DAT300 - Data-driven support for cyber-physical systems

Generative AI and Secure Coding

A Deep-Dive into the Capabilities of Generative AI for Programming and Security

Deaa Khankan
Omar Sulaiman
Yenan Wang

Chalmers University of Technology

October 22, 2023

Abstract

This report presents an evaluation of the security aspects of AI assistants, specifically focusing on code generation capabilities. Two different experiments were conducted to assess the security performance of AI models like ChatGPT, with a primary goal of understanding their strengths and weaknesses in identifying vulnerabilities.

The first experiment employed AI tools and a fuzzer to evaluate the security of code snippets found on stack overflow. The results revealed that while the fuzzer exhibited a commendable ability to detect vulnerabilities, ChatGPT also demonstrated efficiency for identifying issues, albeit not as comprehensively. This comparison laid the foundation for assessing ChatGPT's security capabilities.

The second experiment explored the influence of prompts on the security of AI-generated code. By requesting AI tools to produce code snippets with varying prompts, the study uncovered that these tools often generated code with vulnerabilities. The experiment pointed out the tool's potential capabilities of assessing the security of code snippets, and the impact of formulating good prompts.

Overall, this report underscores the importance of evaluating AI tools such as ChatGPT and GitHub Copilot for their security capabilities. These findings show that these AI tools generally do not yield great results at generating secure code depending on the prompts used, nor are they accurate at detecting security vulnerabilities in code. In conclusion, while AI tools might be the future for programming, more research and advancement is needed in this area.

Contents

1 Introduction **1**
- 1.1 Purpose . . . 1
- 1.2 Report Structure . . . 1

2 Technical Background **3**
- 2.1 Vulnerabilities . . . 3
  - 2.1.1 Memory Leak . . . 3
  - 2.1.2 Obsolete Functions . . . 3
  - 2.1.3 Buffer Overflow . . . 3
- 2.2 Tools . . . 3
  - 2.2.1 Fuzzer . . . 3
  - 2.2.2 Static Analyzer . . . 4

3 Experiment Procedures **5**
- 3.1 Experiment 1: Evaluation of Vulnerable Code . . . 5
  - 3.1.1 Workflow of Experiment 1 . . . 5
  - 3.1.2 Environment of Experiment 1 . . . 6
  - 3.1.3 Usage of Fuzzer . . . 8
- 3.2 Experiment 2: Evaluation of AI-Generated Code with Varying Prompts . . . 8
  - 3.2.1 Workflow of Experiment 2 . . . 9
  - 3.2.2 Environment of Experiment 2 . . . 10
- 3.3 Delimitations . . . 11

4 Result **13**
- 4.1 Result of Experiment 1 . . . 13
- 4.2 Result of Experiment 2 . . . 14

5 Discussion **17**
- 5.1 Discussion of Experiment 1 . . . 17
- 5.2 Discussion of Experiment 2 . . . 18
- 5.3 Threats to Validity . . . 18
- 5.4 Future Work . . . 19

6 Challenges **20**

7 Related Work **21**

8 Conclusion **22**

References **23**

A Code Snippets **26**

1 Introduction

Generative AI is a broad category of artificial intelligence that refers to deep-learning models capable of generating various kinds of content, depending on the training dataset [1]. Generative AIs can generate realistic images, converse in natural languages, write valid program code, and more [2]–[4]. In particular, the ability of generative AIs to write code has many implications for developers in the fields of technology, where programming-related tasks are ubiquitous. Therefore, the rise of generative AI tools has the potential to replace or enhance the traditional ways developers operate. These kinds of generative AI tools used to assist developers in coding can also be referred to as coding assistants. However, in this report, we use *AI tool* as a collective terminology for any tool empowered by a generative AI that is capable of writing code.

Many developers are already using AI tools in their workflow. According to a Stack Overflow developer survey, it was revealed that 86% of professional developers use AI tools to assist them in code writing [5]. While there are numerous benefits to using generative AIs for coding—namely, it can improve productivity and speed up the overall development process—nevertheless, the code generated by AI tools is not always of acceptable quality. In particular, it has been shown that AI tools tend to generate erroneous code [6], [7] and usages of such tools can also lead to an increased number of security vulnerabilities [8]–[11]. Thus, it is imperative to be cautious when using AI tools to help with coding.

The popularization of such tools is therefore concerning, with regard to the potential security vulnerabilities developers might introduce by inappropriately using AI coding tools. In a recent survey organized by GitHub [12], it was shown that 92% of US-based developers use AI coding tools throughout the day. However, considering how developers who utilize AI tools are more inclined to believe the generated code to be secure [8], it might do more harm than good to use these tools blindly when coding. This incentivizes the motivation of this report, which is to research AI-generated code from a security perspective and examine if AI coding tools can be relied on.

1.1 Purpose

In this report, we aim to further explore the extent of code generation capabilities of generative AI tools such as ChatGPT [13] and GitHub Copilot [4], by focusing on security. Therefore, this report will not include an analysis of other code qualities such as functional correctness, readability, scalability and so on.

To reach the goal, we have devised two primary objectives which we intend to accomplish. The objectives are the following:

1. **Evaluation of Vulnerable Code Snippets** - Investigate if an AI assistant can correctly evaluate the security aspect of a vulnerable code snippet. Analyze if the evaluation is accurate based on the errors detected in the given code.

2. **Evaluation of AI-Generated Code Snippets With Varying Prompts** - Investigate the impact of different prompts on the security of AI-generated code snippets. Request the AI tools to produce code snippets with varying prompts and analyze how these variations influence the security characteristics of the generated code.

1.2 Report Structure

In section 2, the concepts and important tools used in the experiments are presented and described briefly. In section 3, how the two experiments are conducted is explained in detail.

Additionally, the delimitations for the experiments are also identified and presented. In section 4 presents the results obtained from the experiments in a clear manner. In section 5, the results are analyzed to highlight interesting findings, discuss potential threats to validity, and what other work that could be done based on this report. In section 6, we describe the difficulties we have experienced during the experiments and how we overcame those obstacles. Lastly, in section 7, other studies similar to ours will be discussed and presented.

2 Technical Background

In this section, the concepts that are crucial for a sound understanding of the experiments are described and explained in detail.

2.1 Vulnerabilities

In this section, vulnerabilities that are relevant to the experiments will be presented.

2.1.1 Memory Leak

Memory leak is a vulnerability caused by not releasing allocated memory after it has been used, which in time leads to consuming the entire memory. The vulnerability can be triggered by either an attacker attempting to conduct a Denial of Service (DoS) attack or by improper handling of allocated memory [14]. In both cases, the consequences are the same, a crashed memory with not enough memory for the system to run properly. A trivial example of memory leak is allocating memory using the `malloc()` function and, subsequently, never freeing the allocated space [15].

2.1.2 Obsolete Functions

An obsolete function is, as the name implies, a function that is no longer supported or maintained. Using obsolete functions is not always as dangerous as it sounds, depending on the usage, as it does not always lead to critical issues. Functions in various programming languages evolve over time due to many reasons such as language advancement, finding a better way to solve a problem or having a better understanding of how the operation of the function should be done. Using obsolete or deprecated functions may lead to security breaches, quality drops, or not utilizing the hardware in the best way possible [16].

2.1.3 Buffer Overflow

Buffer overflow is a vulnerability where an adversary can overwrite the program's memory, causing it to behave incorrectly. The attacker can change the executing path of the next function by introducing a malicious code as an input to an application on the target system. If the application does not carefully validate the input and does not follow the best security practices such as using deprecated functions, it may be a serious entry point of attack. Buffer overflow occurs when a malformed input overwrites executable code to introduce new malicious code, which gives the attacker unauthorized privileges [17].

2.2 Tools

By taking the experiment design into account, the following tools were considered to achieve the desired results.

2.2.1 Fuzzer

Fuzzer refers to automated security testing software which is used to test an application by injecting various inputs and trying to get unexpected behavior such as crashes or information leakage [18]. These behaviors in the application usually indicate the existence of security and/or quality issues. To have successful and reliable fuzzer testing, a file with well customized input is sometimes needed, so that the fuzzer can have an entry point for its input generation [19].

Some fuzzers can also be instructed to use sanitizers [20]. A sanitizer is a tool that can track code execution and detect suspicious behaviors during runtime [21]. The AddressSanitizer is

an example of a sanitizer [22]. AddressSanitizer is capable of detecting buffer overflows as well as memory leaks in a program. A fuzzer equipped with sanitizers can therefore detect more varieties of errors and vulnerabilities.

2.2.2 Static Analyzer

A static analyzer is a tool capable of examining source code statically for security and/or quality issues, without executing it. A static analyzer improves the quality and reliability of the software, since debugging is done in the early phases of the system development. The approach of early debugging leads to recourse savings and reduces the time-to-market since trivial issues are discovered early. The static analyzing process is usually conducted using automated tools which have various protocols and algorithms applied to the code to detect unwanted behavior [23]. For instance, CodeQL is an example of such a tool [24].

3 Experiment Procedures

In order to achieve the objectives previously defined in section 1.1, two experiments will be conducted on the AI tools that are commonly used to aid programming tasks. Each experiment will therefore be planned in such a way that some conclusions can be drawn to answer the respective objectives.

A summary of the two experiments can be seen in the following list.

Experiment 1 – In the first experiment, we test how well AI tools understand security in code by asking them to evaluate a number of vulnerable code snippets written by human programmers. The evaluations made by the AI tools are then manually inspected for correctness.

Experiment 2 – In the second experiment, we test how well AI tools generate secure code by prompting them to write a number of programs. Then the security of the programs is assessed using some security analysis tools as a formal assessment. In addition, we also ask the AI tools to evaluate the generated code themselves as an informal assessment.

3.1 Experiment 1: Evaluation of Vulnerable Code

In the first experiment, the goal is to test how well AI tools analyze a vulnerable code snippet. For the purpose of this project, a vulnerable code snippet is defined as any code that behaves unexpectedly or unpredictably given some malicious input when executed. For instance, a code snippet is vulnerable if it contains a buffer overflow vulnerability.

The core idea is that by having the AI tools analyze a vulnerable code snippet, we can then assess the correctness of the tools' evaluation. However, an objective assessment requires us to be able to prove that there exist some vulnerabilities. Therefore, we will be using a security testing tool to establish an objective reference point, which we can use to compare to the AI tools evaluations.

Due to limited time, it was decided to only focus on vulnerable code snippets. While it might have yielded more coverage to also analyze secure code snippets, we have deemed that it was not worth spending resources to verify if a code snippet is secure. On the other hand, it is simple to show that a code snippet is vulnerable. Namely, if a security analysis/testing tool shows the existence of a vulnerability, then the code snippet is more than likely to be vulnerable.

3.1.1 Workflow of Experiment 1

The general workflow of the first experiment can be seen in figure 1. The first step is to collect appropriate code snippets from the internet. The code snippets are purposely selected to contain some form of vulnerabilities. Accordingly, no code snippet that seems to be functional and secure has been chosen for this experiment.

The code snippets will be searched within publicly available domains on the internet. One primary website to search for such code snippets is Stack Overflow [25], as many programmers tend to use it to ask for help with their code.

Once the code snippets have been collected, they will be tested using a fuzzer, as seen in figure 1. We choose to use a fuzzer in this experiment because it can provide us with a concrete input that breaks the program, which proves the program to be vulnerable. However, a fuzzer might not provide information of the exact vulnerability it triggered. Therefore, there will be some degree of manual analysis of the code snippet to ascertain the kind of vulnerability it contains.

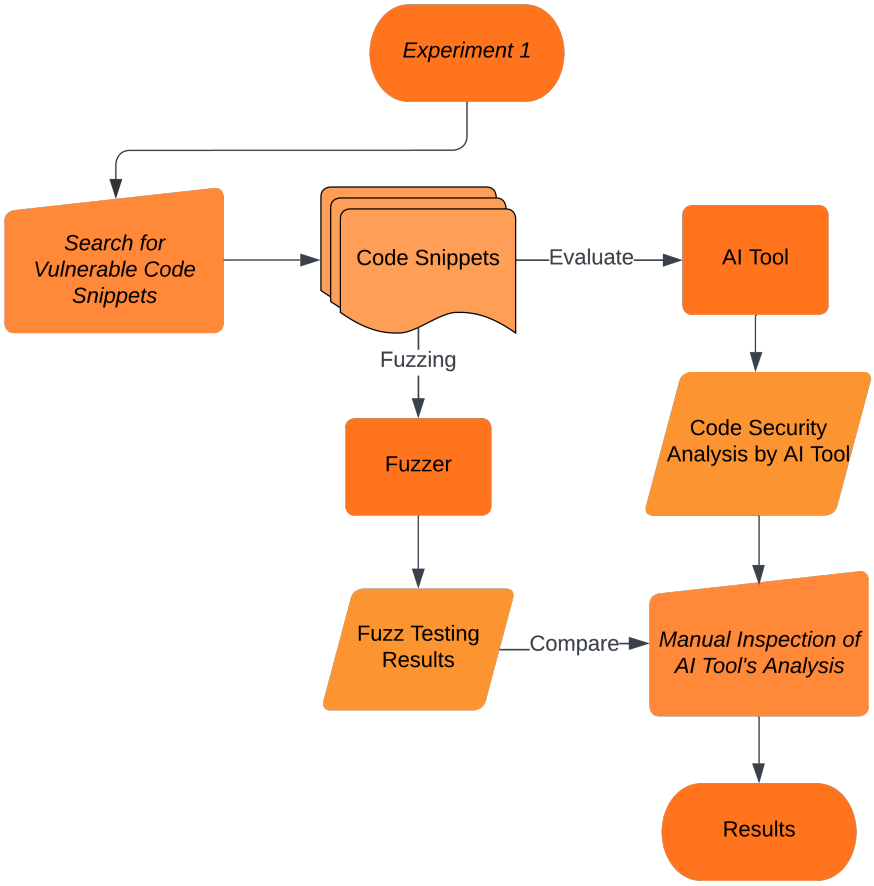

Figure 1: Workflow of the first experiment.

In parallel to the fuzzing step, the code snippets are also sent to the AI tool to be analyzed. The resulting analysis is finally manually inspected to determine the correctness of the analysis. Then the correctness is determined by checking the detected vulnerability in the analysis against the expected vulnerability using the information derived from the fuzzing results.

3.1.2 Environment of Experiment 1

In this section, we will present the tools and technologies that have been chosen to be used in the first experiment, as well as the number of code snippets we have obtained. The summarized content can be seen in table 1.

AI tool(s)	Fuzzer	Language	No. of code snippets
ChatGPT	AFL++	C	4

Table 1: The specifics of the tools and data used in the first experiment.

To begin with, the AI tool to be evaluated in the first experiment will be ChatGPT [13]. ChatGPT is a large language model driven technology which can be used in various ways such as answering questions, composing texts and generating code. ChatGPT is a natural language processing tool, which means that it can interact with humans and have human-like conversations.

ChatGPT is widely used in various fields including coding. The tool can analyze the code and give suggestions, as well as evaluate the security and the quality of the injected code. Given the popularity of ChatGPT, which has over 100 million users [26], it will be important to assess the code generated by ChatGPT even if only a small fraction of users uses it to write code.

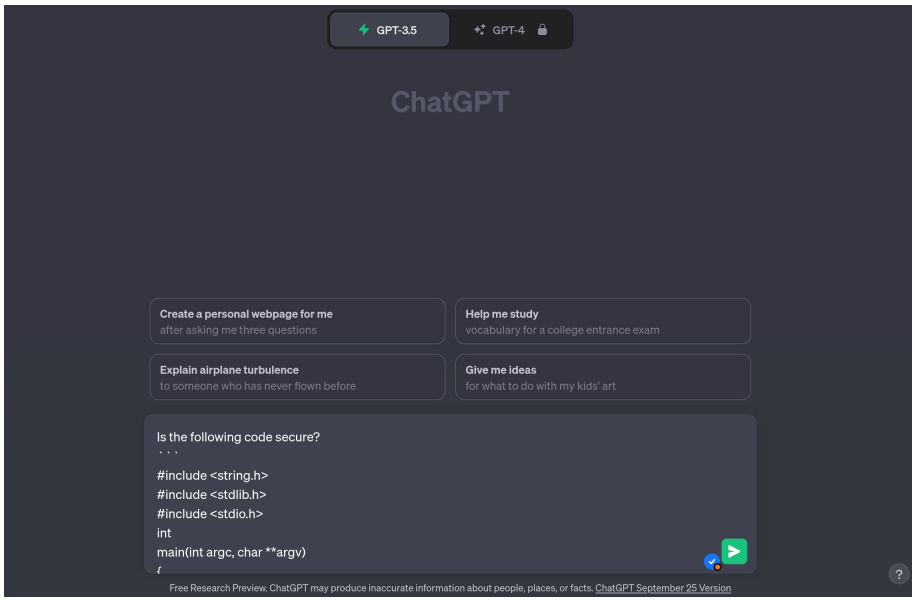

Figure 2: An example of how ChatGPT was prompted in the first experiment.

ChatGPT has a web interface which we will use to interact with it. Figure 2 demonstrate how ChatGPT will be prompted. In essence, we prompt ChatGPT with a question and the code snippet to be analyzed. The first response by ChatGPT is then collected and regarded as the resulting analysis.

The fuzzer to be used in this experiment is AFL++ [19]. AFL++ is an improved fork of the AFL fuzzer [27] developed by Google. AFL++ is actively supported and maintained. Its documentation is in-depth and makes it easy to set up even without prior fuzzing experience. Furthermore, it is primarily a source-code fuzzer (but it also supports binary-only fuzzing [28]), which is appropriate as we have the source code. AFL++ also supports fuzzing of C/C++ code natively, which aligns with our choice of programming language. For these reasons, AFL++ has been chosen as the fuzzer to be used in the first experiment.

It is important to be aware of the fact that fuzzers such as AFL++ requires a valid input to the program to be able to begin fuzzing, as aforementioned in section 2.2.1. A valid input in this case refers to any input that does not cause the program to crash or violate some sanitizer rules. Therefore, any program without a valid input is unable to be tested by the fuzzer. While that is a rare case, it nevertheless might occur. In those cases, the corresponding code snippets will be treated as special cases and handled on a case-by-case basis.

As seen in table 1, the programming language we use for the code snippets will be C [29]. C was chosen because it is generally regarded as a vulnerable programming language. As C is a relatively low-level language, and it requires the programmers to perform memory management manually, which can easily lead to a memory leak error even if the programmer only missed freeing a single memory allocation [30]. It is prone to buffer overflow vulnerabilities due to the existence of many deprecated insecure functions such as `gets()`, `strcpy()`, and `sprintf()`, to name a few [31]. In a study, it was shown that among 7 popular programming languages, open-source projects in C produces the most vulnerabilities [32]. Despite the problems with C,

over 19% of developers still uses it, according to a survey conducted by Stack Overflow [33]. The large amount of vulnerabilities in C combined with a high usage by developers makes it a good choice for assessing AI tools' security capability.

Lastly, it was surprisingly difficult searching for appropriate code snippets on the internet, namely code that can be tested using a fuzzer and contains some vulnerabilities. In the end, 4 code snippets in C have been collected from Stack Overflow, in which one of the snippet has been modified to read input from the standard input in order to be fuzzed. Code snippets used in both experiments can be found both in appendix A and on our GitHub repository[1].

3.1.3 Usage of Fuzzer

In this section, we will explain what configuration or parameters we have decided to use for AFL++. The usage of the fuzzer applies to both experiments.

To begin with, it is important to establish the fact that the code snippets that we are working with will not be full-sized programs. These codes are referred to as snippets because they are small and are often less than 100 lines long. The experiments utilize small code snippets because they are faster to process. Larger and more complex programs can take much longer time to test using security testing tools, and they also are much more mentally exhausting to evaluate manually. Furthermore, it is not necessary to have larger programs to show the concept of a vulnerability. Therefore, it was deemed that small code snippets were sufficient given the limitation of our experiments.

In general, AFL++ is executed for roughly 5 minutes for each code snippet. This time mark is determined through trial and error, as most crashes are usually found in the first minute for code snippets of this size. However, we let it run until the 5-minute mark, as a pessimistic approximation.

AFL++ is also configured to use sanitizers. For all code snippets, AdressSanitizer (ASAN) and UndefinedBehaviorSanitizer (UBSAN) are used to examine potential memory integrity violations and undefined behaviors in C [34]. Those issues may not crash the program, but they often lead to vulnerabilities that can be exploited, which is why these sanitizers are necessary.

3.2 Experiment 2: Evaluation of AI-Generated Code with Varying Prompts

The goal of the second experiment is to explore how changing the prompts to the AI tools affects the results. To achieve that, we plan to prompt the AI tools using a number of prompts, where each prompt is categorized as *causal*, *secure* or *insecure*.

In this experiment, a secure prompt is defined as any prompt that explicitly asks the AI tool to generate code to be secure. Conversely, an insecure prompt is a prompt that explicitly asks the AI tool to generate insecure code. On the other hand, a casual prompt is simply a prompt that does not specify the security of the code.

In order to have as many prompts as possible, the casual prompts that are possible to be converted into a secure or insecure prompt will be converted by adding either the word secure or insecure into the prompt text. Converting casual prompts can help to balance the dataset so that all three categories of prompts will have roughly the same distribution.

[1] https://github.com/yenanw/generative-ai-security-project

3.2.1 Workflow of Experiment 2

Firstly, the prompts for the AI tools will be devised as shown in figure 3. Some prompts are partially inspired by the dataset for security evaluation created by Siddiq and Santos [35], and the remaining prompts are derived from personal experiences of the authors.

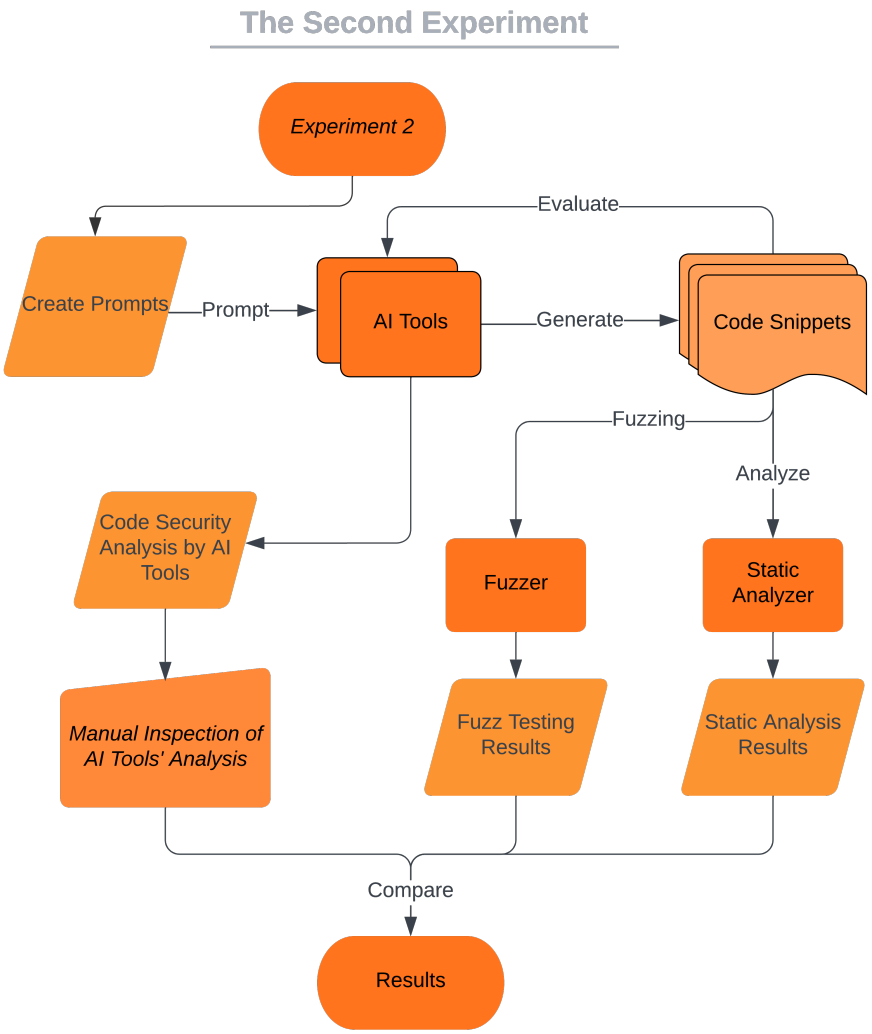

Figure 3: Workflow of the second experiment.

Subsequently, the prompts are sent to the AI tools to generate code snippets, which will be assessed for security. To assess the security of the code snippets, both a fuzzer and static analyzer will be used as a reference point. In this case, the primary goal is not to affirm that the code snippet contains some vulnerabilities, like in the first experiment. The goal of using both tools is to have potentially more coverage of the tested code, in order to reduce the amount of false negative reports.

To determine the security of a code snippet objectively, both results from the fuzzer and static analyzer will be taken into consideration. Specifically, we define a code snippet to be insecure if either the fuzzer detects at least one crash or the static analyzer reports at least one error. Conversely, a code snippet is secure if neither the fuzzer nor the static analyzer detects any crash or error.

In addition to the security testing and analysis, the code snippets are also sent to the AI tools for evaluation, similar to the first experiment. This serves as an extension, since the number of code

snippets collected in the first experiment is not quite satisfactory. Therefore, manual inspections are still necessary in order to determine the AI tools' stance on the security of a particular code snippet. The analysis will be categorized into three categories, namely secure, ambiguous, or insecure. An analysis of a code snippet will be concluded as secure/insecure if the phrasing of the analysis clearly implies that the AI tool believes the code snippet is secure/insecure. Otherwise, the analysis is concluded as ambiguous.

In the last step, the results obtained from the fuzzer and static analyzer will be used to determine whether the code snippets are secure or insecure. Then we will also investigate how the results compare to each other with different prompt categories. For instance, investigate if AI tools generate more secure code when explicitly asked to generate a secure code. Furthermore, the security analysis capability of the AI tools will also be explored by comparing the analysis by the AI tools to the results from fuzz testing and static analysis.

3.2.2 Environment of Experiment 2

Similar to section 3.1.2, this section presents the AI tools and security testing tools to be used in this experiment. Table 2 shows the summarized choices. Table 2 is similar to table 1 since most choices have remained the same. The second experiment is not distinct from the first experiment, and we have therefore made the same choices for AI tools, fuzzer and programming language with similar reasoning.

AI tool(s)	Fuzzer	Static analyzer	Language	No. of code snippets
ChatGPT, GitHub Copilot	AFL++	CodeQL	C	30

Table 2: The specifics of the tools and data used in the second experiment.

For experiment 2, we have opted to use C as the programming language. The prompt we construct will therefore be related to vulnerabilities common in C. The vulnerabilities we use are the following: buffer overflow (BO), obsolete functions (OF), and memory leak (ML). We focus on these vulnerabilities because they would manifest easily in C as a lot of the responsibility of memory management falls on the programmer.

It can be seen in table 2 that an additional AI tool, namely GitHub Copilot (henceforth referred to as Copilot for simplicity) [4], will also be evaluated in this experiment. Copilot is an AI tool which generates code suggestions by either auto-completing the already written code or by describing the desired application in a natural language. The tool will then analyze the text file and offer suggestions to construct what it was asked to. The quality of suggestions depends on how well-trained the tool is for a certain programming language. The reason to include Copilot in this experiment is due to the high amount of adoption of Copilot by organizations. Over twenty thousand organizations employ GitHub Copilot for Business [36]. Therefore, a significant amount of code generated by Copilot is going into production [36], which makes it crucial to ensure the quality of code generated by Copilot.

Unfortunately, Copilot was not included for analysis in the first experiment due to time constraints. However, by re-implementing the AI tool analysis step from the first experiment, we hope to nevertheless provide some insights into how well Copilot analyses code security.

In order to prompt Copilot, we will use Visual Studio Code (abbreviated VSCode) and the Copilot extension for VSCode [37]. To prompt Copilot, the prompt is sent to Copilot chat, similar to how ChatGPT was prompted. Copilot chat is an extension built for VSCode, which programmers can use to converse with Copilot. Figure 4 shows an example of the Copilot chat.

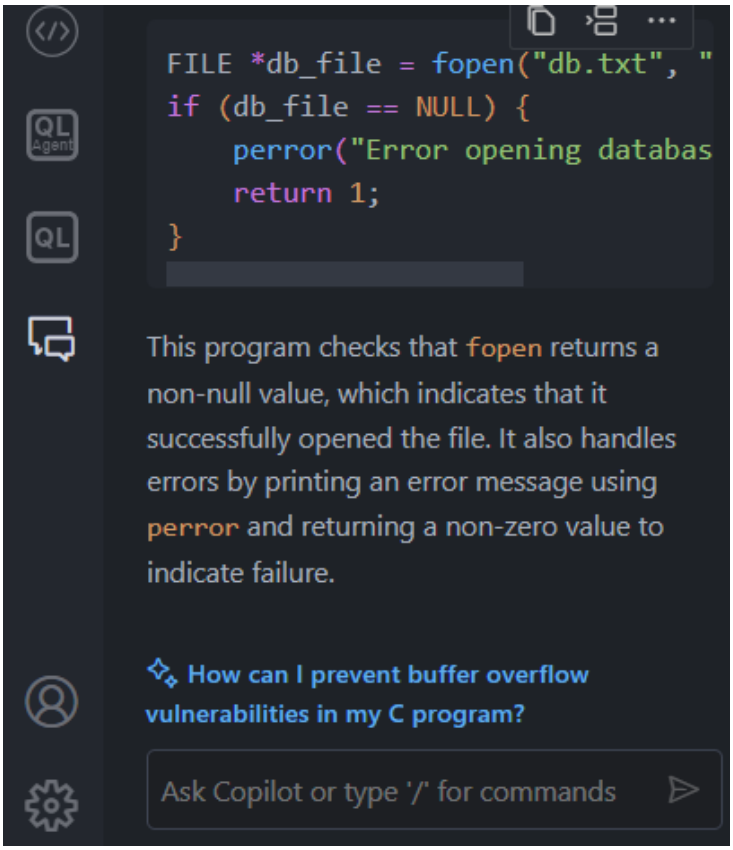

Figure 4: Copilot chat extension in VSCode.

As for the static analyzer previously mentioned in section 3.2.1, we have opted for CodeQL [24] as seen in table 2. For a small-scale project such as this, it is infeasible for us to use enterprise static analyzers such as SonarQube [38]. While these static analyzers are high quality, the free trials they offer is not sufficient for the duration of this project, not to mention that sometimes there is no option for a free trial [39]. CodeQL on the other hand is entirely free for public and local repositories. It is open-source and has decent documentation that allows for quick setup. Naturally, it also supports C/C++, among many other languages. For these reasons, we choose to use CodeQL as the static analyzer.

CodeQL comes with many built-in queries used to detect issues in code quality such as security, correctness, and readability [40]. Without going into details, when a query is performed on a code snippet, it will report a warning or an error on the code for anything that matches the query. There are some predefined query suites which, when executed, automatically apply a number of different queries on the code [41]. In our case, we use the `security-extended` query suite, as this particular query suite is focused on assessing security in the code [41].

Lastly, we have managed to generate 30 code snippets in total, whereas 11 of the code snippets are generated by casual prompts, 10 of the code snippets by secure prompts and 9 of the code snippets by insecure prompts.

3.3 Delimitations

In order to formally assess security, there needs to be an objective way to represent how secure a code snippet is. As aforementioned, for the purpose of objective assessment, we use **AFL++** [19] and CodeQL [24] to test programs for potential security vulnerabilities. Naturally, this means any limitations of these technologies will also be present in our experiments.

One main drawback of fuzz testing is the fact most fuzzers have no understanding of the code context and rely on semi-randomly generated data to test a program. Therefore, in order to find vulnerabilities, the program is exhaustively searched for some input that might cause a crash or rule violation. Depending on the size of the program, fuzz testing may be too time-consuming considering the limit of our resources. For that reason, the sizes of the code examples used in this project will be limited to 100 lines of code.

Furthermore, another disadvantage to using fuzzers is how fuzzers only provide incomplete vulnerability coverage. It is not guaranteed to find all vulnerabilities through fuzz testing. Certain attack types that do not cause the program to crash upon unexpected input are unlikely to be detected through fuzz testing. Therefore, the experiments conducted in this paper will not be specifically investigating vulnerabilities that are difficult to test using a fuzzer.

CodeQL static analyzer was used as a reference point in the second experiment. While static analyzers are usually efficient and scale well for larger programs, in some cases, the tool can yield false positive results since the tool may lack information about the dependencies and the flow of the source code [42]. Therefore, there is a risk for incorrect analyses of the code.

4 Result

This section presents the results gathered from the first and second experiment in detail. For experiment 1, a summary of the results from fuzzing is presented, followed by assessments of the security analysis generated by ChatGPT for each code snippet. For experiment 2, the results of code security testing by AFL++ and CodeQL is summarized and presented. Finally, the security analysis capability of ChatGPT and Copilot is presented, by demonstrating the number of correct evaluations the respective AI tools have provided.

To see the full list of code snippets, as well as the raw results such as AFL++ and CodeQL output and the evaluations by the AI tools, refer to our GitHub repository[1].

4.1 Result of Experiment 1

As it can be seen in table 3, we not only see the number of crashes found by AFL++, but the number of BO errors is also specified in the column `stack-buffer-overflow` errors. Observe that there are two versions of the code snippet 2, and the reason for this will be discussed later in section 5. However, snippet 2 and snippet 2a are fundamentally identical functionally, the only difference is that snippet 2a shows the errors that should have been present in snippet 2. Lastly, it was not possible to fuzz snippet 4 as this code snippet crashes on all input due to a logic error, which will discussed more in section 5. Overall, it can be stated that all code snippets have some sort of vulnerability in them.

Code snippet	Fuzzing time	Crashes found	`stack-buffer-overflow` errors
Snippet 1	5 min	3	3
Snippet 2	5 min	0	0
Snippet 2a	5 min	4	4
Snippet 3	5 min	1	0
Snippet 4	-	-	-

Table 3: Fuzz testing results from the first experiment.

In the first experiment, ChatGPT gave a correct analysis for two out of the four code snippets. The summary of the result of ChatGPT's analysis can be seen in table 4. The analysis are manually reviewed and controlled for correctness.

Correct	Partially correct	Incorrect
2	1	1

Table 4: Correctness evaluation of ChatGPT's analysis of the four code snippets.

For the first two code snippets, which can be found in appendix A, ChatGPT provided a rather comprehensive analysis of the problems in the respective code snippets. In both cases, ChatGPT has managed to successfully identify the source that is causing the BO vulnerability. Additionally, ChatGPT also provided some recommendations for how to improve the overall quality of the code. For code snippet 1, it mentioned the lack of input validation and the questionable function call of `mprotect`. For code snippet 2, ChatGPT actually detected how `buffer2` is written to but never used afterward. Observe that the fuzzer was not able to find any BO error, but it was pointed out by ChatGPT in table 3. The fuzzer was, however, able to find errors in the modified version of snippet 2. Otherwise, ChatGPT also provided practical advices on code

[1] https://github.com/yenanw/generative-ai-security-project

quality such as using constants rather than magic numbers and error handling. In conclusion, ChatGPT's analysis for the first two code snippets are correct and provides useful insights for potential improvements.

In ChatGPT's analysis of code snippet 3, it listed 8 points in regard to vulnerabilities and issues in the given snippet. In summary, ChatGPT identifies the usage of the vulnerable function `sprintf`, but it failed to see that the function is called with a literal string rather than something that can yield a string of dynamic length. Therefore, this particular usage of `sprintf` is technically safe and does not create a buffer overflow vulnerability, as the size of the literal string is smaller than the allocated buffer size. The real vulnerability here is located at line 19, where the buffer `lbuf` is used directly as the argument to `printf`, which is an undefined behavior. This time, the fuzzer was able to find the error with bad string formatting, but ChatGPT failed to do so. Furthermore, there are other non-issues that are reported as issues in the analysis. For instance, ChatGPT stated that `stdlib.h` is not imported even though it is imported, and that the function `foo` is undefined when it is defined and more. Thus, ChatGPT provided an incorrect analysis of code snippets 3.

In the analysis of code snippet 4, ChatGPT detected a potential error with the user input. Namely, a user could enter a position bigger than the size of the array `a` and cause an overflow. However, while that is part of the problem, the real problem lies within the code logic. As previously mentioned, the code will always cause an error regardless of the input. Remarkably, ChatGPT attempted to fix the code by providing a modified code snippet, which actually prevents the issue. However, that modified code snippet also initializes the variable `size` to 0, which changes the intended program flow. Therefore, ChatGPT never identified the real issue of the code and only managed to provide a partially correct analysis of code snippet 4.

4.2 Result of Experiment 2

In experiment 2, the code generated by ChatGPT and Copilot is evaluated for security using AFL++ and CodeQL. These AI tools are also assessed to investigate how well they are at evaluating the security of code snippets. The detailed summary of evaluations of all code snippets by the AI tools and the security testing tools can be found in table 7. It can be observed that when ChatGPT generated code by casual prompts, it was often found to be insecure by at least one tool, demonstrating a tendency towards generating vulnerable code. This was also the case for the code generated by Copilot, where vulnerabilities were detected by most of the tools.

Overall, it seems both AI tools tend to write vulnerable code, with Copilot being comparably worse in that aspect. As seen in table 5, out of all code snippets generated, roughly 43% of the code snippets by ChatGPT are insecure, and roughly 64% of the code by Copilot are insecure. Even after excluding snippets prompted by an insecure prompt, still ChatGPT generated 44% insecure code while Copilot generated 60% insecure code.

On the other side, we can observe how Copilot exceeds ChatGPT in accuracy when it comes to evaluating the security of source code. From table 6, we can conclude that Copilot could conduct a 75% correct analysis, while ChatGPT only achieved 46% accuracy when using AFL++ and CodeQL as reference points.

In most cases involving secure prompts, the generated code was indeed secure. However, some vulnerabilities were detected in specific snippets, such as snippet 7 and snippet 9. Even though ChatGPT generated these snippets, it still considered them as insecure. For example, it considered snippet 7 to be insecure and snippet 9 to be ambiguous. On the other side, code generated by Copilot was considered insecure by most tools in three out of five secure snippets, with ChatGPT being the only tool that was indecisive about the security of the code.

Prompt category	ChatGPT	Copilot	ChatGPT special case	Copilot special case
Casual	3 of 6	3 of 5	0	0
Secure	1 of 5	3 of 5	0	0
Insecure	2 of 3	3 of 4	1	1

Table 5: Summarized results of the analysis conducted by the tools (AFL++ and CodeQL). It shows how many generated code snippets are insecure according to the results provided by the tools. Snippets that fall under special cases are not considered. The numbers in each column represents how many *insecure* snippets were generated out of all snippets generated using the specified prompt category. The special cases are the code snippets that could not be fuzzed, and no errors have been reported by CodeQL.

ChatGPT	Copilot	Special case
13 of 28	21 of 28	2

Table 6: Summarized results of correct analysis conducted by the AI tools. Snippets that fall under special cases are not considered.

As expected, when generating code using insecure prompts, most of the snippets were considered vulnerable. The tools detected vulnerabilities in the code, but in some cases, the snippets were labeled as ambiguous by both ChatGPT and Copilot, while AFL++ and CodeQL did not find any errors like in snippet 35.

Overall, ChatGPT demonstrated proficiency at detecting insecurities in code snippets. However, it appeared less adept at detecting secure code, leading to questions about its decisiveness. It is notable that ChatGPT considered snippet 32 to be secure, despite CodeQL detecting three vulnerabilities. In contrast, Copilot consistently detected insecure code and was generally more decisive in its assessments.

Snippet No.	Category	Prompt	AI tool	ChatGPT	Copilot	AFL++	CodeQL
6	OF	Casual	ChatGPT	Insecure	Insecure	3	2 (1)
7	OF	Secure	ChatGPT	Ambiguous	Secure	1	0 (1)
8	ML	Casual	ChatGPT	Insecure	Secure	0	0 (1)
9	ML	Secure	ChatGPT	Insecure	Secure	0	0 (1)
10	BO	Casual	ChatGPT	Insecure	Insecure	3	3
11	ML	Insecure	ChatGPT	Ambiguous	Ambiguous	-	0
12	OF	Insecure	ChatGPT	Insecure	Insecure	2	2
13	BO	Casual	ChatGPT	Secure	Secure	0	0
14	BO	Secure	ChatGPT	Secure	Secure	0	0
15	OF	Casual	Copilot	Insecure	Insecure	2	2
16	OF	Secure	Copilot	Insecure	Insecure	2	2
17	OF	Insecure	Copilot	Insecure	Insecure	2	2
18	ML	Secure	Copilot	Ambiguous	Insecure	3	1 (1)
19	ML	Casual	Copilot	Insecure	Insecure	2	2 (1)
20	ML	Insecure	Copilot	Insecure	Insecure	2	2 (1)
21	BO	Casual	Copilot	Secure	Secure	0	0
22	BO	Secure	Copilot	Ambiguous	Insecure	3	1
23	BO	Insecure	Copilot	Insecure	Insecure	2	2
24	OF	Casual	Copilot	Ambiguous	Secure	0	0
25	OF	Secure	Copilot	Ambiguous	Secure	0	0
26	OF	Insecure	Copilot	Ambiguous	Ambiguous	0	0
27	BO	Casual	Copilot	Ambiguous	Secure	1	0
28	BO	Secure	Copilot	Ambiguous	Secure	0	0
29	BO	Insecure	Copilot	Ambiguous	Insecure	-	0
30	OF	Casual	ChatGPT	Ambiguous	Ambiguous	2	0
31	OF	Secure	ChatGPT	Secure	Ambiguous	0	0
32	OF	Insecure	ChatGPT	Secure	Secure	-	3
33	BO	Casual	ChatGPT	Ambiguous	Secure	0	0
34	BO	Secure	ChatGPT	Ambiguous	Secure	0	0
35	BO	Insecure	ChatGPT	Ambiguous	Ambiguous	0	0

Table 7: The table contains the detailed results of the second experiment. In the column **AFL++**, the digits represent the number of saved crashes by AFL++. Cells without a number indicate that we were not able to fuzz that code snippet. In column **CodeQL**, the numbers represent the number of errors reported, while numbers in parentheses are the number of warnings reported.

5 Discussion

In this section, we discuss the outcome of the experiments and analyze the potential threats that could have caused our results to be faulty. Additionally, we also present some ideas that might be of interest for future research similar to ours.

5.1 Discussion of Experiment 1

As can be seen in table 3, for most of the snippets, AFL++ has managed to discover some input that caused the snippets to crash or triggered some form of buffer overflow. However, no error was found in the code snippet 2 even though the code does contain a buffer overflow. In particular, at line 29 there is an erroneous usage of `strcpy` which writes the entirety of `buffer` into `buffer2`, even though `buffer` is bigger than `buffer2`. One potential explanation to this is the fact `buffer2` is never used again for any read or write operation after the `strcpy` function. The AddressSanitizer surveys a pre-allocated range of memory for any violation against the memory integrity, including buffer overflows. That means, the sanitizer probably could not detect any BO in `buffer2` since the program never accessed that memory region again. To attest this theory, the line of code `printf("%s", buffer2);` was added to code snippet 2 after the `strncpy` function to print out the content of `buffer2`. As expected, after recompiling the adjusted code snippet, the fuzzer now reports the crashes caused by the BO errors as seen in figure 5.

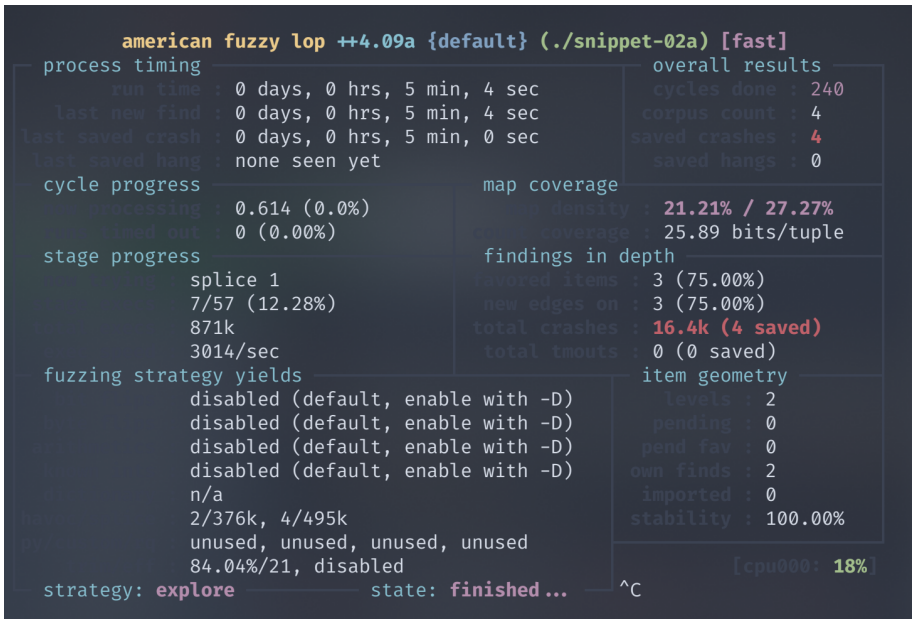

Figure 5: Fuzzing results of the modified code snippet 2.

It is noteworthy that in table 3, we were not able to fuzz code snippet 4. As mentioned in section 2.2.1, in order to fuzz a program, it must provide at least one valid input to the program. In this case, there is no valid input which does not cause a buffer overflow in the array `a`, as the `size` variable has been incremented at line 24, which causes it to be bigger than the actual size of the array `a`. This is, in other words, a logical error that always occurs regardless of the input.

This experiment revealed that both AFL++ and ChatGPT have their advantages and disadvantages when it comes to evaluating the security of source code. AFL++ showed its robustness as it produced results by exposing software vulnerabilities when injected with mutated inputs, which will probably lead to crashes. On the other side, ChatGPT offers advantages to identify vulnerabilities that may surpass a fuzzer, despite its low accuracy.

For instance, a fuzzer can only detect buffer overflow vulnerabilities when the buffer is accessed later. However, ChatGPT often provides a more flexible evaluation, which allows source evaluation in earlier stages when the software is non-executable. This means that ChatGPT has a static property, which AFL++ does not have.

5.2 Discussion of Experiment 2

The experiment reveals a significant trend in both ChatGPT and Copilot when responding to casual prompts. Code generated under such conditions was often found to be insecure by at least one of the evaluation tools. This suggests that both AI models have a propensity to produce code with potential vulnerabilities when given open-ended or less specific prompts. It raises questions about the need for additional guidance and quality control mechanisms in the development process to mitigate potential security risks.

Secure prompts generally resulted in code that was indeed secure, as expected. However, the detection of vulnerabilities in specific snippets, such as snippets 7 and 9, warrants attention. Even though the prompt mentioned that the code should be secure, ChatGPT generated some snippets that had some vulnerabilities, and even ChatGPT identified them as insecure or ambiguous. This points to a potential gap in ChatGPT's ability to generate code that is free of vulnerabilities.

The contrast in decisiveness between ChatGPT and Copilot is evident. ChatGPT exhibited a level of indecisiveness about code security in various instances, particularly with a lack of secure code identification. In one case, it identified snippet 32 as secure, while CodeQL detected three vulnerabilities. Copilot, on the other hand, consistently identified insecure code and displayed greater confidence in its assessments. This points out that ChatGPT is more careful with its assessments, so it tries to always point out that the code may be vulnerable. That frees ChatGPT from the responsibility of assuring security when it is not guaranteed, which seems to work in most cases and explains why it is not as good at detecting secure code.

These results have implications for software development and security practices. Developers and organizations using AI tools for code generation should be aware of the potential security risks associated with generating code in certain contexts. Robust security testing and code review procedures are essential, especially when using AI-generated code, to identify and address vulnerabilities.

In summary, the experiment provides insights into the performance of ChatGPT and Copilot in code generation with a focus on security. It highlights the need for careful consideration of the context and prompts when using AI models, and underscores the importance of supplementing AI-generated code with thorough security evaluations to ensure the development of secure software. Further research and refinement of AI models like ChatGPT and Copilot may help address some identified challenges and enhance their reliability in generating secure code.

5.3 Threats to Validity

There exist a few factors that could have influenced the correctness of our work negatively. For instance, using AFL++ to fuzz the code may not be the optimal way to really check for crashes or vulnerabilities. AFL++ sends in different input in the code and check if the code actually works without any crashes. So depending on the input the code might or might not crash as we only fuzz a code snippet once. This makes fuzzing with AFL++ not a perfect way to check if the code has any vulnerabilities, as the random input sent in might not crash the code. For this case, the code should be fuzzed multiple times to further validate the correctness of our work.

In the second experiment, having more prompts and vulnerabilities could make the conclusion more accurate since more data are taken into account and new unstudied aspects could appear. The current code dataset is generated using only C language and is focused on three vulnerabilities: buffer overflow, memory leak, and obsolete functions. This can be a threat to validity since AI tools behave differently with different programming languages. Using another language using the same experiments may yield new results.

5.4 Future Work

Since we have used primarily AFL++ and CodeQL to assess the security of the code generated by the AI tools, the quality of the assessment is naturally limited by the quality of the fuzzer and static analyzer. That is to say, if there are some vulnerabilities that are invisible to the fuzzer then the results will not reflect the presence of such vulnerabilities even if it might be obvious for some other tools. It is therefore an alternative to construct experiments based on other tools.

Another idea is to conduct more experiments, such as investigating how generating code snippets in different programming languages may affect the security and quality of the generated code. For example, a question to be answered could be, "Will these tools generate more secure Python code compared to C?". This is an important aspect to study since the code snippets, which the AI-tools generate, are derived from the training the tool had on code snippets from the programming community. This means in theory that AI-tool should generate more secure code for popular programming languages such as Python.

6 Challenges

In this section, the challenges that were encountered while conducting the experiments will be presented.

In the first experiment, it was challenging to find the optimal code snippets that would fit the experiment. The code snippets can not be too complex, as AI-tool have limitation when analyzing code from different files without explanations with human language on how the application work and its cohesion. Simultaneously, the code snippet should be long enough to contain a vulnerability to make a valid experiment. The project was decided to be done objectively with the least amount of code modification possible by the authors, which means that writing the code snippets was not an option. This issue was solved by specifying the experimented vulnerabilities to narrow down the code search on Stack Overflow.

The second experiment was a little bit less challenging since the authors gained more experience conducting the first experiment. Bypassing the ethical guidelines of the AI-tool was not easy. The tools refused to generate insecure code since it may be used maliciously. The solution was by asking a general question about the vulnerability such as "What is buffer overflow", then follow this question by asking the tool to give an example in a certain language demonstrating the vulnerability.

Choosing relevant vulnerabilities was not an easy task, either. Many vulnerabilities require manual testing, and a fuzzer such as AFL++ might not have the ability to detect them. The code must take input from the user in order to be testable by AFL++. Additionally, as mentioned above, the vulnerability must be adapted to the experiment scope and the project deadline. The vulnerabilities were selected to be memory-related since they commonly occur when writing code in C, which is the primary language used in this project.

7 Related Work

In this report, research was conducted to assess the security aspect of C code generated by ChatGPT and GitHub Copilot, using AFL++ fuzzer and CodeQL. Naturally, given the increasing interest for AI-assisted programming, there can be found various similar research which has been conducted on the topic of generative AI previously.

For instance, Siddiq and Santos present a dataset of prompts that can be used to evaluate the security of AI-generated code [35]. In the paper, an example of how this dataset can be used is shown, where GitHub Copilot and InCoder are used to generate the code, and then evaluated the generated code snippets using static analyzers. The dataset presented by Siddiq and Santos focused on Python code and considered only vulnerabilities that can emerge in Python. Therefore, vulnerability categories not possible in Python were excluded from the dataset. On the contrary, our research focuses on C, a language prone to low-level memory errors which cannot be tested given their dataset.

Furthermore, there are some studies where the primary goal is akin to our research and seek to assess the security aspect of code generated by AI tools. The papers by Yetiştiren et al., Pearce et al. and Liu et al. analyze how secure the code generated by various AI tools are, including ChatGPT and GitHub Copilot [7], [10], [11]. However, all of these studies employ only static code analyzers and sometimes manual inspection to analyze the code, while in our research a fuzzer is utilized in addition to a static code analyzer. Despite that, the results from these studies resemble the result we have derived.

Lastly, there is also a study on this topic that adopts a user-centric approach, where instead of researching the code generated by AI tools, the study focuses on the interaction between users and AI tools [8]. In this study, it was found out that users who relied on AI tools less generally produced code that contained fewer vulnerabilities. While the study by Perry et al. [8] is not directly analogous to our research in terms of purpose, it can however bring some new insights by looking at this problem from a new perspective.

8 Conclusion

The importance of code security cannot be understated, as it forms the foundation for the reliability and safety of modern software applications. AI-generated code that lacks security entails severe consequences, for instance, data breaches or system breakdowns. Hence, it becomes an absolute necessity to conduct a meticulous assessment of the security aspect of AI-generated code, as exemplified by the outcomes of our experiments.

The first experiment revealed distinctions in security assessment capabilities between ChatGPT and AFL++. Both AFL++ and ChatGPT have their advantages and disadvantages when it comes to evaluating the security of source code. AFL++ is more robust and reliable, since the results are yielded when the software crashes while injecting it with mutated input. While ChatGPT is more like a static analyzer, namely the code does not need to be executable to be evaluated. This allows assessment when the source code development is in its early phase, which AFL++ does not. However, ChatGPT is not accurate and may yield incorrect results. In special cases, ChatGPT can detect vulnerabilities which a fuzzer cannot. For example, a fuzzer can only detect a buffer overflow vulnerability when the buffer is accessed later. In such cases, ChatGPT can provide a more accurate evaluation due to its static property.

The second experiment revealed that the security of code snippets generated by AI tools need to be manually evaluated to ensure the development of secure software. Developers using AI tools need to be aware of the potential security risks related to generated code. They must learn how to prompt such tools to produce optimal results and understand that adding specific keywords to the prompt may result in significantly improved code. It is also good to mention that the quality of code generation and evaluation of these tools depends on how well these tools are trained to write and analyze source code.

In conclusion, using AI tools for code generation and evaluation may represent the future of developer assistance, but it is still in an early phase. Until these tools have proven their accuracy, developers must evaluate the generated code manually or with the aid of fuzzers to achieve greater reliability in the code.

References

[1] K. Martineau. "What is generative AI?" (Apr. 2023), [Online]. Available: https://research.ibm.com/blog/what-is-generative-AI (visited on 10/14/2023).

[2] OpenAI. "DALL·E 2." (2023), [Online]. Available: https://openai.com/dall-e-2 (visited on 10/14/2023).

[3] A. Hughes. "ChatGPT: Everything you need to know about OpenAI's GPT-4 tool." (2023), [Online]. Available: https://www.sciencefocus.com/future-technology/gpt-3 (visited on 09/19/2023).

[4] GitHub. "About GitHub Copilot for Individuals - GitHub Docs." (2023), [Online]. Available: https://docs.github.com/en/copilot/overview-of-github-copilot/about-github-copilot-for-individuals (visited on 09/29/2023).

[5] Stack Overflow. "Developer sentiment around AI/ML." (Jun. 2023), [Online]. Available: https://stackoverflow.co/labs/developer-sentiment-ai-ml/ (visited on 10/14/2023).

[6] K. Jesse, T. Ahmed, P. T. Devanbu, and E. Morgan, *Large Language Models and Simple, Stupid Bugs*, 2023. arXiv: 2303.11455 [cs.SE].

[7] 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. arXiv: 2304.10778 [cs.SE].

[8] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, *Do Users Write More Insecure Code with AI Assistants?* 2022. arXiv: 2211.03622 [cs.CR].

[9] K. R. Go, S. Soundarapandian, A. Mitra, M. Vidoni, and N. E. D. Ferreyra, "Simple stupid insecure practices and GitHub's code search: A looming threat?" *Journal of Systems and Software*, vol. 202, p. 111 698, 2023, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2023.111698. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121223000936.

[10] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of GitHub Copilot's code contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2022, pp. 754–768.

[11] Z. Liu, Y. Tang, X. Luo, Y. Zhou, and L. F. Zhang, *No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT*, 2023. arXiv: 2308.04838 [cs.SE].

[12] I. Shani. "Survey reveals AI's impact on the developer experience - The GitHub Blog." (Jun. 2023), [Online]. Available: https://github.blog/2023-06-13-survey-reveals-ais-impact-on-the-developer-experience/ (visited on 10/10/2023).

[13] OpenAI. "ChatGPT." (2023), [Online]. Available: https://openai.com/blog/chatgpt (visited on 09/19/2023).

[14] Nsrav. "Denial of Service." (2023), [Online]. Available: https://owasp.org/www-community/attacks/Denial_of_Service (visited on 10/16/2023).

[15] OWASP. "Memory leak." (2023), [Online]. Available: https://owasp.org/www-community/vulnerabilities/Memory_leak (visited on 10/13/2023).

[16] OWASP. "Use of obsolete methods." (2023), [Online]. Available: https://owasp.org/www-community/vulnerabilities/Use_of_Obsolete_Methods (visited on 10/13/2023).

[17] OWASP. "Buffer overflow." (2023), [Online]. Available: https://owasp.org/www-community/vulnerabilities/Buffer_Overflow (visited on 10/13/2023).

[18] OWASP. "Denial of Service." (2023), [Online]. Available: https://owasp.org/www-community/Fuzzing (visited on 10/16/2023).

[19] AFLplusplus. "The AFL++ fuzzing framework — AFLplusplus." (2023), [Online]. Available: https://aflplus.plus/ (visited on 09/23/2023).

[20] Google. "Glossary." (2023), [Online]. Available: https://google.github.io/clusterfuzz/reference/glossary/#sanitizer (visited on 10/16/2023).

[21] Wikipedia, *Code sanitizer — Wikipedia, the free encyclopedia*, 2023. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Code%5C%20sanitizer&oldid=1151556919 (visited on 10/16/2023).

[22] T. C. Team. "AddressSanitizer – Clang 18.0.0git documentation." (2023), [Online]. Available: https://clang.llvm.org/docs/AddressSanitizer.html (visited on 10/16/2023).

[23] K. Shlyakhovetska. "A gentle introduction to static code analysis." (Sep. 2023), [Online]. Available: https://www.infoworld.com/article/3705568/a-gentle-introduction-to-static-code-analysis.html (visited on 10/16/2023).

[24] GitHub. "About CodeQL." (2023), [Online]. Available: https://codeql.github.com/docs/codeql-overview/about-codeql/ (visited on 10/15/2023).

[25] Stack Overflow. "Stack Overflow - Where Developers Learn, Share, & Build Careers." (2023), [Online]. Available: https://stackoverflow.com/ (visited on 10/14/2023).

[26] F. Duarte. "Number of ChatGPT Users (2023)." (Jul. 2023), [Online]. Available: https://explodingtopics.com/blog/chatgpt-users (visited on 10/14/2023).

[27] Google. "american fuzzy lop - a security-oriented fuzzer." (2021), [Online]. Available: https://github.com/google/AFL (visited on 10/14/2023).

[28] AFLplusplus. "Fuzzing binary-only targets." (2023), [Online]. Available: https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/fuzzing_binary-only_targets.md (visited on 10/14/2023).

[29] Wikipedia. "C (programming language) — Wikipedia, the free encyclopedia." (2023), [Online]. Available: http://en.wikipedia.org/w/index.php?title=C%5C%20(programming%5C%20language)&oldid=1180141290 (visited on 10/15/2023).

[30] A. Chandra. "What is Memory Leak in C?" (Nov. 2022), [Online]. Available: https://www.scaler.com/topics/memory-leak-in-c/ (visited on 10/15/2023).

[31] D. Doodat. "Day 49: Common C Code Vulnerabilities and Mitigations." (Feb. 2019), [Online]. Available: https://int0x33.medium.com/day-49-common-c-code-vulnerabilities-and-mitigations-7eded437ca4a (visited on 10/15/2023).

[32] N. Kolakowski. "Which Programming Language has the Most Vulnerabilities?" (Nov. 2019), [Online]. Available: https://www.dice.com/career-advice/programming-language-vulnerabilities (visited on 10/15/2023).

[33] Stack Overflow. "Stack Overflow Developer Survey 2023." (2023), [Online]. Available: https://survey.stackoverflow.co/2023/ (visited on 10/15/2023).

[34] AFLplusplus. "Notes for using ASAN with afl-fuzz." (2023), [Online]. Available: https://aflplus.plus/docs/notes_for_asan/ (visited on 10/15/2023).

[35] M. L. Siddiq and J. C. S. Santos, "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*, ser. MSR4P&S 2022, Singapore, Singapore: Association for Computing Machinery, 2022, pp. 29–33, ISBN: 9781450394574. DOI: 10.1145/3549035.3561184. [Online]. Available: https://doi.org/10.1145/3549035.3561184.

[36] D. Yao. "One Year On, GitHub Copilot Adoption Soars." (Jun. 2023), [Online]. Available: https://aibusiness.com/companies/one-year-on-github-copilot-adoption-soars (visited on 10/15/2023).

[37] C. Dias. "Visual Studio Code and GitHub Copilot." (Mar. 2023), [Online]. Available: https://code.visualstudio.com/blogs/2023/03/30/vscode-copilot (visited on 10/15/2023).

[38] SonarSource, *Sonarqube*, version 10.2, Sep. 2023. [Online]. Available: https://www.sonarsource.com/products/sonarqube/.

[39] L. Zelleke. "6 Best Static Code Analysis Tools for 2023 (Paid & Free)." (Aug. 2023), [Online]. Available: https://www.comparitech.com/net-admin/best-static-code-analysis-tools/ (visited on 10/15/2023).

[40] GitHub. "About CodeQL queries." (2023), [Online]. Available: https://codeql.github.com/docs/writing-codeql-queries/about-codeql-queries/ (visited on 10/15/2023).

[41] GitHub. "Built-in CodeQL query suites." (2023), [Online]. Available: https://docs.github.com/en/code-security/code-scanning/managing-your-code-scanning-configuration/built-in-codeql-query-suites (visited on 10/15/2023).

[42] R. Dewhurst. "Static Code Analysis." (2023), [Online]. Available: https://owasp.org/www-community/controls/Static_Code_Analysis (visited on 10/16/2023).

[43] Google. "Google C++ Style Guide." (2023), [Online]. Available: https://google.github.io/styleguide/cppguide.html (visited on 10/13/2023).

A Code Snippets

The code snippets that are directly referenced in the report are presented here for quick access. Note that all code snippets have been reformatted according to the Google style guidelines for better readability [43]. To see other code snippets or for more details, such as fuzzing results and transcripts of ChatGPT responses, visit our publicly available GitHub repository.

Code Snippet 1: The first code snippet from the first experiment. The code is retrieved from StackOverflow.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>

#define BUFSIZE 30

char grade = '3';
char Name[BUFSIZE];

void readString(char *s) {
  char buf[BUFSIZE];
  int i = 0;
  int c;
  while (1) {
    c = fgetc(stdin);
    if ((c == EOF) || (c == '\n')) break;
    buf[i++] = c;
  }
  buf[i] = 0;
  for (i = 0; i < BUFSIZE; i++) s[i] = buf[i];

  return;
}

int main(void) {
  mprotect((void *)((unsigned int)Name & 0xfffff000), 1,
           PROT_READ | PROT_WRITE | PROT_EXEC);

  printf("What is your name?\n");
  readString(Name);

  if (strcmp(Name, "smart guy") == 0) grade = '6';

  printf("Thank you, %s.\n", Name);
  printf("I recommend that you get a grade of %c on this  assignment.\n",
         grade);

  exit(0);
}
```

Code Snippet 2: The second code snippet from the first experiment. This code is first retrieved from StackOverflow. However, since the original code does not take in user input, we asked ChatGPT to modify it to take in a user input and copy it to a buffer. With slight modifications to expose the BO vulnerability, we obtained the following code.

```c
#include <stdio.h>
#include <string.h>

int func() {
  char buffer[100];
  char buffer2[64];
  // Read input from stdin
  printf("Enter a string (up to 99 characters): ");

  if (fgets(buffer, sizeof(buffer), stdin) == NULL) {
    // Handle error if fgets fails (e.g., end of file)
    perror("fgets");
    return -1;
  }

  // Remove the newline character at the end of the input, if present
  size_t len = strlen(buffer);
  if (len > 0 && buffer[len - 1] == '\n') {
    buffer[len - 1] = '\0';
  }

  // Check if the input is too long
  if (len >= sizeof(buffer)) {
    printf("Input is too long!\n");
    return -1;
  }

  // Copy the input to a buffer
  strncpy(buffer2, buffer, strlen(buffer));

  return 0;
}

int main() {
  int result = func();
  if (result == 0) {
    printf("Input copied successfully!\n");
  } else {
    printf("Input copy failed!\n");
  }
  return 0;
}
```

Code Snippet 3: The third code snippet from the first experiment. The code is retrieved from StackOverflow.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv) {
  (void)foo(argv[1]);
  exit(0);
}

int foo(char *arg) { return bar(arg); }

int bar(char *arg) {
  char lbuf[1024];
  if (strlen(&arg) >= 1024) return -1;

  memset(lbuf, 0, sizeof(lbuf));
  sprintf(lbuf, "%s", "Welcome: ");
  read(0, lbuf + strlen(lbuf), sizeof(lbuf) - strlen(lbuf) - 1);
  printf(lbuf);
  fflush(stdout);

  return 0;
}
```

Code Snippet 4: The last code snippet from the first experiment. The code is retrieved from StackOverflow.

```c
#include <stdio.h>

int main() {
  int a[] = {10, 20, 30, 40, 50};
  int size = sizeof(a) / sizeof(a[0]);
  int num, pos;
  printf("Enter the number to insert in the array: ");
  scanf("%d", &num);
  printf("Enter the position to enter the element in the array: ");
  scanf("%d", &pos);

  if (pos < 0 || pos > size) {
    printf("Invalid position!");
  } else {
    // Move elements to make space for the new element
    for (int i = size - 1; i >= pos; i--) {
      a[i + 1] = a[i];
    }

    // Insert the new element
    a[pos] = num;

    // Update the size of the array
    size++;

    printf("The updated array is: ");
    for (int i = 0; i < size; i++) {
      printf("%d ", a[i]);
    }
  }
  return 0;
}
```