets, and CWE-89, found in 170 snippets.

CWE Type	CodeQL	Bandit	Merged
CWE-20	1	22	23
CWE-22	53	7	60
CWE-78	20	377	377
CWE-79	52	0	52
CWE-89	148	165	170
CWE-94	0	2	2
CWE-209	4	0	4

CWE Type	CodeQL	Bandit	Merged
CWE-259	0	222	222
CWE-312	14	0	14
CWE-327	61	28	61
CWE-377	0	5	5
CWE-502	20	0	20
CWE-601	19	0	19
CWE-703	0	2	2
CWE-730	1	0	1
CWE-732	3	2	3
CWE-798	7	0	7
Total	403	832	1042
Copilot CWE Scenarios Dataset Results			

As a result of the given prompts, Copilot produced the most vulnerable codes of the types CWE-20, CWE-79, CWE-259, CWE-78, CWE-89, CWE-327 and CWE-22. 9 of the emerging vulnerability types are included in the "2023 CWE Top 25 Most Dangerous Software Vulnerabilities[16]" list. These vulnerability types are shown in red columns in the table below.

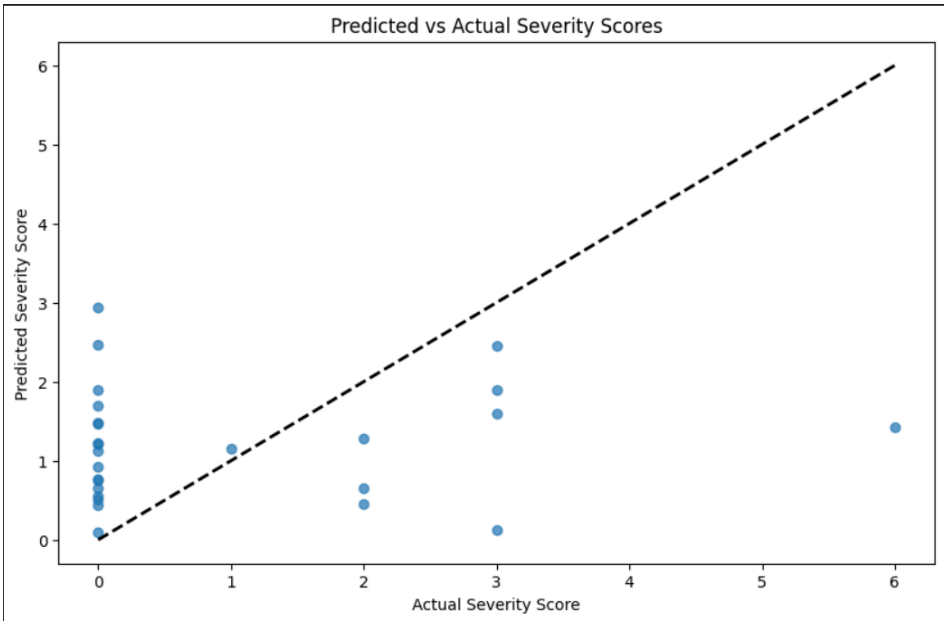
Rank	CWE Id	Weakness Name
1	CWE-787	Out-of-bounds Write
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
4	CWE-416	Use After Free
5	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
6	CWE-20	Improper Input Validation
7	CWE-125	Out-of-bounds Read
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
9	CWE-352	Cross-Site Request Forgery (CSRF)
10	CWE-434	Unrestricted Upload of File with Dangerous Type
11	CWE-862	Missing Authorization
12	CWE-476	NULL Pointer Dereference
13	CWE-287	Improper Authentication
14	CWE-190	Integer Overflow or Wraparound

15	CWE-502	Deserialization of Untrusted Data
16	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')
17	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer
18	CWE-798	Use of Hard-coded Credentials
19	CWE-918	Server-Side Request Forgery (SSRF)
20	CWE-306	Missing Authentication for Critical Function
21	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization
22	CWE-269	Improper Privilege Management
23	CWE-94	Improper Control of Generation of Code ('Code Injection')
24	CWE-863	Incorrect Authorization
25	CWE-276	Incorrect Default Permissions
2023 CWE Top 25 Most Dangerous Software Weaknesses		

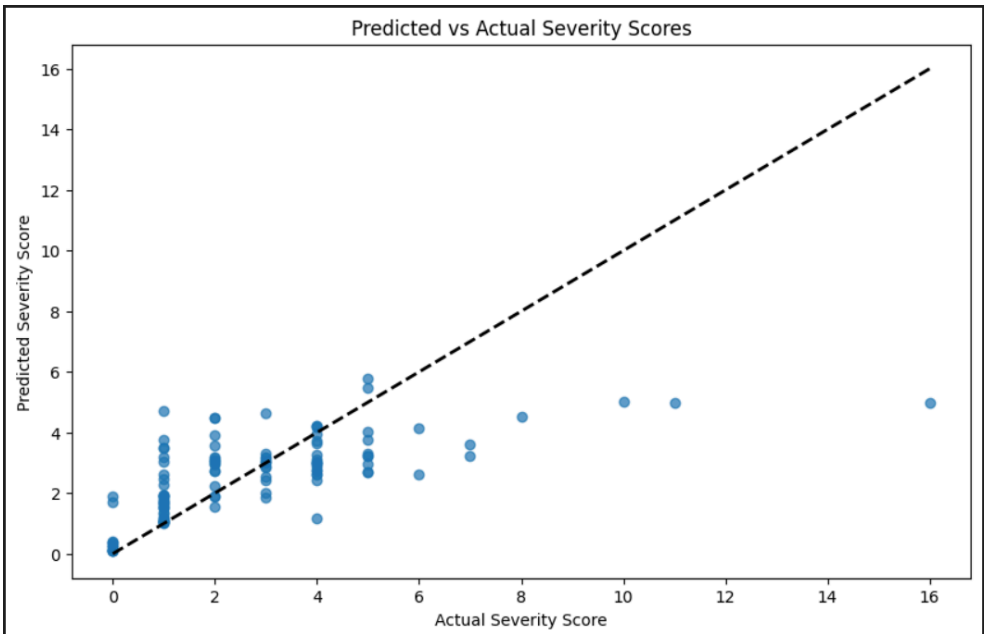
Code Snippet	Severity Level and Count	Severity Score
Example Input 1	0 High, 2 Medium, 1 Low	2*2+1*1 = 5
Example Input 2	1 High, 1Medium, 0 Low	1*3+1*2 = 5
…		

As methodology required, the next step was to use CodeBert to get embeddings for the datasets that describe the features of each code snippet. After extracting the embeddings, we feed the embeddings as feature vector and Bandit results of each Python code as ground truth to RandomForestRegressor. We defined a scoring system for vulnerability. For each low severity code in the Bandit we added 1 point to the total score , for medium severity we added 2 points and the high severity we added 3 points. The results of regression did not indicate strong results of how our methodology performed. We got a Mean Squared Error(MSE) of around 2.7. The regressor plot results are given below for each Dataset.

Security Eval Dataset:

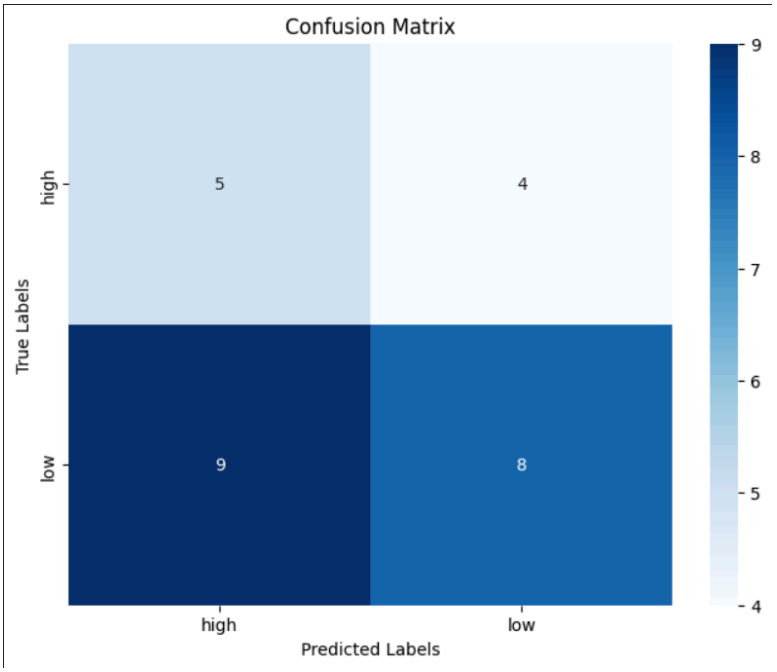


Copilot CWE Scenarios Dataset:

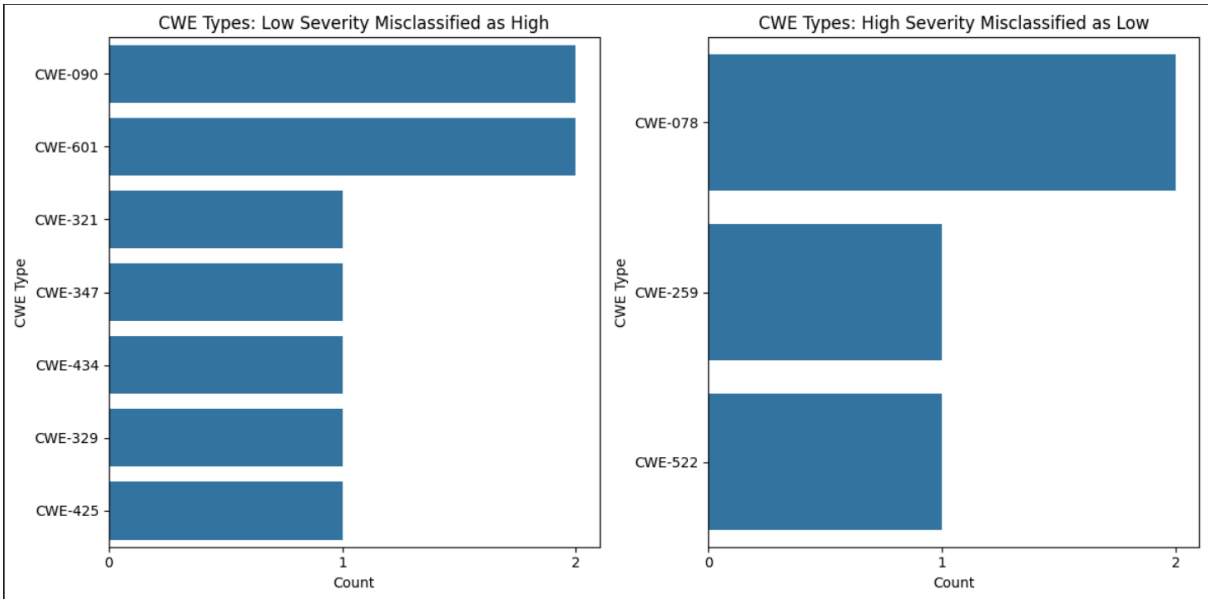


After this we need to classify the vulnerability scores as vulnerable or not. For this we feed the embeddings feature and Bandit scores to RandomForestClassifier. We created the results as the confusion matrices. The results of the classifier are given below for both dataset.

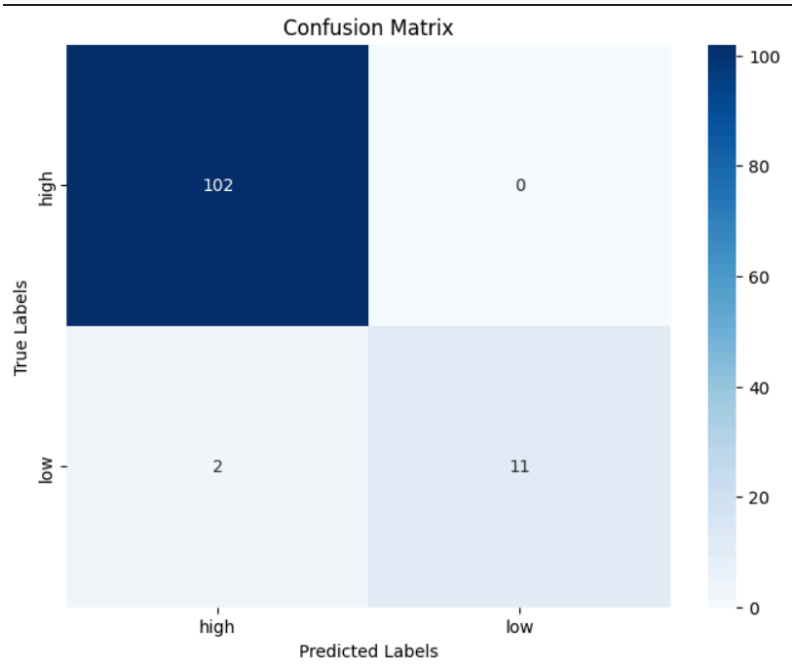
Security Eval Dataset:



The first dataset has resulted in many wrong classifications because of the lack of data. The mostly classified CWE types are:

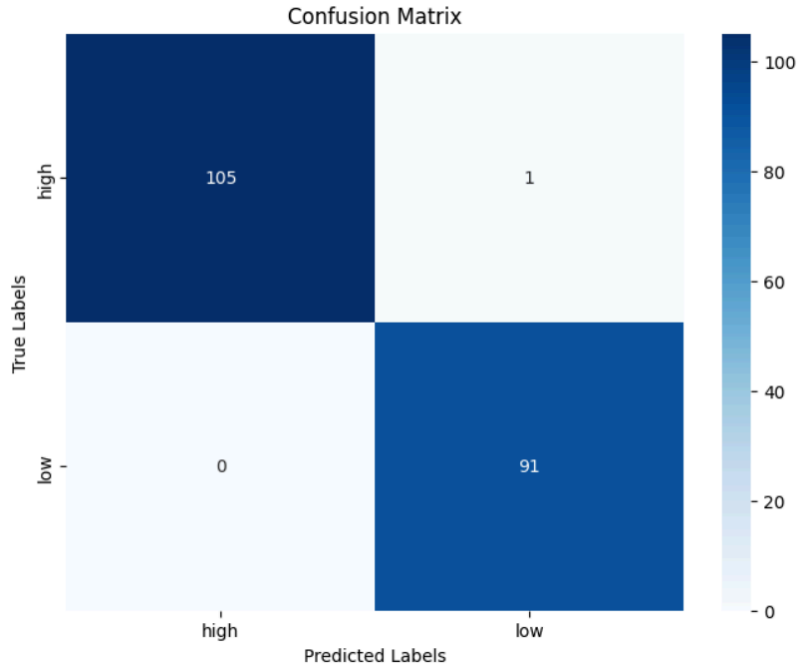


Copilot CWE Scenarios Dataset:

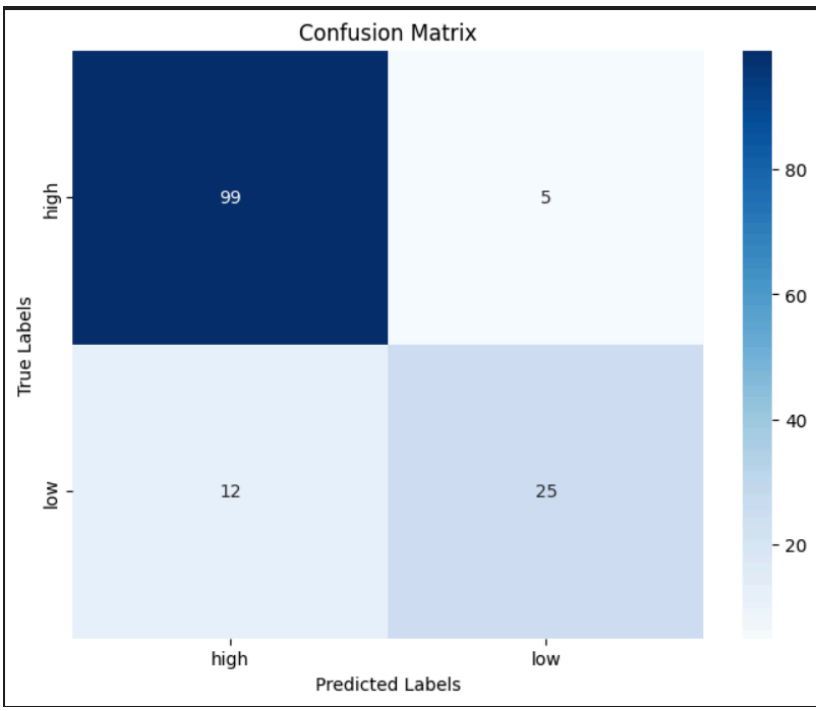


This confusion matrix resulted in 2 wrong classifications. The wrongly classified CWE types are CWE-522 and CWE-79.

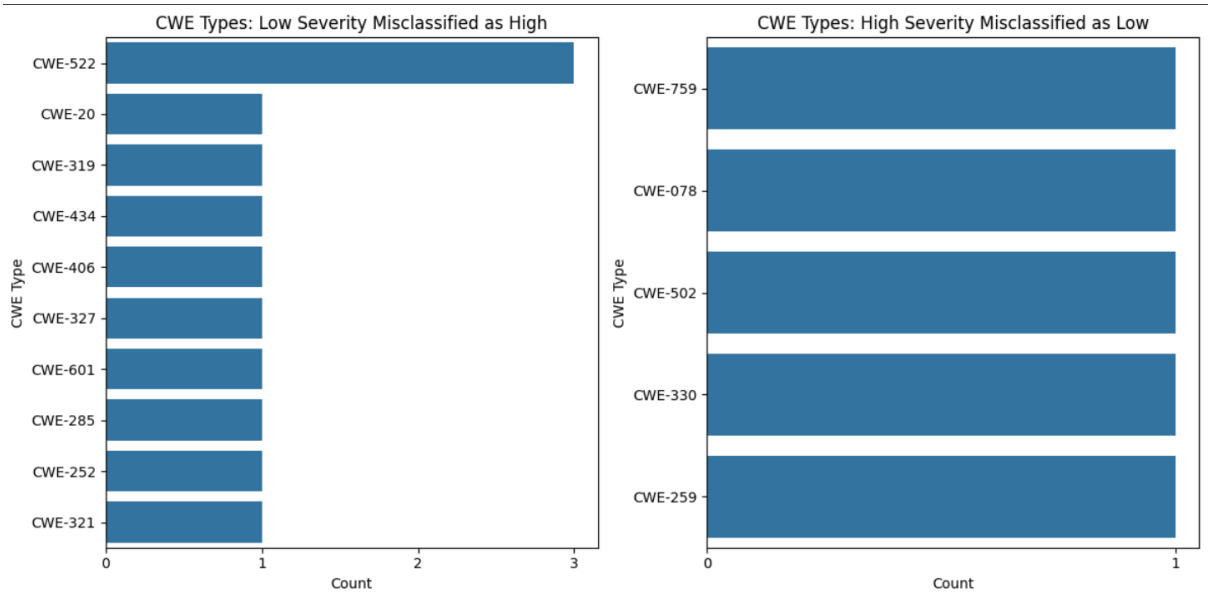
As we can see because of the more data in the second dataset, we were able to get a better classification result.
We applied an augmentation technique **Smote** to the second dataset to see if we could obtain a better result. The resulted confusion matrix is below:



We needed to see how our methodology worked on mixed code snippets. For this we concatenated both datasets and looked at the results.



In a mixed version we achieved 88% accuracy. However we suspect the misclassifications result highly from the first dataset inputs. And the misclassifications are below:



Our total evaluation results are stated in the given table.

	Precision	Recall	Accuracy	F1 Score
Security Eval Dataset	0.36	0.56	0.58	0.48
Copilot CWE Scenarios Dataset	0.93	1.0	0.94	0.96
Copilot CWE Scenarios Dataset, SMOTED	1.0	0.99	0.99	1.0
Security Eval & Copilot CWE Scenarios	0.89	0.95	0.88	0.92

Discussion

Our study reveals significant vulnerabilities in code generated by GitHub Copilot. The static analysis performed using CodeQL and Bandit tools identified a variety of security issues, underscoring the need for careful scrutiny when utilizing AI-assisted code generation tools. The findings indicate that Copilot-generated code frequently includes vulnerabilities such as CWE-20 (Improper Input Validation), CWE-259 (Use of Hard-coded Password), and CWE-79 (Improper Neutralization of Input During Web Page Generation). These vulnerabilities could lead to serious security breaches if not properly mitigated.

The automated nature of code generation tools may inadvertently introduce vulnerabilities or propagate insecure coding practices, as the underlying machine learning models might not always adhere to best practices for secure coding. Second, the reliance on these tools can lead to a diminished understanding of the code among developers, potentially resulting in oversight of critical security flaws. Finally, the integration of such tools into the development workflow necessitates an evaluation of their compliance with existing security standards and regulatory requirements.

Furthermore, the prevalence of vulnerabilities highlights the limitations of current AI models in understanding and applying secure coding practices. Despite Copilot's potential to enhance productivity, our results suggest that developers should not solely rely on AI-generated code without thorough review and testing. The study also points out that the effectiveness of Copilot in generating secure code varies with the programming language and the specificity of the prompts provided.

Conclusion

In conclusion, while GitHub Copilot represents a significant advancement in AI-assisted software development, it also introduces notable security risks. Our research indicates that a substantial proportion of Copilot-generated code contains vulnerabilities that could compromise the security of software systems. Developers must exercise caution and implement rigorous testing and validation procedures when incorporating AI-generated code into their projects.

The study underscores the importance of continuing to refine AI models to better adhere to secure coding standards. Future research should focus on improving the security awareness of these models and exploring additional methods to mitigate the risks associated with AI-generated code. By addressing these challenges, we can better harness the benefits of AI in software development while minimizing its potential drawbacks.

- ## References

[1] I. Sommerville, *Software engineering*. Pearson, 2011.

[2] OpenAI, "Chatgpt: Optimizing language models for dialogue." Accessed: Jan. 02, 2024. [Online]. Available: https://openai.com/blog/chatgpt/

[3] GitHub, "GitHub Copilot · Your AI pair programmer." Accessed: Jan. 02, 2024. [Online]. Available: https://copilot.github.com/

[4] E. Nijkamp *et al.*, "CODEGEN: AN OPEN LARGE LANGUAGE MODEL FOR CODE WITH MULTI-TURN PROGRAM SYNTHESIS." [Online]. Available: https://github.com/salesforce/jaxformer

[5] A. Moradi Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, and Z. M. (Jack) Jiang, "GitHub Copilot AI pair programmer: Asset or Liability?," *Journal of Systems and Software*, vol. 203, Sep. 2023, doi: 10.1016/j.jss.2023.111734.

[6] B. Yetiştiren, I. Özsoy, M. Ayerdem, and E. Tüzün, "Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT," 2023.

[7] Pearce Hammond, Ahmad Baleegh, Tan Benjamin, Dolan-Gavitt Brendan, and Karri Ramesh, “Asleep at the Keyboard? Assessing the Security of GitHub Copilot’s Code Contributions,” in *IEEE Symposium on Security and Privacy (SP)*, 2022.

[8] M. L. Siddiq and J. C. S. Santos, “SecurityEval dataset: Mining vulnerability examples to evaluate machine learning-based code generation techniques,” in *MSR4P and S 2022 - Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security, co-located with ESEC/FSE 2022*, Association for Computing Machinery, Inc, Nov. 2022, pp. 29–33. doi: 10.1145/3549035.3561184.

[9] S. Haque, Z. Eberhart, A. Bansal, and C. McMillan, “Security Weaknesses of Copilot Generated Code in GitHub,” in *IEEE International Conference on Program Comprehension*, IEEE Computer Society, 2022, pp. 36–47. doi: 10.1145/nnnnnnn.nnnnnnn.

[10] O. Asare, M. Nagappan, and N. Asokan, “Is GitHub’s Copilot as bad as humans at introducing vulnerabilities in code?,” *Empir Softw Eng*, vol. 28, no. 6, Nov. 2023, doi: 10.1007/s10664-023-10380-1.

[11] GitHub, “GitHub CodeQL”, Accessed: Jan. 02, 2024. [Online]. Available: https://codeql.github.com/

[12] H. Hajipour, K. Hassler, T. Holz, L. Schönherr, and M. Fritz, “CodeLMSec Benchmark: Systematically Evaluating and Finding Security Vulnerabilities in Black-Box Code Language Models,” Feb. 2023, [Online]. Available: http://arxiv.org/abs/2302.04012

[13] PyCQA, “Bandit GitHub Repository”, Accessed: Jan. 02, 2024. [Online]. Available: https://github.com/PyCQA/bandit

[14] F. Feng, et al., "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," arXiv preprint arXiv:2002.08155, 2020. Available: https://arxiv.org/abs/2002.08155

[15] Microsoft."CodeBERT GitHub Repository." GitHub, 2024. https://github.com/microsoft/CodeBERT

[15] OWASP, “Vulnerabilities | OWASP Foundation”, 2024. https://owasp.org/www-community/vulnerabilities/

[16] MITRE. "2023 CWE Top 25 Most Dangerous Software Weaknesses." [Online]. Available: https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html. [Accessed: Jun. 2, 2024]