Analysis of nature language code summaries to detect software vulnerabilities

Bachelor Thesis

Author:

Yossef Al Buni

Matr. Nr.: 53803

Informatik-Ingenieurwesen

Supervisors:

Prof. Dr-Ing. Riccardo Scandariato
Dr-Ing. Nicolas Diaz Ferreyra

Scientific Advisors:

Torge Hinrichs, M. Sc.

Institute of Software Security

Blohmstraße 15, 21079 Hamburg, Germany

Juni 2023

Declaration

I hereby declare that I have written this thesis independently and that I have not made use of any aid other than those acknowledge in this thesis. I further declare that neither this thesis nor any other similar work has been previously submitted to any examination board.

Hamburg, June 20, 2023

Yossef Al Buni

Abstract

The popularity of LLM models such as ChatGPT is rising due to their capacity to generate text closely resembling human language and effectively tackle coding exercises.
This thesis explores the reliability of LLM models in generating code reviews, which might help developers and specialists in the future by saving time and afford to write them manually. Therefore, the employed method entailed the utilization of two essential components. Firstly, the adoption of the gpt-3.5-turbo engine, developed by OpenAI and released in March 2023. Secondly, a dataset consisting of Java code snippets was employed, wherein each snippet was associated with CWE-type flags and a corresponding patched version.
Using OpenAI's API, we generated 50 code reviews for each vulnerable and patched code snippet. We thoroughly reviewed each generated code review, labeling it as "Yes" if it identified the correct vulnerability and "No" if it didn't. Subsequently, we assessed the performance of generating a good code review by employing evaluation metrics, such as confusion matrix, accuracy, recall, precision and F1-score. We repeated the code review generation process using a more security-detailed prompt and compared both. Furthermore, we performed a text analysis of the generated code reviews, where we computed first, the word count of the generated output and the character count of the input code snippet, and second, the TF-IDF matrix to obtain if elaborating the prompt influences the security attention in the code reviews. Later, we analyzed the behavior of the GPT model by focusing on misclassified cases and their TF-IDF matrices between code reviews of vulnerable and fixed code snippets. Finally, we will present reasons for misclassifications based on the obtained results.
We found that a more detailed prompt increases accuracy from 55% to 60%, heightening security attention. It generates more code reviews containing correct vulnerabilities and others containing non-existent ones in the code, which are very similar to each other. Our outcome indicates that the gpt-3.5-turbo model is capable of generating effective code reviews, but not for every code snippet, which diminishes its reliability, particularly in safeguarding software systems.

Contents

Chapter 1

Introduction

Considering security in software systems is essential in today's digital environment, where code vulnerabilities can have serious consequences. In recent years, large-scale language models (LLM) have made remarkable strides in producing human-like texts in various industries. One of the most exciting applications of LLM is the generation of code analysis [9], which can provide valuable insights and feedback for software development. This study explores the capability of a LLM models in code analysis, focusing on identifying security vulnerabilities.

1.1 Overview

Security vulnerabilities are widespread in our IT world. Attackers can exploit them to gain unauthorized access, steal essential and sensitive information, or cause software damage to a system. To protect our software system from such vulnerabilities we need developers and specialists to review the source code, which could take a long time to achieve, considering a developer's time analyzing and understanding the code. It would be easier if an AI technique explained the code by generating a code review with security aspects. The AI technique takes a code snippet as an input prompt and describes briefly the code in a generated text, which is understandable and readable for everyone [9].
These techniques are becoming popular in the last few months, for example, the currently known ChatGPT [25].
ChatGPT, developed by OpenAI, is one of the most significant publicly available language models, capable of capturing the nuances and intricacies of human language, allowing it to generate appropriate and contextually relevant responses across a broad spectrum of prompts[25]. ChatGPT can generate code for simple or repetitive tasks, such as file I/O operations, data manipulation, and database queries. However, its ability to write code is limited, and the generated code may not always be accurate, optimized, or the desired output [18].
It is a fine-tuned version of GPT-3, with 175 Billion parameters, and trained to interact with users and receive feedback from them. GPT-3 has been trained on a diverse range of texts and can perform a variety of programming-related tasks, including code completion

and correction, code snippet prediction and suggestion, automatic syntax error fixing, code optimization and refactoring suggestions, missing code generation, document generation, chatbot development, text-to-code generation, and answering technical queries [5].

In most cases, ChatGPT helps developers nowadays with programming tasks. Now people can write and understand code in every programming language much more accessible than before using this model by just typing a prompt.
Since we know that the gpt-3.5-turbo model (released in March 2023) has the best performance compared with all GPT-3.5 models with the lowest cost [25] and powers the big ChatGPT model [22], We would now discover if the gpt-3.5-turbo model can help us in generating reliable code reviews which highlight present vulnerabilities in the code.

1.2 Research Question

The central research question addressed in this study is:

Can large-language models effectively generate informative code reviews that encompass essential security aspects, particularly vulnerabilities?

This question explores the capabilities of large-language models in providing comprehensive and meaningful code reviews that highlight security concerns.
We can approach this issue by breaking it down into smaller sub-questions:

1. *What is the Performance of the large-language model "gpt-3.5-turbo" in generating effective code reviews, which include the correct vulnerabilities in the code?*
2. *How does a "detailed" prompt influence the large-language model's efficiency in generating code reviews and finding vulnerabilities? Does this improve or worsen the generated output?*
3. *Can the effectiveness of the large-language model be influenced by the size of the input code or the length of the generated code review?*
4. *How does the LLM behave in case of false predictions, and how does the behavior change between generating code reviews of vulnerable code snippets and code reviews of secure code snippets?*
5. *Does the security relation change by elaborating the prompt?*
6. *Is it trustworthy to rely on the large-language model "gpt-3.5-turbo" to identify security vulnerabilities in the code instead of manually reviewing the original code?*

To find the answers, We will present the methods in chapter 3, and present the outcomes in chapter 4. Finally, we will discuss each sub-question in detail in Chapter 5 and present the final result, which answers the main research question 1.2, 1, 2, 3, 4, 5 and especially question number 6.

1.3 Contributions

This research makes several significant contributions to code review generation using LLM models, with a specific focus on identifying security vulnerabilities. The following contributions highlight the aspects of this study:

1. **Model's Quality in Generating Code Reviews**:
   We created a pipeline 3.1 for the code generation process and evaluated the performance in generating informative code reviews.
2. **Prompt Analysis**:
   We used two prompts in this work, the first one generates code reviews and the second generates code reviews with a focus on security aspects, and we compared their outcomes using the TF-IDF matrix.
3. **Word count and Code size**:
   We measured the word count of each generated code review and the character count of each input code snippet, and compared the result to identify any model's dependency on Input/Output size.
4. **Comparative Analysis of Vulnerable and Fixed Code Reviews**:
   Additionally, this research contributes to understanding the model's behavior by comparing the generated reviews of vulnerable and fixed code snippets. By examining the similarities and differences between these reviews, we gain insights into how the model captures and addresses security vulnerabilities.
5. **Reliability in Security Field** This analysis provides valuable insights into the reliability and trustworthiness of the LLM model in addressing security concerns and identifying vulnerabilities.

Chapter 2

Related work and Background

2.1 General Overview about Transformer architecture

The Transformer architecture, introduced by *Vaswani et al. "Attention is all you need"* (2017) [3], has quickly become one of the most popular models for natural language processing (NLP) tasks, including machine translation, language modeling, and question answering. The Transformer architecture is based on the concept of attention, which allows the model to selectively attend to different parts of the input sequence when generating the output. Since its introduction, the Transformer has been the subject of extensive research, with numerous variants and extensions proposed in the literature.
Benyamin Ghojogh, Ali Ghodsi in their paper *Attention Mechanism, Transformers, BERT, and GPT: Tutorial and Survey* [4] described the autoencoder model. Figure 2.1 shows the transformer architecture with the encoder and decoder.
The encoder takes in an input sequence and generates a sequence of hidden states, while the decoder takes in this sequence of hidden states and generates an output sequence one token at a time.
The key components of the transformer architecture are multi-head self-attention, multi-head cross-attention, and feedforward neural networks. Multi-head self-attention allows each token in the input sequence to attend to all other tokens in the same sequence, while multi-head cross-attention allows each token in the output sequence to attend to all tokens in the input sequence.
In general the decoder uses multi-head attention to attend to different parts of the encoder's output at each step, allowing it to selectively focus on relevant information when generating the output sequence.
Positional encoding is used to provide information about the position of each token within its sequence, which is important for capturing sequential dependencies. The feed-forward neural networks are used to transform the hidden states between different layers of the encoder and decoder [4].

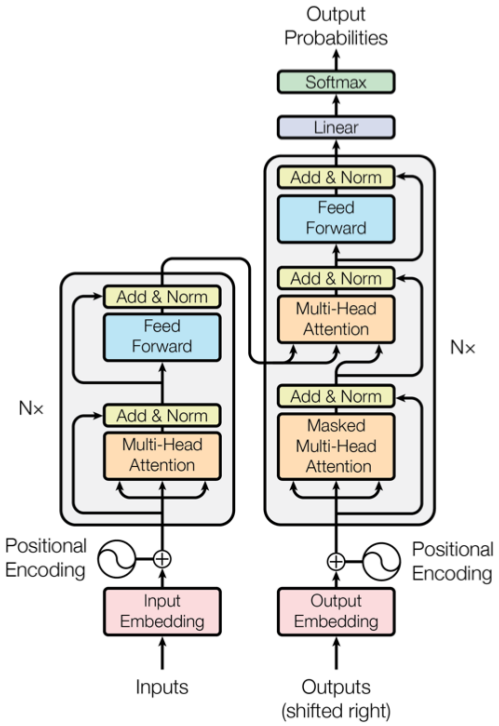


Figure 2.1: Transformer architecture [3]

2.2 Transformer Architectures In Security Fields

One application of this architecture is vulnerability detection, where it has been integrated into developer-friendly tools for code security [7]. For example, VDet for Java is a transformer-based VS Code extension that can detect up to 21 vulnerability types with an accuracy of 98.9% for multi-label classification [7]. In another study, CodeBERT, a deep contextualized model based on the Transformer architecture, was used to detect vulnerabilities in C open-source projects [17].
Kanchan Singh; Sakshi S Grover and Ranjini Kishen Kumar experimented CodeBERT on both vulnerability classification and multiclass vulnerability classification. They came out with the results in paper [10], that CodeBERT achieved an accuracy of 94% in Vulnerability classification and 95% in Multiclass vulnerability classification, which means that the CodeBERT model can find and categorize software vulnerabilities in code with outstanding results.
The Transformer architecture has also been used in other areas, such as network attack detection in software-defined networking [16], and improving the robustness of object detectors to adversarial attacks [13], [8].

2.3 Introducing Large Language Models

LLM or large language models are neural network architectures used for natural language processing. They typically have billions of parameters and are pre-trained on auto-supervised tasks specific to a given language. LLM models have been used for various tasks such as text classification, syntactic and semantic matching, and sequence-to-sequence text generation [11].
Generative Pre-trained Transformer (GPT) is perhaps the most widely known LLM and is part of the transformer family of architectures.
OpenAI researchers *Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever* published paper "Improving Language Understanding by Generative Pre-Training" [1] (2018) and showed that pre-training the GPT model on a diverse corpus with long stretches of contiguous text allows the model to achieve significant world knowledge and ability to process long-range dependencies which are then successfully transferred to solving discriminative tasks.

Benyamin Ghojogh, Ali Ghodsi introduced after two years in the paper "Attention Mechanism, Transformers, BERT, and GPT: Tutorial and Survey" (2023) [4] GPT (Generative Pre-trained Transformer) as a stack of decoders in a transformer architecture. GPT-2 and GPT-3 are subsequent iterations of the GPT model, featuring larger models with more attention heads and layers. These enhancements result in increased learnable parameters, enabling better language modeling and inference capabilities compared to GPT-1. These models have been trained on large amounts of internet data to generate text in any subject and style of interest.

Haoye Tian, Weiqi Lu evaluated this year in paper "Is ChatGPT the Ultimate Programming Assistant - How far is it?" [12] (2023) the ChatGPT's capabilities in three key areas: code generation, program repair, and code summarization. The researchers found that ChatGPT effectively handles typical programming challenges, such as generating and repairing incorrect code. They showed us the dominance of ChatGPT while revealing that ChatGPT struggles to generalize to new and unseen problems. They validate the negative impact of long prompts on the inference capabilities of ChatGPT.
Tom B. Brown explained in "Language Models are Few-Shot Learners" [15] how to create a prompt and gave us some examples. One example is the prompt used to generate poetry in the style of Wallace Stevens, which included the title of the poem and specified the desired style, such as "Write a poem titled 'The Snowman' in the style of Wallace Stevens" [15].
Based on the research "In ChatGPT We Trust? Measuring and Characterizing the Reliability of ChatGPT" from *Xinyue Shen and Zeyuan Chen* [6], ChatGPT allows users to steer its behaviors by describing directions via system role, the authors have observed that ChatGPT's dependability differs, particularly when it comes to law and science inquiries. Additionally, they have proven that the system roles, which OpenAI initially created to enable users to influence ChatGPT's actions, can affect its consistency.

2.4 Large Language Models In Vulnerability Detection

Anton Cheshkov, Pavel Zadorozhny, and Rodion Levichev evaluated in their paper [2] (2023) the performance of the ChatGPT and GPT-3 models for the task of vulnerability detection in code. According to their results, the ChatGPT model did not perform better than a dummy classifier for both binary and multi-label classification tasks for code vulnerability detection. The dummy classifier does not learn anything from the data and simply makes predictions based on the distribution of classes in the training set [26]. The researchers conducted binary classification tests on the gpt-3.5-turbo model and gathered a dataset of 308 samples. This dataset comprised both vulnerable and patched Java functions. The GPT model was then employed to determine whether the code was vulnerable, without specifically classifying the nature of the vulnerability. They also tested the multi-label classification of this model, where they compiled a dataset of 120 samples, with 60 vulnerable functions and 60 patched samples, focusing on the top five Common Weakness Enumeration (CWE). The results using the gpt-3.5-turbo engine are as follows:

- for binary classification presented in table 2.1
- for multi-label classification of 60 samples in figure 2.2a
- for multi-label classification of 120 samples in figures 2.2b

They approved after these results, that the current capabilities of GPT-3 and ChatGPT for effectively detecting vulnerabilities in code are limited and that their application in vulnerability detection tasks requires further refinement and investigation.

Model	**Binary**		
	Precision	**Recall**	**F1-score**
gpt-3.5-turbo	0.51	0.8	0.62

Table 2.1: The results of gpt-3.5-Turbo in binary classification with 308 samples [2]

Experienced developers conduct code reviews as an essential aspect of software quality assurance, in addition to static analysis and testing. Code review can identify errors that may go undetected during testing or static analysis, such as possible vulnerabilities in the libraries utilized. However, due to their busy schedules, these experienced developers may not always have the necessary time to conduct thorough code reviews [9]. *Giriprasad Sridhara, Ranjani H.G. and Sourav Mazumdar* explored in their paper "ChatGPT: A Study on its Utility for Ubiquitous Software Engineering Tasks" (2023) [9] the Capability of ChatGPT in common software engineering tasks, and tested the model in code reviews generation. They provided 10 Java methods to ChatGPT and request it to provide a code review of each method. They compared the generated review with the original developers' review. Out of 10 code reviews, 4 code reviews matched the developer's reviews and 6 didn't. Therefore, they concluded that ChatGPT in its current form cannot be used for code review generation. [9].

	Multi			
CWE type	Precision	Recall	F1-Score	support
CWE-20	1.000	0.071	0.133	14
CWE-200	0.000	0.000	0.000	9
CWE-502	1.000	0.143	0.250	7
CWE-611	0.250	0.077	0.118	13
CWE-79	1.000	0.176	0.300	17
accuracy			0.100	60
macro avg	0.542	0.078	0.133	60
weighted avg	0.688	0.100	0.171	60

(a) Performance Evaluation of the gpt-3.5-turbo model on 5 CWE types using 60 samples [2]

	Multi			
CWE type	Precision	Recall	F1-Score	support
CWE-20	0.333	0.071	0.118	14
CWE-200	0.000	0.000	0.000	9
CWE-502	0.200	0.143	0.167	7
CWE-611	0.083	0.077	0.080	13
CWE-79	0.333	0.176	0.231	17
Negative	0.440	0.667	0.530	60
accuracy			0.383	120
macro avg	0.232	0.189	0.187	120
weighted avg	0.327	0.383	0.330	120

(b) Performance Evaluation of the gpt-3.5-turbo model on 5 CWE types using 120 samples [2]

Figure 2.2: Performance Evaluation of the gpt-3.5-turbo model in multi-label classification [2]

Chapter 3

Research Methodology

This thesis aims to evaluate the effectiveness of LLM models in generating code reviews with vulnerability identification. Therefore, the methods employed in this chapter are introduced to address questions 1.2, 1, 2, 3, 4, 5 and 6 in this research.
Figure 3.1 offers a general overview of the work. Each colored part of the sketch will be examined, described, and the reason for its use will be mentioned. A detailed discussion of the analysis component will be presented in the dedicated analysis sections: 3.5 and 3.6.

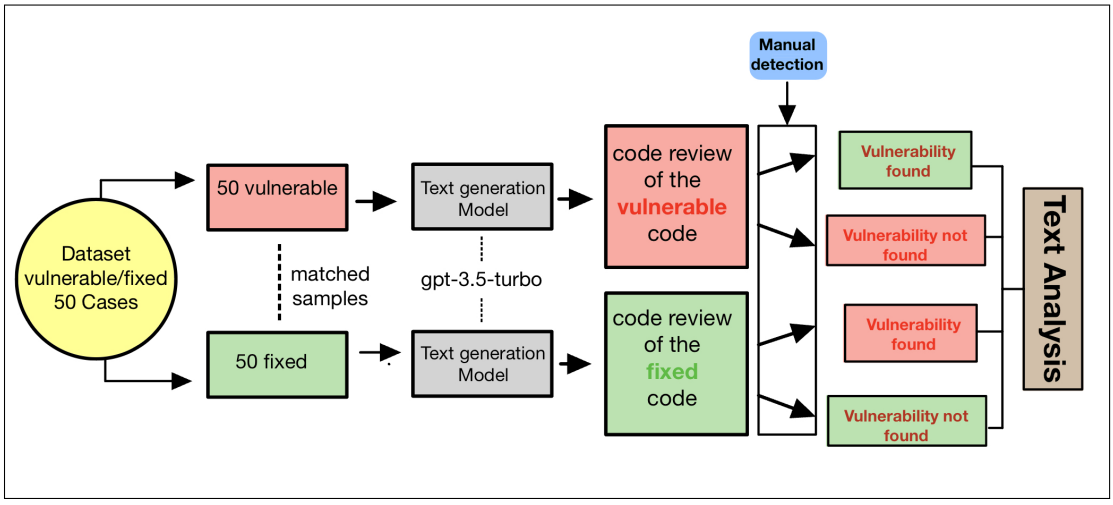


Figure 3.1: The general pipeline of the research

3.1 Dataset

The yellow circle in the methodology sketch 3.1 refers to the dataset used in this research. The research used 50 samples out of 196 samples from the dataset, these are colored with red and blue. The dataset is curated by Torge Hinrichs, M. Sc. and Quang

Cuong Bui, M. Sc. at the Institute of Software Security (E-22) in Hamburg University of Technology (TUHH), characterized by the following structural composition for each sample in the JSON-list: Each sample of the 50 contains a vulnerable code snippet and

Key	Value
"idx"	Index ID
"vuln_id"	The vulnerability ID
"desc"	A short description of the current problem
"cwe_id"	the CWE ID
"cwe_name"	The CWE ID's definition
"repo"	The git repository of the code snippet
"commit"	The commit of the code snippet
"method_before"	The code snippet in vulnerable form
"method_after"	The code snippet in fixed form

Table 3.1: The JSON structure of the dataset

its corresponding fixed code snippet. Two code reviews were generated for both vulnerable and fixed code snippets using the LLM model, which will be discussed further in section 3.2.
cwe_id in the JSON list 3.1 means the vulnerability type which has been the focus of our research. Common Weakness Enumeration is known as CWE. It is a community-developed list of software and hardware weakness types that serves as a common language and a measuring stick for security tools and issues maintained by the MITRE Corporation [24]. The CWE is used to identify and classify software weaknesses and vulnerabilities, which is our main focus in this research. Table 3.2 shows the used CWEs in the research with their names and occurrence. Mitre Web page [24] offers more specific information about each vulnerabilitiy type (CWE). Note that the dataset contains 78 cases without a CWE flag and is marked as "Not Mapping". A Filter has been implemented in the python project to exclude these cases from consideration.
idx, *repo*, *commit* are not considered in the research, because they do not have a relation to any research question.
vuln_id is the vulnerability ID, it is also called CVE, which stands for Common Vulnerabilities and Exposures. It is a list of publicly disclosed computer security flaws. It provides a standardized identifier for a given vulnerability or exposure, which allows security professionals to access information about specific cyber threats across multiple information sources using the same common name [24]. For example "Infinite Loop, CWE-835" contain a lot of vulnerabilities that lead to an infinite loop. Each vulnerability is defined as a CVE and labeled in the dataset as "vuln_id".

CWE ID	CWE name	occurrence
CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	4
CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	3
CWE-310	Cryptographic Issues 3	1
CWE-287	Improper Authentication	5
CWE-264	Permissions, Privileges, and Access Controls	2
CWE-284	Improper Access Control	1
CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	2
CWE-20	Improper Input Validation	7
CWE-835	Loop with Unreachable Exit Condition ('Infinite Loop')	6
CWE-611	Improper Restriction of XML External Entity Reference	4
CWE-74	Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')	1
CWE-532	Insertion of Sensitive Information into Log File	2
CWE-918	Server-Side Request Forgery (SSRF)	3
CWE-787	Out-of-bounds Write	1
CWE-502	Deserialization of Untrusted Data	1
CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	1
CWE-295	Improper Certificate Validation	1
CWE-352	Cross-Site Request Forgery (CSRF)	2
CWE-862	Missing Authorization	1
CWE-601	Row 21 URL Redirection to Untrusted Site ('Open Redirect')	1
CWE-755	Improper Handling of Exceptional Conditions	1

Table 3.2: The used CWEs in this research

3.2 LLM Model

"gpt-3.5-turbo" has been chosen as a large-language-model to answer the main research question 1.2 and its subordinate questions, which is an updated version GPT-3 from OpenAI [25]. This is the gray box in Figure 3.1. The Generative Pre-trained Transformer was chosen because GPT is a large-scale language model that has been pre-trained on massive amounts of text data [4], and the reason of choosing gpt-3.5-turbo is because of OpenAI's recommendation in their documentation. They advice to use this engine over the other GPT-3.5 models because of its lower cost and improved performance [25]. Furthermore, it powers the popular ChatGPT model [22], which holds today the top position in code generation and comprehension.
With the access point interface (API) of OpenAI, and the `openai` python library, communication to gpt-3.5-turbo engine was established.

3.3 Code Review Generation Process

Code reviews can find errors that are not found by testing or static analysis. For example, potential vulnerabilities in the libraries used [9]. According to this, the LLM model was given a prompt to generate a code review of a code snippet that was also written within the prompt, and the other prompt included additionally a focus on security aspects.
According to *Tom B. Brown*, the prompt creation process involved attempting to build the prompt with the same structure (in an imperative tone). The first prompt is built as follows **provide a detailed code review of the following code snippet**.
It is crucial to keep in mind that this prompt only provides a summary or review of the code, without taking security into account. To steer the behavior of the GPT model, additional so-called **system messages** were implemented based on the research "In ChatGPT We Trust? Measuring and Characterizing the Reliability of ChatGPT" from *Xinyue Shen and Zeyuan Chen* [6].
The system message, as recommended in OpenAI's official documentation [25], was utilized. Additionally, the API request incorporated the following system roles that align with our project requirements:

- **You are a helpful assistant with analyzing Java code snippets.**
- **You are a helpful assistant that detects security vulnerabilities.**

In response to research question number 2, the behavior of the GPT model was tested with an alternative prompt. The first prompt was modified by including the phrase "and focus on security aspects," while keeping the system messages unchanged. The code review generation process was then repeated, yielding different results. The outcomes for both prompts can be observed in chapter 4. The used prompts for the code review generation in this work:

- **First Prompt: provide a detailed code review of the following code snippet (followed by the code snippet)**
- **Second Prompt: provide a detailed code review of the following code snippet, and focus on security aspects (followed by the code snippet)**

Additionally, consideration was given to the rate limits set by the gpt-3.5-turbo model, allowing a maximum of three requests per minute. To address this limitation, the implemented solution involved utilizing OpenAI's provided method [25], which is Tenacity Apache 2.0. It is a retrying library, written in Python, to simplify the task of adding retry behavior to just about anything [30]. The function waits 1 to 60 seconds to send another API request. After 6 unsuccessful trials an error exception is returned. This intelligent approach effectively manages the timing of the model's requests to guarantee adherence to the rate limit.
A code review was conducted on the vulnerable code snippet in the dataset, followed by a code review of its fixed version using the same prompt. The results have been uploaded to the Gitlab repository. [21].

3.4 Manual detection

The small blue box in Figure 3.1 describes the manual vulnerability detection in the generated code reviews.

In order to address the research questions, an investigation was conducted into each code review generated by the model to identify security vulnerabilities. The primary focus was on the assigned "Common Weakness Enumeration (CWE) labels" corresponding to each code snippet. For instance, if the code snippet exhibited vulnerability to CWE-20 and indications were found in the code review sentence such as "The function might be vulnerable because there is no Input validation in the If-statement," it was marked as **Yes**; otherwise, it was marked as **No** if no hints were observed.
It is also important to note, that the model generates in a many cases sentences followed by the word "**unclear**" or for example "**No context about the class used in the code, the class might be vulnerable to the CWE-xx**", which means that GPT requires more information about the code snippet to catch the vulnerability. Such sentences are not considered as vulnerability hints to the type CWE-xx, and the flag remains as "No".

The findings were compiled into a CSV file that covered the 50 cases, recording a pair of Yes/No flags for each case. The first flag indicated the presence of CWE in the vulnerable code, while the second flag indicated the presence of CWE in the fixed version. Additionally, two additional Yes/No flags were incorporated to indicate the existence of other vulnerabilities in the generated code reviews. The corresponding reference for each Yes-flag was documented by extracting the sentence from the review that provided a clue about the identified vulnerability. For No-flags, the case was indicated as "**no apparent security vulnerabilities**". These references were noted in separate columns within the CSV file.
Furthermore, a column is created to document the differences between the vulnerable and the fixed code snippets to easily find the changed line and another column to capture the sentences from the code reviews describing these differences. If the code review does not describe the changed line, the case is noted with "**No review about the change**".
The CSV file contains the following columns:

`Case number; CWE-ID; description; CVE-ID; (vulnerable code) Hints to the target CWE; reference; (vulnerable code) Hints to other vulnerabilities; reference; (Fixed) Hints to the target CWE; reference; (Fixed) Hints to other vulnerabilities; reference; code difference; difference as generated text`.

During the manual analysis, the code snippets are prepared by formatting them and printing them alongside their generated code reviews into a text file with the corresponding CWE flag.
To ensure the integrity of the data, the file is set as read-only, preventing any modifications. Subsequently, after reading the entire file and recording the two flags (CWE found, other vulnerabilities found) of each corresponding code review in the CSV file, the tables and the text files were uploaded on the GitLab repository [21].

3.5 Evaluation Metrics

To answer the research question number 1, the metrics are computed, which are used in machine learning and data analysis to evaluate the performance of the LLM model. Evaluation metrics play an important role in evaluating the performance and effectiveness of predictive models. They are used to assess the performance and effectiveness of the LLM models especially in detecting vulnerabilities and provide a quantitative measure that helps us understand the capability of this model to trust in its predictions [10][2].
Commonly used evaluation criteria include confusion matrix, accuracy, precision, recall, and F1 score.

3.5.1 Confusion Matrix

The confusion matrix is a table that is often used to describe the performance of a classification model on a set of test data for which the true values are known. [19] It allows visualization of the performance of an algorithm by showing the number of true positives, false positives, true negatives, and false negatives.

- **True positives** are the number of correct positive predictions, which is in this case the number of code reviews containing vulnerability hints, where the code is actually vulnerable. [19]
- **False positives** are the number of incorrect positive predictions, which is in this case the number of code reviews containing vulnerability hints, where the code is actually fixed. [19]
- **False negatives** are the number of incorrect negative predictions, which is in this case the number of code reviews without vulnerability hints, where the code is actually vulnerable. [19]
- **True negatives** are the number of correct negative predictions, which is in this case the number of code reviews without vulnerability hints, where the code is actually fixed. [19]

The results of tables 4.2 and 4.3 are visualized using the `matplotlib.pyplot` Python library. The results were presented in a statistical model, highlighting the color and structure to facilitate future discussions. Please refer to Figure A.1a, A.1b for the first prompt and Figure A.2a, A.2b for the second prompt.
In these visualizations, the colors orange and blue are used. The presence of the color blue indicates that the model provided hints related to the targeted CWE in the generated code review, while the color orange indicates the absence of CWE in the code review. The x-axis represents the names of the CWEs, while the y-axis represents the occurrence frequency of these CWEs in either the vulnerable code A.2 or fixed code A.1.

3.5.2 Precision

Precision is an evaluation metric used to assess the accuracy of positive predictions made by a classification model. It quantifies the proportion of correctly classified positive instances (true positives) out of all instances predicted as positive (true positives plus false positives) [23].
In other words, precision focuses on the quality of positive predictions by measuring the ability of the model to avoid false positives. It provides insights into how reliable the model is when it declares an instance as belonging to the positive class. A high precision score indicates a low rate of false positives, meaning that the model is more precise in identifying positive instances correctly [23].
Precision is essential when the cost of false positives is high. In such cases, achieving high precision minimizes incorrect positive predictions. For example, to prevent spending extra time detecting the code again carefully if the test suggests that the given code is vulnerable. This process would cause us time and unnecessary code changes, leading to other potential security problems.
The formula to calculate precision is:

$$\text{Precision} = \frac{TP}{TP + FP}$$

where TP represents true positives (correctly classified positive instances), and FP represents false positives (instances incorrectly classified as positive).
Regarding the mathematical perspective, increasing the FP value to infinity, the minimal precision is reached. In contracts, setting FP to 0 and TP to a constant unequal zero, the precision would be equal to 1, which is the maximum of this metric.

3.5.3 Recall

Recall, also known as sensitivity or true positive rate, is an evaluation metric used to measure the ability of a classification model to identify all positive instances which are also correct. It quantifies the proportion of true positives identified by the model out of all actual positive instances in the dataset [23].
In other words, recall focuses on the ability of the model to avoid false negatives by capturing as many positive instances as possible. It provides insights into the model's ability to correctly identify positive instances, regardless of any false positive predictions it might make [23].
Recall is more important than precision in scenarios with a high cost of false negatives. For example, running a static analysis tool to detect the vulnerability in a code, if the tool misclassifies a minor vulnerability in the code, an attack can happen, which could have been avoided. In such cases, high recall is crucial to minimize missed positive instances. The formula to calculate recall is:

$$\text{Recall} = \frac{TP}{TP + FN}$$

TP represents true positives (correctly classified positive instances), and FN represents

false negatives (instances incorrectly classified as negative but should be positive).
Looking into the mathematical side, raising the FN to infinity would minimize the recall, which will be a disaster. In contrast, to achieve the maximum value of the recall, keeping FN very low (toward zero) is necessary.

3.5.4 Accuracy

Accuracy is an evaluation metric used to measure the overall correctness of predictions made by a classification model. It quantifies the proportion of correctly classified instances (both true positives and true negatives) out of the total number of instances in the dataset [23].
In other words, accuracy assesses how well the model correctly classifies instances, regardless of the class label. It considers both true positives and true negatives and indicates the overall correctness of the model's predictions [23].
Accuracy is widely used as a performance metric when the dataset is balanced, meaning the number of instances in each class is roughly equal. However, it can be misleading in class imbalance, where one class dominates the dataset. In such cases, accuracy may give a high score even if the model performs poorly in the minority class. However, it is balanced in this case because the number of instances of each class (binary: vulnerable or fixed) is 50.
The formula to calculate accuracy is:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Overall, accuracy generally measures the model's correctness in predicting instances across all classes. It is a commonly used metric to evaluate the overall performance of a classification model when the dataset is balanced. Nevertheless, it is important to consider one last metric, the F1 score.

3.5.5 F1-score

F1-score is the harmonic mean of precision and recall, which provides a single score that balances precision and recall. It ranges from 0 to 1, where a score of 1 represents perfect precision and recall, and a score of 0 means the model is not predicting any positive instances correctly [23].
The F1 score is calculated as the harmonic mean of precision and recall, providing a balanced measure of the model's performance:

$$\text{F1 Score} = 2 * \frac{Precision * Recall}{Precision + Recall}$$

The F1 score is important when false positives and false negatives are equally important. It helps evaluate the model's overall performance by considering the ability to identify positive instances and avoid false positives correctly. [23]

3.6 Text analysis

This section introduce methods, that can answer the research questions 3, 4, and 5.
The generated code reviews were analyzed to extract meaningful insights. Various text analysis metrics, such as word count (question 3), TF-IDF matrix (question 5), and text similarity (question 4), were extracted between the vulnerable code review and the fixed code review.

3.6.1 Word- and Character count

The number of generated words in the code reviews and the number of characters in the code snippets are crucial to answer the research questions number 3 and 4, which is to find out if the performance of GPT is dependent on its output's size or the input's size, and also to find out if the model generates more or less if the code is actually secure. A thorough examination will be conducted on the word count and code character counts in cases of true and false predictions, to extract information when the model makes true and false predictions.
The generated text was tokenized using the "word_tokenize" function from the NLTK library, which splits the words separated by " ", and stores them in a list. The length of the list is the number of words in the specific text. The character count is basically the number of characters in the code without the space character " ", to prevent getting large numbers. We also calculated the number of code characters between the vulnerable and the fixed code snippets with the help of DiffChecker website [20]. Chapter 4 presents the outcomes.
This analysis enables us to obtain statistical insights into the outputs generated by the GPT model, with the aim of evaluating its ability to provide code reviews that exhibit strong attention to security aspects in programming. To achieve this, we examine the precise manner in which these texts are generated and how they relate to one another, therefore word count plays a crucial role in conducting this analysis.
The analysis also incorporates a significant aspect of word count, particularly focused on specific words in the generated texts. This aspect, commonly referred to as "co-occurrence," examines the occurrence of words related to security areas. Further exploration of this topic will be discussed in the TF-IDF section. 3.6.3

3.6.2 Text Similarity

To answer research question number 4 we calculated the text similarity between the vulnerable and the fixed code snippet using the SpaCy library. SpaCy uses word vectors to determine how similar two texts are to each other.
The similarity score is calculated using the cosine similarity between the word vectors of the texts.[27]
The text similarity is calculated with the following steps:

1. Tokenization: The text is first tokenized, which means that it is split into individual words or tokens. [28]
2. Vectorization: Each token is then converted into a vector representation, which is a mathematical representation of the token's meaning. This is done using a

pre-trained language model, such as GloVe or Word2Vec. [28]

3. Similarity calculation: The similarity between the two pieces of text is then calculated using the cosine similarity between the vector representations of each token. The concept of cosine similarity involves measuring the similarity between two vectors in an inner product space. This is achieved by calculating the cosine of the angle between two embeddings and determining whether they are pointing in similar directions. If the embeddings are pointing in the same direction, the angle between them is zero, and their cosine similarity is 1. If the embeddings are perpendicular to each other, the angle between them is 90 degrees, and the cosine similarity is 0. Finally, when the angle between them is 180 degrees, the cosine similarity is -1. [28]

$$\text{similarity(A, B)} = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$$

4. Aggregation: The similarity scores for each token are then aggregated to produce an overall similarity score for the two pieces of text. This can be done using a variety of methods, such as taking the average or maximum similarity score. [28]

We chose SpaCy over NLTK in calculating Text similarities, because SpaCy has support for word vectors whereas NLTK does not. As spaCy uses the latest and best algorithms, its performance is usually good as compared to NLTK. In word tokenization and POS-tagging SpaCy performs better [29].

3.6.3 TF-IDF matrix

To answer the research question number 5, we calculate the TF-IDF matrix short for "Term Frequency-Inverse Document Frequency" to get a score for each word in the code review, indicating its importance in the vulnerable or fixed code reviews and also in both of them.
TF-IDF is a well-known method for quantifying the significance of words within a collection of documents. It is widely utilized in the fields of Information Retrieval and Text Mining, TF-IDF has become a prevalent technique for analyzing and extracting meaningful insights from textual data [31].
For the implementation, We removed the stopwords from the code review and returned the remained words to their original form using the lemmatize function from the NLTK.stem python library. We converted the words to lowercase and used TfidfVectorizer from sklearn's Python library to calculate the TF-IDF vectors, the vectors are converted into a Pandas DataFrame. The column names of the DataFrame are set to the feature names obtained from the vectorizer. We transposed the matrix to make the words the matrix lines and the code reviews the matrix columns. Then we printed only the top 20 common words due to the large size of the matrices because we received on average after 20 words a lot of zeros or very small numbers.
TF-IDF matrix is calculated by combining two components, which are the term frequency (TF) and the inverse document frequency (IDF).

The Term Frequency (TF): It measures the occurrence of a specific term within a document (or in our case a code review). It calculates how frequently a word appears

in a vulnerable or fixed code review relative to both vulnerable and fixed code reviews in each case. TF is calculated using the following formula:

$$TF(\text{term, document}) = \frac{\text{Number of occurrences of the term in the document}}{\text{Total number of terms in the document}}$$

In our case, the document is either the generated code review of the vulnerable or the fixed code in one case. The Terms are the words in one of them. [31]
Inverse Document Frequency (IDF) The inverse document frequency measures the significance of a word across the vulnerable and the fixed code reviews. It helps to identify words that are more unique or informative by giving higher weights to words that appear less frequently across the reviews of the vulnerable and the fixed code snippets. IDF is calculated using the following formula:

$$IDF(\text{term, corpus}) = \frac{log(\text{Total number of documents in the corpus})}{\text{Number of documents containing the term}}$$

The corpus means, both code reviews of the vulnerable and the fixed code snippets. [31]
TF-IDF Calculation The TF-IDF score is obtained by multiplying the term frequency (TF) of a term in a document by its inverse document frequency (IDF) across the corpus. This calculation emphasizes terms with a high frequency within a code review (indicating their importance within that code review) while also considering their rarity across the all documents (code reviews). The formula for calculating TF-IDF is:

$$\text{TF-IDF(term, document, corpus)} = \text{TF(term, document)} * \text{IDF(term, corpus)}$$

We performed the TF-IDF matrices in two ways:

First, **TF-IDF in one case:** To determine detailed text comparison results for both the vulnerable and fixed versions of the code, we analyzed the TF-IDF matrix for the "vulnerable case" and "fixed case" documents. The TF-IDF matrix is available in a JSON file named `analysis_output.json` and `combined_text.txt` in our Git repository [21], which can be accessed through `output_1` directory for the first prompt and `output_2` directory for the second prompt.

Second, **TF-IDF in general case:** In order to measure the impact of the prompt on code reviews, we find it useful to analyze the TF-IDF of the "vulnerable set of all vulnerable cases using the first prompt" as a corpus, and also for the fixed set. We do the same for the second prompt. By selecting the top 20 words that appear and considering the 50 cases in the corpus, we generate a 20x50 matrix. To improve accuracy, we sum up all rows in the matrix to obtain a single column that indicates the importance of each word across all cases in the corpus. We have the 4 matrices A.4a A.4b, A.5a A.5b in chapter A.

3.7 Threats Validity

1. External Validity:

- A limitation of this research is that it focuses exclusively on analyzing cases involving Java code snippets processed through the GPT-3.5-turbo engine. Therefore, the findings and conclusions drawn from this study may not be directly applicable to code snippets written in other programming languages or processed by different language models. The generalizability of the results to a broader range of programming languages and language models should be considered with caution.

2. Sampling Bias:
   - Our analysis was based on a subset of the dataset, specifically 50 cases out of the total 196 cases. Additionally, we focused our discussion on 21 common weakness enumerations (CWEs) that were identified within the dataset. It is crucial to acknowledge that the dataset contains a larger number of CWEs beyond the scope of our analysis.
   - One limitation of this research is the relatively low redundancy of the discussed vulnerabilities and their corresponding code reviews. This limited redundancy can potentially impact the overall performance of the model. To obtain more detailed metrics and enhance the robustness of the analysis, we recommend conducting further testing and analysis on a larger number of cases to generate a greater variety of model predictions.
   - We do not have additional context about the variables, functions, or classes used in the individual code snippets, this might reduce the capability of the GPT model to detect vulnerabilities in the code snippets.
3. Internal Validity:
   - The limitation of this study lies in the inability to execute the code snippets for comprehensive vulnerability analysis due to the lack of additional contextual information. Consequently, the task becomes challenging for GPT to accurately identify security vulnerabilities or other code-related issues and specifically detect the targeted vulnerability.
4. Human Bias:
   - The primary focus of this research was to carefully examine each code review and identify any indications of Common Weakness Enumeration (CWE) within the text. However, it is important to acknowledge that this humanized classification represents a significant limitation of this study.
5. Limitations of GPT Model:
   - Understanding of Contextual Programming Knowledge: GPT's understanding of programming concepts and contextual programming knowledge may be limited. The GPT-3.5-turbo model has a significant limitation in that it is trained up to September 2021 and it can only accommodate a maximum of 4096 tokens per prompt [25]. This poses a challenge when analyzing code snippets that exceed this token limit, as they cannot be effectively processed or analyzed within the model's constraints. This limitation is also considered in this research [14].

Based on this limitation in our research, the model may struggle to accurately capture the intricacies of code vulnerabilities or security-related nuances, leading to potential inaccuracies in the generated code reviews.

- Limited Training Data on Security Aspects: We use in this research the real-world dataset, which is the 175 Billion parameter dataset [2] [14]. Despite all this size, the model may still not be mature enough to recognize all the CWEs in various code snippets.
  Last but not least, the GPT-3.5-Turbo model does not always pay strong attention to system messages [25], which might influence the behavior of the GPT model in vulnerability detection using the first prompt in a huge way.

Chapter 4

Results

This chapter presents the findings and outcomes of the evaluation metrics and the text analysis conducted to find answers for the research questions we asked before, beginning with 1 and ending with 6. We present the results, which find assess the performance of the GPT model in generating code reviews with a focus on security aspects. In the chapter 5, we will discuss the implications and significance of these findings, provide a comprehensive interpretation of the outcomes, and finally answer the research questions.

4.1 Manual Detection Results

For each prompt, we generated a dedicated CSV file. The first prompt focused on generating code reviews, while the second prompt emphasized the inclusion of security aspects. Table 4.1 shows the number of positive responses in each column for the first prompt. The columns are defined as follows:
CWE hint: A Hint about the targeted CWE was found in the code review of the vulnerable/fixed code snippet.
Other vulnerabilities: Other vulnerabilities were found in the review of the vulnerable/fixed code snippet.
The prompts are noted as follows:
Prompt 1: provide a detailed code review of the following code snippet. (followed by the code snippet)
Prompt 2: provide a detailed code review of the following code snippet, and focus on security aspects: (followed by the code snippet)

This table shows how many code reviews contained correct vulnerabilities and also other vulnerabilities.

Table 4.2 and table 4.3 show the confusion matrices we discussed in subsection 3.5.1 chapter 3

Considering both code reviews of vulnerable and fixed codes snippets, we have 4 prediction classes as follows:
Yes/Yes: The vulnerable code review is correctly predicted, but the fixed version is not. **Yes/No**: Both vulnerable and fixed code reviews are correctly predicted. **No/Yes**: Both

Prompt	vulnerable		fixed	
	CWE hint	Other vulnerabilities	CWE hint	Other vulnerabilities
Prompt 1	13	13	8	18
Prompt 2	27	39	17	34

Table 4.1: Number of positive responses in each class

Actual	Predicted	
	vulnerable	**not vulnerable**
vulnerable	13	37
fixed	8	42

Table 4.2: Confusion Matrix for the prompt: Generate code review

Actual	Predicted	
	vulnerable	**not vulnerable**
vulnerable	27	23
fixed	17	33

Table 4.3: Confusion Matrix for the prompt: Generate code review, with focus on security aspects

vulnerable and fixed code reviews are wrongly predicted. **No/No**: The fixed code review is correctly predicted, but its vulnerable version is not.
These classes are presented in table 4.4:

Prompt	Yes/Yes	Yes/No	No/Yes	No/No
Prompt 1	4	9	4	33
Prompt 2	10	17	7	16

Table 4.4: Prediction statistic table

4.2 Evaluation Metrics

All the metrics explained in the previous chapter 3 are typically used together to give a more comprehensive evaluation of a model's performance. Table 4.5 shows the results of previous metrics for the two prompts we used in this research.

We see that the recall increases 28% using the security detailed prompt. On the other hand, we see small changes in the precision and accuracy. The accuracy increases by 5% and the precision decreases by 0.6 %. Due to the strong increas of the recall, the F1 score increases by 20.8% using the second prompt.

Metric	Prompt 1	Prompt 2
Precision	0.62	0.614
Recall	0.26	0.54
Accuracy	0.55	0.6
F1 Score	0.366	0.574

Table 4.5: Classification Metrics

4.3 Text Analysis Results

Table A.2 shows the results for the first prompt 4.1. It includes the vulnerability flags, which define if a hint to a CWE is found in the code reviews (Yes: found, and No: No hints). These flags are located in the columns **vuln**, which means the code reviews of the vulnerable code snippets, and **fixed** for the code review of the fixed code snippets. For each one of them we added the word count results in column **#words** and the code character count in column **#char** and in the last two columns we attached the number of changed characters **char_diff** between the vulnerable code and its fixed version and the text similarity between the code review of the vulnerable and the fixed code snippet. We filled the table out with 50 cases, each case has a number shown in column **Case**. The CWE type for each case is written in the column **CWE ID**. Table A.3 shows the same results using the second prompt 4.1.
Considering the word count using the first prompt, we have received on average 367 generated words describing vulnerable code snippets and 358 words describing secure code snippets. The standard deviations are 194 and 150 for the vulnerable and secure code snippets, this indicates that the sizes of the generated texts differ greatly from each other. The maximum and minimum show this strong difference with Max: 1390 and Min: 149 words for vulnerable code reviews and Max: 898 and Min: 75 words for secure code reviews.
Surprisingly we received lower maximum values (521 and 595) using the second prompt, but the mean does not decrease significantly by 29 words in the vulnerable case and 11 words in the fixed case. For now, we know that the number of generated words does not exceed 600 words using the detailed prompt.
In the final column, we can observe the level of similarity between the generated code review for a vulnerable code snippet and the generated code review for the same code snippet after implementing a single bug fix. Investigating the first prompt, we received an average similarity of about 93.1%, a maximum of 98.8%, and a minimum of 66.7%. Using the detailed prompt, we get higher similarity about on average 97.3%, a maximum of 99.1 %, and a minimum of 89.36%., which yields that the generated code reviews using the second prompt are more similar to each other than using the first prompt.
The maximum of character count of the code changes is 198 and the minimum is not 0 as you see in the table, it is the indentation in the code snippet. Negative numbers mean that we removed characters from the vulnerable code and not having a sign is a character added to the vulnerable code. We also calculated the mean and the standard deviation, which are presented at the end of the table. we added in the tables A.2 and

A.3 the mean, standard deviation, maximum and minimum values for word count, code character count, and text similarity.

It is important to also visualize the results of the true and false predictions. Table 4.6 shows us the average word count for true and false predictions. We also obtained the corresponding standard deviation values in table 4.7.

	True positive	**False positive**	**True negative**	**False negative**
Prompt 1	319	384	356	370
Prompt 2	337	338	330	381

Table 4.6: Mean of word count results of fixed (negative) and vulnerable (positive) cases

	True positive	**False positive**	**True negative**	**False negative**
Prompt 1	97	215	158	103
Prompt 2	73	74	87	53

Table 4.7: Standard deviation of word count results of fixed (negative) and vulnerable (positive) cases

The general TF-IDF matrices A.4a, A.5a, A.4b and A.5b are presented in chapter A. We could not include all of the TF-IDF matrices (for each case, comparing vulnerable and fixed versions) in the paper because they are 50 large matrices, which can not fit in this paper. Therefore, we have provided a GitLab repository [21], where the source code and the output are accessible for further exploration. We will mention some relevant examples, in chapter 5 to discuss the differences between the vulnerable and the fixed version.

Chapter 5

Discussion

In this chapter, we will analyze the results obtained in the previous chapter (refer to chapter 4) and address the main research question (refer to 1.2) along with its subordinate research questions.
We begin with question number 1, which puts light on how the GPT model performed in generating code reviews that identify security vulnerabilities in code snippets, therefore we will take a look into our results 4.2, 4.3 and 4.5.
We will also answer questions 2 and 5 comparing the code review generations using the two prompts 4.1 and 4.1, and find out which prompt performed better based on the evaluation matrices and the general TF-IDF matrices A.4a, A.5a, A.4b and A.5b to find out how much are the code reviews related to the security field.
Then we will discuss the word count of each generated code review and the character count of the corresponding code snippets and find out if the performance of the GPT model depends on the size of the code or the number of generated words to find answers for question number 3.
Furthermore, We will focus on research question 4 to examine the model's general behavior in generating the code reviews in some false prediction cases and try to interpret the reason for the misclassification. In other words, we will discuss cases where wrong predictions were unexpected.
Lastly, we will compare the vulnerable code snippet's code review with its fixed version in cases that belong to Yes/Yes or No/Yes classes 4.4. After this discussion, we might be able to answer the last research question 6 and the main research question 1.2.

5.1 Evaluation Metrics and Text Analysis

Using the first prompt in the code review generation, we see that the GPT model generates inefficient code reviews. We found correct vulnerabilities only in 13 vulnerable cases out of 50 cases 4.2 and achieved an accuracy of 55% and a precision of 62%. The recall is at 26% because of the strong presence of vulnerability types, which were falsely predicted by the GPT model, for example, CWE-853, CWE-918, and CWE-352 and other vulnerability types presented in Figure A.1a.
It is clear that there are no vulnerabilities from type CWE-835 (infinite loop) detected,

and due to the strong occurrence of this type (6 times), we achieved lower evaluation metrics than we expected. In other words, if we replace these code snippets with code snippets that are vulnerable to different vulnerability types, such as CWE-20 (input validation), which is the most truly predicted vulnerability using the second prompt A.2a, we might get better performance.
As discussed in chapter 3, receiving low recall is a complete disaster because the permanent misclassification and missing important security hints in the code reviews might lead to several security attacks, and the reason is the faith and trust in generating good code reviews by the GPT model.

The second prompt achieved a higher recall because of the increased number of code reviews of vulnerable code snippets containing relevant and correct warnings. We observed 14 more true positives, and the recall improved from 26% to 54% 4.5 by adding the sequence "and focus on security aspects" to the input prompt. The recall's huge jump (+28%) presents a better performance in the vulnerable set, but the accuracy and precision have not changed significantly. The accuracy (60 %) improved by 5% 4.5, which indicates that the second prompt is better than the first one, although the difference is small. In addition, the precision (61.4%) became 0.6% less. The reason for the precision lowering is that the GPT model generates unnecessary warnings in the code snippets, which are secure and do not contain this vulnerability, leading to false predictions in the fixed code reviews. That also explains the small improvement in accuracy compared with the recall.

We saw in chapter 2 that CodeBERT achieved higher metrics results: which are 95% accuracy, 96% precision, 93% recall, and 94% F1 score in multiclass vulnerability classification. These results are way better than our results. Of course, they used other vulnerabilities and tested C/C++ programs, which is not similar to our work, but maybe testing CodeBERT instead of gpt-3.5-turbo might bring us better results.
Our resulting metrics are not surprising because *Anton Cheshkov, Pavel Zadorozhny, and Rodion Levichev* also tested Java code snippets using the same engine we used in this research, which is the gpt-3.5-turbo model. The F1-score is 4.6% higher than our F1-score using the second prompt 2.1. It is important to note that the two works are not similar because the authors request GPT to detect vulnerabilities rather than asking it to generate code reviews.
Using a detailed security prompt generates more security vulnerabilities in the code review, but that does not mean the model performs better. It generates vulnerabilities in many code reviews of secure code snippets, meaning we might spend extra time reviewing the code again and fixing bugs that do not exist.
To answer the research question number 5, we checked out the general TF-IDF matrices A.4 and A.5, which where discussed in section 3.6.3 to discover if we obtain more security-related code reviews using the second prompt 4.1 compared with the first prompt 4.1.
Upon analyzing the TF-IDF matrices of the code reviews of **vulnerable code snippets** A.4a and A.5a, it becomes evident that the code reviews generated using the second prompt exhibit a higher occurrence of security-related words compared to those generated using the first prompt. For example, the words "**attack**", "**validation**", and

"**injection**", which receive the values (2.368386, 2.024380, and 1.828469) are present in the list of top 20 occurring words in the generated code reviews of vulnerable code snippets using the second prompt. On the other hand, these words are not present in the matrix of the code reviews generated using the first prompt, which decreases the focus on security elements in the first code reviews. Furthermore, "**security**" is repeated several times in the second prompt and receives the TF-IDF value of 4.06. Nevertheless, using the first prompt, it receives the value of 1.67, which indicates that this word is not important in the first code reviews as in the second.
Furthermore, the top 20 occurring words in the code reviews using the second prompt receive higher TF-IDF values than the first prompt. In other words, the word "**code**" is the most occurring word with a value of 6.79, and the word "**handling**" is in place 20 with a value of 1.761 using the second prompt. Compared with the first prompt (Max = 5.040051 "**method**", Min = 1.51918 "**parameter**").
The TF-IDF matrices of 50 secure code snippets also reveal interesting results. Table A.4b and A.5b substantiate this proposition. Notably, the second prompt consistently features a higher frequency of security-related words, such as "security" (3.43), "user" (2.55), "password" (2.38), "validation" (2.31), "access" (2.22), "attack" (2.1), and "vulnerability" (1.89). In contrast, the first prompt yields fewer security-related terms, including "security" (1.79), "user" (1.86), and "check" (1.72). The difference in frequency between the two prompts is clear, highlighting the focus on security in the code reviews generated using the second prompt.

Table A.2 and A.3 present the difference between the vulnerable and fixed code reviews. There is a maximum difference of 198 characters; for this maximum, we obtained a text similarity of 90.3 using the first prompt, but modifying the prompt to focus on security aspects increased the text similarity for this case up to 98.5 %. Here is another example to clarify: the minimum number of text similarities using the first prompt is 66.7 %, and this is case 17. Case 17 achieved, using the second prompt, a text similarity up to 98.4%. This huge difference in the text similarity means that GPT generates almost the same code reviews of the vulnerable and the fixed code snippet because it is not describing the code like the first prompt. Yet, it enumerates only security concerns or vulnerabilities.
Using the first prompt, we asked GPT to generate code reviews without explicitly asking it to focus on security vulnerabilities. It described the code but did not focus on every line. For example, in case 46 A.2, GPT generated for this code containing 7837 characters a code review containing 1390 words describing the parameters and function calls in detail. After fixing the code by adding 59 characters, GPT generated only 750 words, a big difference from the vulnerable code review. The reason is that GPT ignores a lot of lines in the code. While using the second prompt made this difference smaller. GPT generated for this case using the second prompt 311 words in the vulnerable case and 281 words in the fixed case. The reason is that the code reviews contain only issues and warnings rather than a detailed explanation like the first code reviews. That is why the average text similarity score between the vulnerable and the fixed code reviews using the second prompt is 97.3 %, and the first prompt is 93.1 % A.3, an increase of 4.2%.

We found after this observation answers to research questions number 2, 3 and 5, con-

cluding that modifying the prompt and adding the sequence "focus on security aspects" influences the performance in generating good code reviews containing correct warnings about security vulnerabilities in the code snippets and increases the **attention** to security. The code reviews became more security-related, as we saw in the TF-IDF matrices. Furthermore, the detailed prompt made the vulnerable and the fixed code reviews more similar. The change in the prompt improves the performance by 5% but is not enough to suggest that a detailed prompt can make the GPT trustworthy because of the false warnings in the code reviews about the fixed code snippets.

5.2 Word Count and Code Character Count

Reviewing table A.2, we see that the code size and the number of words might influence the code reviews generated using the first prompt. If the code character count is much larger than the number of generated words, which means that the GPT model does not review all the lines in the code and can lead to missed vulnerabilities in the code review. For example, in case 40, the code snippet containing 1266 characters is vulnerable to the CWE-20 (Input validation type), but there is no presence of vulnerabilities from this type in the generated code review, which contained 251 words. The word count of the generated texts and the character count of the corresponding code snippets reveal a significant difference.
This observation is reinforced by case 37, 'CWE-287: Improper Authentication', with 1243 characters in the code snippet A.2, and 'CWE-918': Server-Side Request Forgery (case 27)A.2, with 2109 characters in the code snippet. The generated reviews consist of only 191 words for CWE-287 and 275 words for CWE-918.
Another example is the case 46 A.2, this code snippet is vulnerable to the CWE-352 type and includes 7837 characters. The generated code review using the first prompt contains 1390 words. Still, we couldn't find any vulnerability hints about the cross-site request forgery (CSRF).
These code reviews are ineffective for a thorough description of the code's security implications. In other words, misclassification can happen if not the entire code snippet is reviewed line by line with focuses on every call, that means if the code review's length is far more smaller that the code's size, as we saw in cases 40, 37, 46, and 27 A.2.
We state that GPT generates incomplete code reviews for code snippets over than 1000 characters.
To prove this statement we calculated the number of code snippets, which have more than 1000 characters. We observed the following table, which displays the result with the vulnerability flag:

Table 5.1a shows the we have in total 19 code snippets in our research, that contain more than 1000 characters. The probability of getting ineffective code reviews of large code snippets (>1000 characters) is 15.8%, and of inefficient code reviews is 84.2%, which is a huge difference about 68.4%.
Reviewing the results in Table 5.1b, we can not generalize the statement we suggested

Flag	Code snippets	probability
Yes	3	15.8 %
No	16	84.2 %
Total	19	

(a) Using the first prompt (1) 4.1

Flag	Code snippets	probability
Yes	10	52.6 %
No	9	47.3 %
Total	19	

(b) Using the detailed prompt (2) 4.1

Table 5.1: Number of code snippets that contain more than 1000 characters

before, because of the probability's closeness (difference = 5.3 %).
In other words, adding a detailed prompt made the GPT model focus on the security aspects instead of explaining the code, so the word generation is not wasted on code lines which are not relevant to security. They are generated only if vulnerabilities or issues are present in the code.

The performance of the model using a **security detailed** prompt might not be influenced by the code character count.
Therefore the statement handles just cases where the prompt only requests the GPT model to generate code reviews without an explicit focus on the security aspects.
This observation answers research question number 3, suggesting that if the code is too large, GPT might generate with a normal "code review generation" prompt a small code review, which is inefficient because it might not contain all the security vulnerabilities in the code. In addition, it is very likely to get "false negative" predictions if the code has over than 1000 characters.

Nevertheless, the standard deviation of the word counts of the vulnerable code reviews using first prompt is 194, and using the second is 74. The reason is that GPT describes the code snippet in some cases in detail; the maximum word count is 1390. In other cases it generates a small code review; the minimum word count is 149. This is not surprising because we requested GPT to generate a code review without an additional focus. The security focus is only present in the system messages as mentioned in chapter 3, but gpt-3.5-turbo does not always pay attention to them.
A code review with a small security focus is not always capable of showing vulnerabilities in the input code, especially if it has a low number of words, GPT might ignore some lines in the code, which leads to missclassification as mentioned before.
The reason why the standard deviation (74 words) of the code reviews generated by the second prompt A.3 is lower than the first one A.2 is that the code reviews do not describe the code snippet anymore. They only mention security vulnerabilities with a short explanation, or in many cases an enumeration of security issues discovered in the code, therefore they have similar word counts and similarity scores.
Because of that, we found that the word count value of code reviews, which are generated by a security detailed prompt does not provide with further performance information about the GPT model in generating informative code reviews.

We discovered through table 4.6, which calculates the mean of generated words in true

and false predictions, that the GPT model in the true predicted cases generates shorter code reviews than in false predicted cases.
It generates using the first prompt code reviews containing correct predictions with an average word count of 337.5 words and code reviews with false predictions with an average word count of 377 words; these are 39.5 words more.
This observation does not change by using the second prompt or considering the fixed cases. The average word count of false predictions remains higher than the true predicted ones; 333.5 words in true predictions and 359.5 words in wrong predictions, 26 words more.
It is important to note that the standard deviations shown in table 4.7 are also high; 97 and 158 words for code reviews with correct predictions, 103 and 215 words for code reviews with false predictions, which enlighten that the ranges of the possible word counts might overlap.
This answers the second part of research question number 3, suggesting that the length of the code reviews is not a differentiating factor in distinguishing between true and false predictions. In other words, the word count alone does not significantly contribute to the model's ability to classify vulnerabilities in the code accurately. Other factors or features need to be considered to improve the performance and reliability of the predictions.

5.3 Behavior in Missclassifications

We examine in this section a selection of code reviews that have been marked with a false prediction flag, for example, code reviews of vulnerable code snippets that do not contain the correct vulnerability and secure code snippets which contain a specific vulnerability. We aim to determine the reason behind the inability of the GPT model to generate correct vulnerability hints and to find answers for the research question number 4.
This section is separated into two subsections. The first subsection considers individual cases. In other words, we look into a simple code snippet with its code review. In the first place, we will discuss the code reviews of the vulnerable code snippets, putting light on the false predictions. In the second one, we focus on false predictions in the code reviews of secure code snippets. In addition, we will analyze the behavior of the GPT model in generating code reviews and try to find reasons for the false predictions. At the end of this section, we will consider the pair-generated code reviews of vulnerable and fixed code snippets. In other words, we will discuss how the code review was before the modification (vulnerable version) and how it became after fixing the code.
We will analyze the behavior of the model and the reaction to the modified changes in the code. Therefore we use the TF-IDF matrices between the (vulnerable and fixed) code reviews to obtain how this behavior changed and find out if it became more or less related to the security field by counting the weighted security-related words in the TF-IDF matrices.
Note: To simplify case referencing, we denote the following information using a triple format (case number, CWE-ID, prompt number).

5.3.1 Code reviews Individual Consideration

As we saw in tables 4.2, the code reviews generated by the GPT model were not efficient, 37 code reviews did not contain a correct vulnerability that is present in the code, and 8 code reviews were suggesting a vulnerability in the code, which did not exist.
As previously mentioned, utilizing a more comprehensive security prompt to generate code reviews did not significantly enhance the accuracy.
Hence, we will delve further into the code reviews to gather insights into the reasons behind the absence of vulnerabilities in the code snippets considered vulnerable, as well as the occurrence of false warnings in secure cases.

Missclassifications in Code reviews of Vulnerable Code Snippets

Relying on the LLM model's security code reviews can be risky. Even though it may suggest that the code is secure, in reality, it could be vulnerable. This poses a significant threat to software systems and increases the likelihood of sensitive information being lost. Additionally, developers may not be able to identify the root cause of the damage, as they had trusted the LLM model's assurance that the system was secure. Therefore we discover vulnerable cases first and then move on to other cases where the secure cases are falsely predicted.
The behavior of the model changes in false predictions, where GPT doesn't find the correct vulnerability in the code. We found that misclassification in both prompts can be caused if **there is no context about the variable implemented in the code**.
For example (12, CWE-835, 1). The issue in the code is hidden in this line:
`for (int i = 0; i < this.rcount; i++)`
The initialized datatype "int" is risky at this position, because of its maximum number, which might be smaller than the variable "this.rcount" causing a wrap-around and the loop will never terminate.
It is difficult for the GPT model to find this vulnerability, because of the low context of the code as the code review says:
`However, to perform a detailed code review, further context and information about the surrounding code would be helpful.`
In other words, if "this.rcount" is defined with a magic number in the code snippet, we might have a chance to see this vulnerability in the code review. This reflects the limitation we mentioned before 3.7, which suggests that the ability of the GPT model might be low in detecting vulnerabilities if we do not have more context about the elements in the code snippets.
This concern appears again (36, CWE-295, 2). GPT's ability to detect the presence of vulnerabilities is limited, saying: "`The security of a SASL protocol depends highly on the mechanism used to negotiate authentication. The code snippet seems to handle the negotiation correctly. However, we do not have enough context to determine if the mechanism employed is secure. If a weak mechanism is used, an attacker could easily impersonate a legitimate user.`"
Stating that the code snippet handles the negotiation correctly is dangerous and means a misclassification in our case, but we see the reason after this sentence, which is not having enough context about the code. Maybe if we write the entire mechanism in the

prompt, we might get a better result, but the model's token limitation is a huge obstacle, as discussed in the methodology Chapter 3.7.
In (49, CWE-755, 2), the model displays a confident hint, which suggested that the Handling exception is secure, which is in reality the exact opposite. The response is generated as follows:

```
3. Proper error handling: The code snippet handles all errors that could
occur during runtime. It uses the TException class to handle exceptions
gracefully, rather than propagating them back up the call stack.
```

It's concerning that GPT is not only misclassifying vulnerabilities but also it's suggesting the opposite - that there is no vulnerability present. This can lead to a false sense of security and decrease trust in LLM models to identify threats in software systems.
Last, we discovered that GPT ignores vulnerability types, such as CWE-502 "Deserialization of Untrusted Data" and CWE-362 "Race Condition" and CWE-295 "Improper Certificate Validation", we couldn't find any vulnerability hints about these types in all generated code reviews.
It is possible for the GPT model to generate more vulnerability hints about some CWE types, such as CWE-79, CWE-22, and CWE-200, but due to the low redundancy of the tested CWE types as mentioned before in the methodology chapter 3.7, we can not safely assume that these types are always presented in the generated code reviews. This research shows that the probability of true prediction for these CWEs is less or equal to 50%.
After reviewing 100 code reviews of **vulnerable code snippets** (50: first prompt, 50: second prompt), we classified the misclassifications into 3 classes. Firstly, the model excludes the existing vulnerability and believes the code is secure from it, and this is the most dangerous class, which reduces the trust in GPT as we saw in (49, CWE-755, 2). Secondly, if GPT requires additional context about the code, which leads to the absence of the vulnerability hint in the code review, such as (36, CWE-295, 2). Thirdly, the GPT model ignores the vulnerability and we do not see any hints about it in the code reviews, such as CWE-502 and CWE-362.

Missclassifications in Code reviews of Secure Code Snippets

Investing time fixing the code based on false predictions in the fixed set is unnecessary. It can lead to wasted effort and time reviewing the code again by a specialist, and the code has already been addressed.
It is important to note that adding the focus on security aspects to the prompt aims to identify vulnerabilities in vulnerable cases, not secure ones. However, the GPT model identifies these in both cases, which lowers the confidence in generating code reviews with the help of GPT. In this sub-subsection, we discuss the false predictions in the fixed set of both prompts.
GPT seems to handle the vulnerability type CWE-20 very carefully. It often generates the word "Input validation" using the second prompt.
Table A.5b shows the most frequent words in the code reviews of the fixed code snippets. We see that "Input" is the second most frequent word in all code reviews, not only in CWE-20 cases. To put it differently, GPT finds in many code snippets other vulner-

abilities, and "Input Validation" is the most occurring vulnerability in the 50 generated code reviews; it appears 11 times as "other vulnerabilities" in other code reviews, which are not explicitly vulnerable to CWE-20.
Out of 8 false positives in the first code generation, we have two false positives from type CWE-20 A.1a, which is surprisingly one-quarter. Considering the second prompt, we see that the vulnerability type CWE-20 is five times falsely predicted out of 7 times A.2b, where the code snippet is actually secure from this vulnerability. We considered the column "other vulnerabilities" in `full_results.csv` in our Git Repository [21] and calculated how many code reviews of secure code snippets contained this vulnerability type (CWE-20 "Input Validation") and obtained 32 code reviews out of 50 (inclusive CWE-20 cases), which shows the high attention to this type.
The reason is as discussed in section 5.1, the addition of "security aspects" to the prompt increases the number of positive predictions even if the secure cases, and due to the strong attention to the "Input validation" vulnerability type, the number of false positives increased in a huge way.
Finally, we declare that mentioning this vulnerability type in multiple instances, even in fixed cases, highlights GPT's capability. To be more precise, this vulnerability is present in many code reviews and the reason is simply because of the missed context and background about the code. For example (38, CWE-835, 2), for GPT it is a no-go to use an input without any if-statement or a Boolean validation function.
GPT: `No input validation for the 'name' variable: The 'name' variable is directly used in the code without any validation.`
The input parameter is not validated in a code snippet, but it might be validated before accessing the function. That means that GPT can not successfully review the code snippet alone if no further context is provided about the input parameters and the used functions in the code.

We saw that vulnerabilities from type CWE-20 were found in the code reviews (4: prompt 1, 5: prompt 2). This type occurred not only in these code reviews but also as "other vulnerability" in other cases (11: prompt 1, 32: prompt 2), which shows that "Input Validation" is the first way for GPT to escape the missed context about the code. If developers depend on GPT to generate code reviews and consistently receive "Input Validation" as the output, they might begin to doubt the system's reliability. This constant occurrence wastes generated words and lowers trust since the actual vulnerability type could exist but goes unnoticed due to its repetitive appearance. Hence, it is more desirable to receive warnings for this vulnerability only when it truly exists rather than encountering them in every instance.
Considering the fixed code review of (18, CWE-20, 1). The code snippet defines a static method named "isValid" which validates whether the input string contains any characters that are considered unsafe or illegal in certain contexts. The list of unsafe or illegal characters defined in this method includes characters like '/', '¿ '?', ':', '<', '>', ';', '(', ')', '@', ',', '[', ']', '=', and '\n'. GPT answered with the following statement:

`there is no check to see if the configuration parameter contains any malicious or unexpected input.`

Assuming the code is ours, it might be secure for us because we implemented our black-

list of special characters to reject only these characters, but the scenario might change in GPT's view. GPT does not have context about the background and thinks that other special characters might belong to this blacklist, telling:

```
"this method has a limited scope, and it only checks for a predefined
set of unsafe characters. It should be used in conjunction with other
security measures to ensure complete protection against security
threats."
```

and leading to suggestions that the code is vulnerable to input validation.
Another example of false positives is (42, CWE-862, 2):

```
The code only checks for the CONFIGURE permission. Depending on how
this code is being used, it may be necessary to check for additional
permissions such as BUILD, DELETE, or READ..
```

We see that GPT is not suggesting unnecessary warnings. It simply provides more information and warnings to ensure the security of the code, but these warnings cause false predictions because the code snippets are secure from them. These presented false predictions are the consequences of the model's limitation as discussed in 3.7, which indicates GPT needs more context about the code to silence these warnings.
A detailed prompt also made the GPT model predict vulnerabilities that were never generated in any code review, such as vulnerabilities from type CWE-835 (Infinite Loop). In the first code review generation, we thought the vulnerabilities from type CWE-853 could never be shown in the code reviews because we tested 6 cases and did not see any hint about this type. Still, GPT responded to the detailed security prompt and generated a warning to the code snippet presented in (11, CWE-835, 2), which suggested the code is vulnerable to a Denial of Service attack caused by an infinite loop or resource exhaustion. GPT's response was as follows:

```
3. Potential denial-of-service (DoS) vulnerabilities: The method is
responsible for invoking a channel listener to process incoming data
frames. However, there's no limit on the amount of data that can be
processed in a single iteration. In theory, an attacker could flood the
channel with a large number of data frames, causing resource exhaustion
or an infinite loop and potentially crashing the application. To
mitigate this risk, it's important to limit the amount of data that
can be processed per iteration or to have a timeout mechanism in place.
```

It is evident that a vulnerability, specifically a Denial of Service (DoS) attack, has been detected, and it is important to note that the infinite loop detection is coincidental because we know that the code does not contain this vulnerability. We could not know why this vulnerability was detected in the fixed case; it could've been better if GPT had suggested this vulnerability in the vulnerable code snippet. This makes the generation of this warning a simple "coincidence."
Sometimes, the GPT model may not explicitly indicate the presence of a critical vulnerability but rather suggest that something is unreliable. As such, `"it may be careful to perform an additional security assessment to ensure the server is not`

`compromised."`
A perfect example of this suggestion is (50, CWE-287, 2); the generated warning is:

```
It assumes that the connection to the LDAP server is trusted and only
checks the user's credentials. It is important to define access control
policies that restrict access to sensitive information and operations
based on a user's role and group membership.
```

This is a connection warning in a secure code snippet. We already have a safe connection because we already fixed it, so it is a waste of generated words, which can be used for other important warnings.

5.3.2 Code Reviews Pairwise Consideration

In this subsection, we will discuss cases, where the code review contains vulnerability hints to CWEs, which do not exist in the code. Reviewing the classes we created before 4.4, we obtain false positives in two classes, which are the Yes/Yes and the No/Yes. A comparison between the secure code and the vulnerable code is required and it is already done in the `code difference` column in file `results_full.csv` in the GitLab repository [21]. For another comparison, we use the TF-IDF matrix between the code reviews of the vulnerable and the fixed code snippets in each case, which means the TF-IDF matrices of the false positives in the 50 cases. The discussed TF-IDF matrices in this paper are presented in the appendix A.1a and A.1b.

GPT finds the change, but it's not enough to fully exclude the vulnerability

There are several cases where the behavior of the GPT model changes in generating the code reviews. The reason was that GPT recognized the changes in the code, nevertheless, it generates warnings about the same CWE. We take a deep look into the code reviews of the vulnerable and the fixed code snippets, generated using the first prompt and focus on (3, CWE-79, 1) A.2. We added the line `write(htmlEncodeButNotSpace(remoteAddr));`, to ensure that the pointer `remonteAddr` is secure. The GPT model generates the following warning in the vulnerable form:

```
1. Cross-site scripting (XSS): The write() method does not properly
sanitize user input before it is written to the output. This could allow
an attacker to inject harmful scripts into the HTML page, leading to XSS
attacks. It is recommended to use a library that provides proper HTML
encoding to prevent this.
```

After we fixed the code and basically did exactly what GPT told us to do, the warning changed to:

```
one potential issue could be that there is no input validation being
done on the session ID parameter passed to the ‘write‘ method. If
an attacker manipulated the session ID parameter to inject malicious
scripts, it could lead to a Cross-Site Scripting (XSS) attack.
```

GPT finds that the write method is insecure, which is correct in the vulnerable version, because `remoteAddr` is a pointer and its value can be changed by an attacker if the pointer is not sanitized, but after the change, the attention changes to input validation, which might cause the XSS attack. The warning in the fixed version suggests that the variable "session" can be manipulated because it is not checked. In our case, the session parameter represents an instance of the "SessionInformations" class, which is secure. If the class is insecure, an attack can happen as GPT mentioned. Maybe if GPT has more context about this class, it might generate a better code review.
Furthermore, GPT pays more attention to the write() function, because it is repeated 33 times in the code, which increases the likelihood of an occurrence of a vulnerability from type cross-site scripting, especially when the function writes XML code. GPT is cautious and will only silence the warning if it can safely assume that there are no vulnerabilities associated with the write function.
On the other hand, the fixed version focused only on this vulnerability, which made the code review fa less security-related than the vulnerable one. The TF-IDF matrix delivered the top 20 occurring words in both of them A.1a. The most occurred security-related words are:

- In the vulnerable version: `session, user, method, code, input, attack, important, resource, injection, prevent, validated`
- In the fixed version: `session, user, displayed, method, code, input, attack`.

The TF-IDF matrix confirms this hypothesis about the changed attention in the fixed version because the word "session" has a TF-IDF value of 0.272112 in the vulnerable version and 0.641554 in the fixed version, which is the maximum of all words. The reason is the strong focus on the input validation type, which is present because of this word. On the other hand, the word "write" achieved a value of 0.152978 in the vulnerable- and 0 in the fixed version, which shows that the focus changed from "write" to "session".
We conclude that changing the code changes the attention of the GPT model from "sanitation" to "validation" and GPT suggests that the cause of Cross-site-scripting might be because of the missed input validation. We had a similar example a few sections before in (11, CWE-835, 2), where we received a false warning that a DoS attack is caused because of the infinite loop. This case is similar, where the Cross-site-scripting is caused because of the Input validation problem.
Having an additional Input validation problem is not surprising, since we discussed before, that if GPT has no context about the code, it generates immediately an input validation warning. This increases the "false positives" and leads to misclassification.
Finally, the fixed code review of (3, CWE-79, 1) focused on one vulnerability and had fewer security-related words compared with the vulnerable one. In addition, GPT does not have context about the code, which is why it generated a warning about the XXS attack.

In other cases from the Yes/Yes class 4.4, the code review generates the same warning in both vulnerable and fixed versions. In other words, the change in the code doesn't impact the behavior of GPT in generating code reviews.

GPT responds in (44, CWE-601, 2) with a perfect reply in the vulnerable version, which describes the vulnerability we are looking for, stating:

```
"A potential security vulnerability that arises from redirecting URLs
is the "open redirect" vulnerability. An attacker can exploit open
redirects to convince a user to click on a URL that takes them to a
malicious page rather than to the intended page. This snippet does
not explicitly check the validity of the redirection URL, which could
be vulnerable to open redirect attacks if the redirection URL is not
properly validated before the redirection."
```

We added the following implementation at the end of the code snippet because it is important to check the *location* variable before the *sendRedirect* call is executed, to ensure that the resulting redirect URL is properly formatted, starting with a single forward slash and a valid protocol, and to avoid any potential security issues associated with protocol-relative redirects.

```
// the added implementation
while (location.length() > 1 && location.charAt(1) == '/') {
    location.deleteCharAt(0);
}
```

Surprisingly, GPT generated the same vulnerability hint, stating:

```
Open Redirect: The code doesn't validate the redirect URI, which could
expose the application to open redirect vulnerabilities. Attackers could
use an alternate URL to redirect users to a malicious website.
```

It generated also an improvement suggestion:

```
To prevent this, it's important to ensure that the redirect URI is owned
by your application or a trusted third party.
```

In the vulnerable case, GPT suggests that there a redirection URL validation is missing, and in the fixed case it suggests that a redirect URI validation is missing. URI stands for Uniform Resource Identifier, which is a sequence of characters that identifies a web resource by location, name, or both available on the internet. On the other hand, URL stands for Uniform Resource Locator, which is a subset of URI that specifies where a resource exists and the mechanism for retrieving it [32]. We see that the attention changed from focusing on the URL to the focus on URI, but it wasn't enough to fully exclude the vulnerability.

bad suggestions and no further context

We redirect our focus to (25, CWE-287, 1) A.2. The code applies authentication to an "LDAPContext" using a simple LDAP password authentication mechanism, and GPT does not accept a simple authentication mechanism to exclude vulnerabilities, telling that it is not secure because it sends user passwords over the network in clear text. This is a good hint because the password is defined as a string input parameter and used in line `ctx.addToEnvironment(Context.SECURITY_CREDENTIALS, password);` in its form. There-

fore GPT suggests, that this might be dangerous, which might allow an attacker to read the password, suggesting `1. Plaintext passwords: The code sends the user's plaintext password over the network, which is inherently insecure. An attacker could intercept and read the password, which can be a serious security risk. It is recommended to use a more secure mechanism for transmitting passwords, such as SSL/TLS or SASL.`
`2. Authentication mechanism: Although the simple authentication mechanism is easy to implement, it is not secure because it sends user passwords over the network in clear text. If possible, stronger authentication mechanisms, such as Kerberos or NTLM, should be used.`
Without explicitly reconnecting, the LDAP context may continue to use the existing connection, which might have been established with different authentication credentials or without any authentication. Because of that, we added `ctx.reconnect(null)` after setting the authentication information using the `addToEnvironment` method to the code snippet to force the LDAP context to reconnect with the new authentication credentials and to explicitly instruct the LDAP context to close the existing connection and establish a new connection using the updated authentication credentials specified in the environment properties.
GPT's responses to `ctx.reconnect(null)`, saying that this change "is necessary if the LDAP context was previously connected without credentials, or if the connection timed out and needs to be re-established". GPT describes the purpose of this addition correctly, but the authentication warning did not disappear. GPT is still focusing on the string "password", it still suggests that "`the use of simple authentication should generally be avoided in favor of stronger authentication mechanisms such as SSL/TLS or SASL. Simple authentication sends the user's password in clear text over the network, which is vulnerable to interception and eavesdropping`".
GPT wasn't satisfied with the simple authentication mechanism, it suggested using a better one, rather than reconnecting. Therefore, it doesn't consider the added line as a solution, to silence the warning in the code review and generated a 98.8% similar code review. The TF-IDF matrix delivered also the same top 20, which are present in both code reviews A.1b.
The most occurred security-related words in both of them are:
`LDAP, password, user, mechanism, code, security, input, credential`
This confirms that both code reviews are similar to each other from the security point of view.
The reconnect option might be the best solution for us, to minimize the cost of changes by adding only 19 characters instead of changing the entire code. GPT's suggestion was to change the authentication mechanism, which costs us time and afford. Therefore we conclude that the advice in the code reviews to fix the code is not always the best solution. Furthermore, GPT does not have context about the background. It is unclear to us if GPT can generate a better code review if the input was "the code supported by its background". The reason is that the limitation of input words is a huge obstacle in our way as mentioned in 3.7.

Losing attention: There are also other cases, which belong to the class "No/Yes".

The behavior changes surprisingly in a strange way, where the code review contains vulnerabilities from targeted types in the fixed case instead of the vulnerable case, which decreases the accuracy of the model in a huge way, such as, (31, CWE-787, 2). It is important to note that we used the second prompt hoping to get vulnerabilities in the correct code snippet. Therefore, the behavior of this case is unexpected.
The GPT model generated in the fixed version 248 words more than the vulnerable version, which means that it became more detailed by adding just 46 characters to the code. The change is adding the following condition to the if condition:

```
0 <= index && ((end - start) <= (data.length - index))
```

The reason for the change is to ensure that the compression or the copy method is called only if the access is inside the array bound.
The code review of the fixed code snippet says that "`if the coder field or the "compressionFlag" field can be set by user input, then it may be possible for an attacker to exploit this method by providing inputs that cause the data to become compressed when it should not be, or vice versa.
For example, an attacker may attempt to supply a very long string that is compressed, but the method does not check for the size of the compressed data. This could potentially lead to a buffer overflow or other security vulnerability.`"
We see that GPT ignores the change we made in the first if statement. Even if the attacker can control the fields "coder" and "compressionFlag", which are simply just Boolean flags, the buffer overflow can never happen, because of the restricted bound check in the first if statement. It appears, that GPT lost the attention to the first if statement, which led it to warn us about this issue.
Some lines before, the code review suggests "`that it is unclear from this code snippet where these fields are set, or if they can be set by user input`". This brings us back to the mentioned limitation before in 3.7, that there is no further context about the code snippets.

5.4 Summery

Summarising the discussion, we conclude that the gpt-3.5-turbo model is capable of generating effective code reviews, we obtained an accuracy of 55% with only a security system massage without any security focus in the prompt.
Using a more security detailed prompt improved the accuracy by only 5%, because of the false positives in the fixed set. We found that the code review became more security related, where there were not only vulnerabilities but also improvement suggestions, and also became more similar to each other.
Furthermore, we found that without a security focus in the prompt and with an input code of more than 1000 characters, we might get incomplete or ineffective code reviews with a probability of 84.3%, which shows, that GPT can't handle big code snippets. The average word counts of the generated code reviews are similar (300-400), and the standard deviation values are very high, which suggests that the performance of the GPT model in generating efficient code reviews is independent of the number of generated words.
The model succeeded in generating warnings about CWE-20, CWE-611, CWE-287, and other vulnerability types presented in table A.2a, but on the other hand, it was not likely to find vulnerabilities about types like CWE-835, CWE-79. It is important to note, that without additional context about the code, for example, the used variables, classes, and functions in the code the GPT model might ignore the vulnerability or changes the attention to other code lines and generate warnings about them although they are secure, which leads to misclassifications and high false positives and false negatives. If we include the entire context of the code with all related functions and classes, we might exceed the maximum allowed number of tokens (gpt-3.5-turbo: 4096), which is a huge obstacle in testing the capabilities of this model.
It is also important to note, that GPT exclude vulnerabilities although they are present, which might be a reason not to rely on this model. Furthermore, it advises us in its generated code review to use another mechanism in the code, but such a solution might cost time and afford to change the entire code, so GPT might be something not very smart to generate a one-line solution.
Finally, based on the results we came up with and the misclassifications using both prompts and considering the limitations 3.7, it is not trustworthy to use the gpt-3.5-turbo model in generating code reviews to find vulnerabilities and risks.

Chapter 6

Conclusion

After discussing the obtained results, we conclude a summary of the work and answer the several research questions we asked in the first chapter.
We evaluated the performance of the gpt-3.5-turbo model. We evaluated the performance of this model in generating effective code reviews using the evaluation metrics and achieved a 55% accuracy result. Elaborating the prompt by focusing on security aspects improved the accuracy by 5%, made the code reviews more similar, and built a stronger connection to the security field, which was confirmed by the TF-IDF matrices. This demonstrates that each word in the prompt changes the behavior powerfully.
Input code size exceeding 1000 characters could significantly impact the effectiveness of code reviews generated by a prompt that lacks a security focus, with a high likelihood, but this probability decreases when using a prompt with a security focus. Furthermore, the word count of the generated code reviews does not have an impact on the efficiency, because of the similar mean results and the high standard deviation values.
The gpt-3.5-turbo model can generate effective and informative code reviews. It appears to be capable of addressing vulnerability categories, such as "Input Validation" and "Improper Authentication". However, there are also vulnerability categories that it cannot handle, like "Infinite Loop". It is crucial to highlight, that misclassifications can occur even in well-addressed vulnerability types, and the primary factor behind these false predictions is that the model acquires more context about the code, which is a limitation in our work. Firstly, because the dataset comprises only code snippets, and secondly, incorporating the complete code project may exceed the 4096 tokens limit.
The use of GPT with the waiver of human security specialists is not trustworthy because, as already seen, there are several cases where GPT fails to detect vulnerabilities, and ignoring one vulnerability can lead to a crash or a significant cyberattack. On the other hand, it is possible to use GPT in the security field only under certain conditions; that is, when we use GPT under a human security specialist's supervision, checking the code before or after GPT gives a result to ensure all vulnerabilities are detected. GPT can help in the security area to confirm the vulnerabilities. If a specialist misses a vulnerability, we still have a chance to find it if GPT looks into the code.

Appendix A

Technical Details

word	vulnerable case	fixed case
session	0.272112	0.641554
user	0.272112	0.213851
properly	0.382444	0.000000
displayed	0.000000	0.300561
method	0.108845	0.183301
code	0.217690	0.061100
input	0.217690	0.061100
attack	0.217690	0.030550
important	0.229467	0.000000
resource	0.229467	0.000000
prevent	0.229467	0.000000
html	0.163267	0.061100
column	0.000000	0.214686
ensure	0.163267	0.030550
information	0.054422	0.122201
validated	0.152978	0.000000
writing	0.152978	0.000000
output	0.152978	0.000000
injection	0.152978	0.000000
appears	0.054422	0.091651

(a) TF-IDF for case 3 using prompt 1 4.1

word	vulnerable case	fixed case
authentication	0.362241	0.321211
ldap	0.197586	0.428281
method	0.230517	0.356901
password	0.395172	0.142760
user	0.230517	0.249830
context	0.131724	0.178450
mechanism	0.230517	0.071380
code	0.131724	0.142760
dn	0.000000	0.250806
server	0.098793	0.142760
security	0.164655	0.071380
plaintext	0.231417	0.000000
simple	0.098793	0.107070
set	0.131724	0.071380
input	0.185133	0.000000
attribute	0.185133	0.000000
credential	0.032931	0.142760
provided	0.000000	0.150483
seems	0.000000	0.150483
could	0.032931	0.107070

(b) TF-IDF for case 25 using prompt 1 4.1

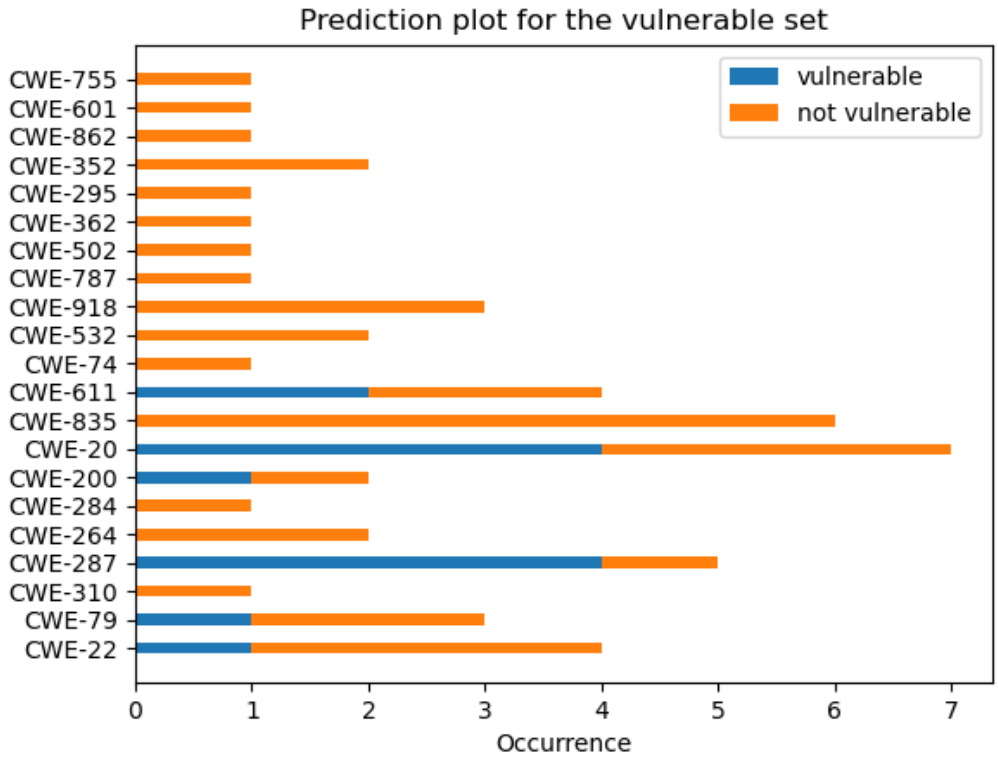
Table A.1: TF-IDF of Yes/Yes classes 4.4, which were discussed in chapter 5

Case	CWE ID	vuln	#words	#char	fixed	#words	#char	char_diff	text similarity
10	CWE-20	Yes	349	777	No	263	702	-75	0.977
18	CWE-20	Yes	244	315	Yes	363	325	10	0.953
24	CWE-20	Yes	335	2279	No	486	2321	42	0.935
30	CWE-20	Yes	211	494	No	315	495	1	0.965
40	CWE-20	No	251	1266	No	264	1266	0	0.904
41	CWE-20	No	790	1175	Yes	391	1296	121	0.838
45	CWE-20	No	449	694	No	306	703	9	0.925
9	CWE-200	Yes	336	628	No	449	647	19	0.920
34	CWE-200	No	253	937	No	276	988	51	0.941
1	CWE-22	No	177	466	No	248	523	57	0.982
5	CWE-22	No	295	651	No	294	680	29	0.981
16	CWE-22	Yes	149	86	No	259	75	-11	0.985
21	CWE-22	No	305	341	No	342	341	0	0.967
7	CWE-264	No	251	295	No	124	296	1	0.819
22	CWE-264	No	237	413	No	333	432	19	0.977
8	CWE-284	No	417	885	Yes	266	938	53	0.829
6	CWE-287	Yes	254	426	No	234	461	35	0.973
25	CWE-287	Yes	406	303	Yes	375	364	61	0.988
37	CWE-287	No	191	1283	No	460	1362	79	0.839
47	CWE-287	Yes	443	968	No	317	1062	94	0.987
50	CWE-287	Yes	503	921	Yes	274	1064	143	0.980
36	CWE-295	No	322	2525	No	333	2502	-23	0.934
4	CWE-310	No	323	3506	No	407	3512	6	0.947
39	CWE-352	No	537	1492	No	214	1628	136	0.949
46	CWE-352	No	1390	7837	No	750	7896	59	0.956
35	CWE-362	No	392	1174	No	351	1237	63	0.975
32	CWE-502	No	337	1600	No	191	1705	105	0.893
19	CWE-532	No	487	921	No	462	933	12	0.975
43	CWE-532	No	454	3377	No	665	3411	34	0.974
44	CWE-601	No	312	368	Yes	244	482	114	0.958
15	CWE-611	Yes	367	384	No	279	440	56	0.903
23	CWE-611	No	378	595	No	267	703	108	0.985
26	CWE-611	No	199	1130	No	233	1067	-63	0.967
29	CWE-611	Yes	334	1043	No	393	1241	198	0.903
17	CWE-74	No	326	3353	No	898	3348	-5	0.667
49	CWE-755	No	751	1385	No	401	1489	104	0.979
31	CWE-787	No	394	392	No	303	438	46	0.904
2	CWE-79	No	204	138	No	208	149	11	0.889
3	CWE-79	Yes	217	1888	Yes	488	1911	23	0.926
20	CWE-79	No	241	1419	No	579	1469	50	0.897
11	CWE-835	No	382	969	No	355	989	20	0.979
12	CWE-835	No	291	669	No	323	670	1	0.825
13	CWE-835	No	394	149	No	247	146	-3	0.911
14	CWE-835	No	181	500	No	408	536	36	0.888
28	CWE-835	No	488	710	No	635	755	45	0.914
38	CWE-835	No	328	1415	No	216	1466	51	0.981
42	CWE-862	No	391	250	Yes	555	342	92	0.920
27	CWE-918	No	275	2109	No	453	2169	60	0.975
33	CWE-918	No	478	920	No	195	853	-67	0.976
48	CWE-918	No	353	863	No	204	945	82	0.957
Max	-	-	1390	7837	-	898	7896	198 / -75	0.988
Min	-	-	149	86	-	124	75	0	0.667
Mean	-	-	367	1174	-	357.92	1215	54 / -35	0.931
Std	-	-	194	1251	-	150.35	1253	45 / 29	0.060

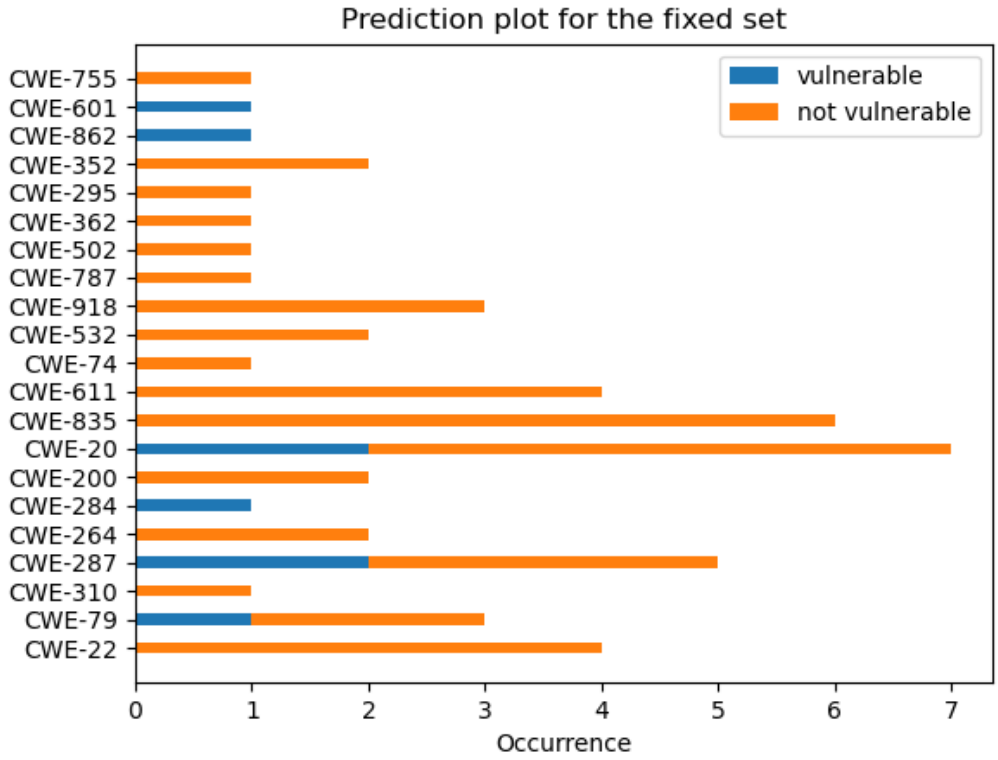
Table A.2: Word count with the text and sentence similarities for **Prompt 1**

Case	CWE ID	vuln	#words	#char	fixed	#words	#char	char_diff	text similarity
10	CWE-20	Yes	348	777	Yes	304	702	-75	0.970
18	CWE-20	No	254	315	No	334	325	10	0.974
24	CWE-20	Yes	403	2279	No	384	2321	42	0.991
30	CWE-20	Yes	294	494	Yes	424	495	1	0.989
40	CWE-20	Yes	376	1266	Yes	391	1266	0	0.982
41	CWE-20	Yes	371	1175	Yes	447	1296	121	0.980
45	CWE-20	No	329	694	Yes	371	703	9	0.985
9	CWE-200	No	199	628	No	417	647	19	0.963
34	CWE-200	Yes	334	937	Yes	398	988	51	0.990
1	CWE-22	No	291	466	No	214	523	57	0.951
5	CWE-22	No	308	651	No	314	680	29	0.982
16	CWE-22	Yes	417	86	No	215	75	-11	0.984
21	CWE-22	Yes	245	341	No	235	341	0	0.905
7	CWE-264	Yes	305	295	No	422	296	1	0.990
22	CWE-264	Yes	298	413	No	282	432	19	0.984
8	CWE-284	Yes	189	885	No	315	938	53	0.988
6	CWE-287	No	317	426	Yes	349	461	35	0.988
25	CWE-287	Yes	493	303	No	347	364	61	0.988
37	CWE-287	Yes	245	1283	Yes	380	1362	79	0.944
47	CWE-287	Yes	421	968	No	391	1062	94	0.984
50	CWE-287	Yes	226	921	Yes	420	1064	143	0.985
36	CWE-295	No	328	2525	No	466	2502	-23	0.963
4	CWE-310	No	329	3506	Yes	345	3512	6	0.975
39	CWE-352	Yes	450	1492	No	242	1628	136	0.977
46	CWE-352	No	311	7837	Yes	281	7896	59	0.894
35	CWE-362	No	448	1174	No	476	1237	63	0.978
32	CWE-502	No	319	1600	No	401	1705	105	0.978
19	CWE-532	No	392	921	No	595	933	12	0.965
43	CWE-532	Yes	334	3377	No	221	3411	34	0.943
44	CWE-601	Yes	355	368	Yes	418	482	114	0.986
15	CWE-611	Yes	296	384	No	303	440	56	0.990
23	CWE-611	Yes	394	595	No	238	703	108	0.972
26	CWE-611	Yes	332	1130	No	284	1067	-63	0.969
29	CWE-611	Yes	385	1043	No	332	1241	198	0.985
17	CWE-74	Yes	248	3353	No	273	3348	-5	0.984
49	CWE-755	No	344	1385	Yes	396	1489	104	0.969
31	CWE-787	No	245	392	Yes	493	438	46	0.989
2	CWE-79	No	359	138	No	267	149	11	0.981
3	CWE-79	Yes	383	1888	No	229	1911	23	0.978
20	CWE-79	No	477	1419	No	270	1469	50	0.955
11	CWE-835	No	383	969	Yes	348	989	20	0.960
12	CWE-835	Yes	237	669	No	352	670	1	0.965
13	CWE-835	No	224	149	No	325	146	-3	0.983
14	CWE-835	No	383	500	No	254	536	36	0.975
28	CWE-835	No	296	710	No	383	755	45	0.958
38	CWE-835	No	521	1415	No	302	1466	51	0.972
42	CWE-862	Yes	400	250	Yes	309	342	92	0.981
27	CWE-918	No	346	2109	No	419	2169	60	0.966
33	CWE-918	Yes	325	920	Yes	410	853	-67	0.973
48	CWE-918	No	383	863	No	388	945	82	0.985
Max	-	-	521	7837	-	595	7896	198 / -75	0.991
Min	-	-	189	86	-	214	75	0	0.8936
Mean	-	-	338	1174	-	347.48	1215.46	54 / -35	0.973
Std	-	-	74	1251	-	80.874	1252.887	45 / 29	0.0192

Table A.3: Word count with the text and sentence similarities for **Prompt 2**

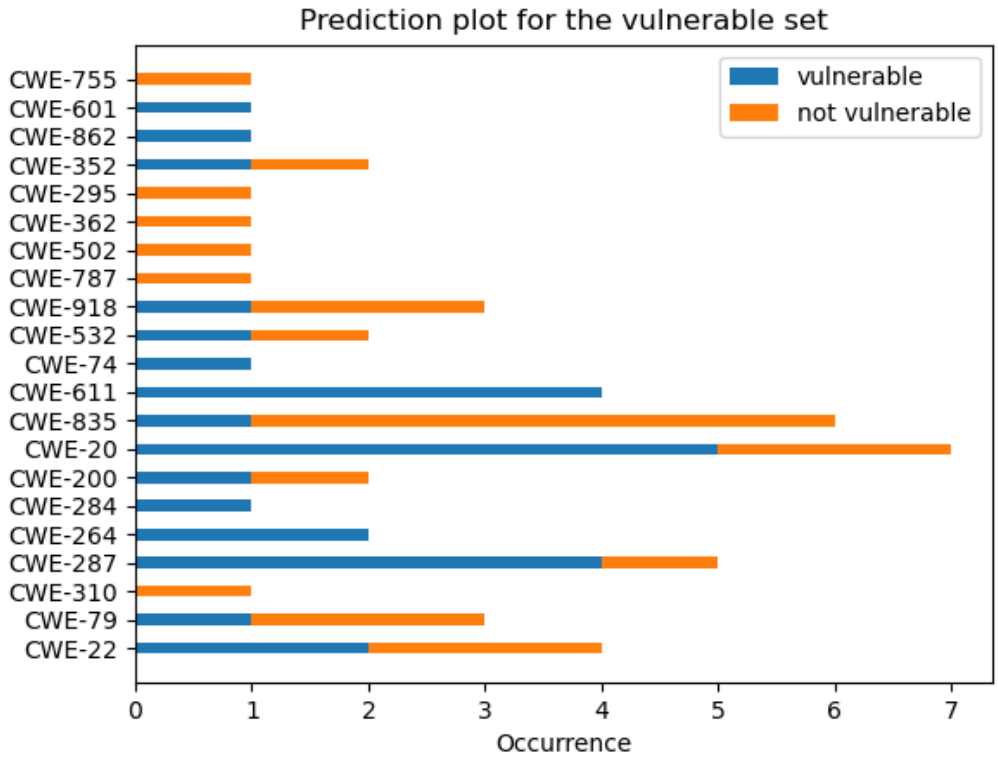


(a) Prediction of generated reviews of vulnerable code snippets

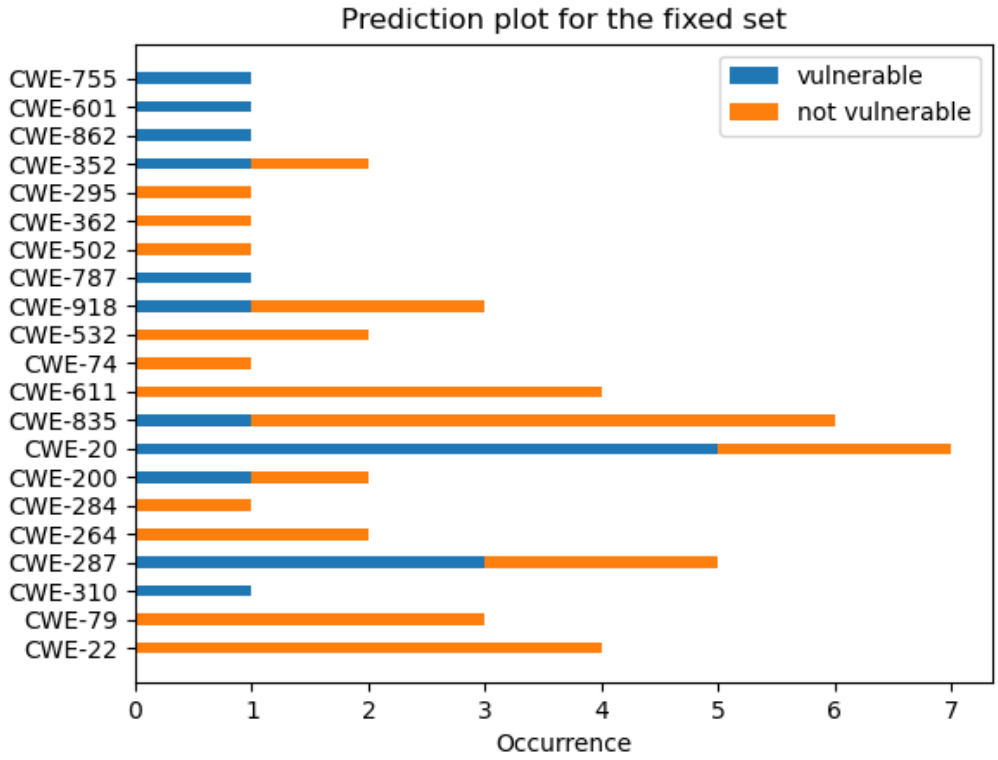


(b) Prediction of generated reviews of fixed code snippets

Figure A.1: The predictions of GPT-3.5-turbo model using the first prompt



(a) Prediction of generated reviews of vulnerable code snippets, with focus on security aspects



(b) Prediction of generated reviews of fixed code snippets, with focus on security aspects

Figure A.2: The predictions of GPT-3.5-turbo model using the second prompt

word	TF-IDF value
method	5.040051
code	4.296335
object	3.151934
request	2.343314
message	1.847012
class	1.845501
set	1.835574
input	1.830358
value	1.702276
using	1.702236
return	1.686225
check	1.674344
security	1.670935
error	1.620076
used	1.593345
file	1.584837
user	1.584354
response	1.574345
snippet	1.521218
parameter	1.519181

(a) Vulnerable set

word	TF-IDF value
method	5.721895
code	4.074083
object	2.747841
request	2.403617
parameter	2.106120
string	2.084690
user	1.862743
security	1.789900
using	1.766601
check	1.716350
session	1.641843
set	1.610485
snippet	1.590480
error	1.553644
return	1.548082
message	1.537094
instance	1.531867
value	1.521549
file	1.514666
used	1.440600

(b) Fixed set

Table A.4: TF-IDF row sum (Prompt 1)

word	TF-IDF value
code	6.794414
security	4.064951
input	4.019423
method	2.837248
snippet	2.563429
data	2.477408
user	2.434708
attack	2.368386
session	2.327738
could	2.299657
error	2.292835
xml	2.242945
access	2.201374
validation	2.024380
information	2.021940
exception	1.905668
class	1.871921
injection	1.828469
sensitive	1.815582
handling	1.760651

(a) Vulnerable set

word	TF-IDF value
code	6.297269
input	4.486592
method	3.576601
security	3.429090
user	2.551430
could	2.402749
password	2.376876
snippet	2.329156
validation	2.307746
error	2.220296
access	2.217130
data	2.162561
attack	2.093864
request	2.088172
vulnerability	1.894873
important	1.861025
class	1.826166
message	1.742038
exception	1.716743
information	1.689933

(b) Fixed set

Table A.5: TF-IDF row sum (Prompt 2)

Appendix B

Code Snippets Discussed in Chapter 5

In this section, we provide you with the code snippets we used in the discussion chapter.

Note: case 3 is too big to be added to this paper. Therefore, we direct you to the GitLab repository to view all cases in the test.txt [21].

```
---------- This is the vulnerable version  ( case: 25) ----------

protected void applyAuthentication(LdapContext ctx, String userDn, String
password) throws NamingException {
ctx.addToEnvironment(Context.SECURITY_AUTHENTICATION, SIMPLE_AUTHENTICATION);
ctx.addToEnvironment(Context.SECURITY_PRINCIPAL, userDn);
ctx.addToEnvironment(Context.SECURITY_CREDENTIALS, password);
}

---------- This is the fixed version  ( case: 25) ----------

protected void applyAuthentication(LdapContext ctx, String userDn, String
password) throws NamingException {
ctx.addToEnvironment(Context.SECURITY_AUTHENTICATION, SIMPLE_AUTHENTICATION);
ctx.addToEnvironment(Context.SECURITY_PRINCIPAL, userDn);
ctx.addToEnvironment(Context.SECURITY_CREDENTIALS, password);
// Force reconnect with user credentials
ctx.reconnect(null);
}
```

```
---------- This is the vulnerable version  ( case: 44) ----------
private void doDirectoryRedirect(HttpServletRequest request,
HttpServletResponse response)
        throws IOException {
    StringBuilder location = new StringBuilder(request.getRequestURI());
    location.append('/');
    if (request.getQueryString() != null) {
        location.append('?');
        location.append(request.getQueryString());
    }
    response.sendRedirect(response.encodeRedirectURL(location.toString()));
}
---------- This is the fixed version  ( case: 44) ----------
private void doDirectoryRedirect(HttpServletRequest request,
HttpServletResponse response)
        throws IOException {
    StringBuilder location = new StringBuilder(request.getRequestURI());
    location.append('/');
    if (request.getQueryString() != null) {
        location.append('?');
        location.append(request.getQueryString());
    }
    // Avoid protocol relative redirects
    while (location.length() > 1 && location.charAt(1) == '/') {
        location.deleteCharAt(0);
    }
    response.sendRedirect(response.encodeRedirectURL(location.toString()));
}
```

```
---------- This is the vulnerable version  ( case: 31) ----------

public void getBytes(int start, int end, byte[] data, int index) {
if (0 <= start && start <= end && end <= lengthInternal()) {
// Check if the String is compressed
if (enableCompression && (null == compressionFlag || coder ==
        LATIN1)) {
compressedArrayCopy(value, start, data, index, end - start);
} else {
compress(value, start, data, index, end - start);
}
} else {
throw new StringIndexOutOfBoundsException();
}
}
---------- This is the fixed version  ( case: 31) ----------

public void getBytes(int start, int end, byte[] data, int index) {
if (0 <= start && start <= end && end <= lengthInternal() && 0 <=
        index && ((end - start) <= (data.length - index))) {
// Check if the String is compressed
if (enableCompression && (null == compressionFlag || coder ==
        LATIN1)) {
compressedArrayCopy(value, start, data, index, end - start);
} else {
compress(value, start, data, index, end - start);
}
} else {
throw new StringIndexOutOfBoundsException();
}
}
```

References

[1] Karthik Narasimhan Alec Radford. "Improving-Language-Understanding-by-Generative-Pre-Training". 2018. URL: https://www.semanticscholar.org/paper/Improving-Language-Understanding-by-Generative-Radford-Narasimhan/cd18800a0fe0b668a1cc19f2ec95b5003d0a5035. Pre-published (cit. on p. 6).

[2] Pavel Zadorozhny Anton Cheshkov and Rodion Levichev. "Technical Report: Evaluation of ChatGPT Model for Vulnerability Detection". 2023. URL: https://arxiv.org/abs/2304.07232. Pre-published (cit. on pp. 7, 8, 14, 21).

[3] Noam Shazeer Ashish Vaswani. "Attention is all you need". 2017. URL: https://arxiv.org/pdf/1706.03762.pdf. Pre-published (cit. on pp. 4, 5).

[4] Ali Ghodsi Benyamin Ghojogh. "Attention Mechanism, Transformers, BERT, and GPT: Tutorial and Survey". 2023. URL: https://osf.io/m6gcn/. Pre-published (cit. on pp. 4, 6, 11).

[5] Som S Biswas. "Role of ChatGPT in Computer Programming." 2022. URL: https://www.semanticscholar.org/paper/Role-of-ChatGPT-in-Computer-Programming.-Biswas/f7079a97baaf8528a4d7b870549d5a969c118da2. Pre-published (cit. on p. 2).

[6] Xinyue Shen1 Zeyuan Chen2. "In ChatGPT We Trust? Measuring and Characterizing the Reliability of ChatGPT". 2023. URL: https://arxiv.org/pdf/2304.08979.pdf. Pre-published (cit. on pp. 6, 12).

[7] Rui Abreu Cláudia Mamede Eduard Pinconschi. "A transformer-based IDE plugin for vulnerability detection". 2022. URL: https://www.semanticscholar.org/paper/A-transformer-based-IDE-plugin-for-vulnerability-Mamede-Pinconschi/f4f43a23274cdf74ac017f877f7bd9cb1283b9c8. Pre-published (cit. on p. 5).

[8] Nicolas Pugeault Faisal Alamri Sinan Kalkan. "Transformer-Encoder Detector Module: Using Context to Improve Robustness to Adversarial Attacks on Object Detection". 2020. URL: https://arxiv.org/abs/2011.06978. Pre-published (cit. on p. 5).

[9] Ranjani H.G. Giriprasad Sridhara and Sourav Mazumdar. "ChatGPT: A Study on its Utility for Ubiquitous Software Engineering Tasks". 2023. URL: https://arxiv.org/pdf/2305.16837.pdf. Pre-published (cit. on pp. 1, 7, 12).

[10] Kanchan Singh; Sakshi S Grover and Ranjini Kishen Kumar. "Cyber Security Vulnerability Detection Using Natural Language Processing". 2022. URL: https://ieeexplore.ieee.org/document/9817336. Pre-published (cit. on pp. 5, 14).

[11] Benjamin Tan Hammond Pearce and Baleegh Ahmad. "Examining Zero-Shot Vulnerability Repair with Large Language Models". 2022. URL: https://arxiv.org/abs/2112.02125. Pre-published (cit. on p. 6).

[12] Weiqi Lu Haoye Tian. "Is ChatGPT the Ultimate Programming Assistant - How far is it?" 2023. URL: https://arxiv.org/pdf/2304.11938.pdf. Pre-published (cit. on p. 6).

[13] Inno Shanghai Juzheng Li. "Patch Vestiges in the Adversarial Examples Against Vision Transformer Can Be Leveraged for Adversarial Detection". 2021. URL: https://www.semanticscholar.org/paper/Patch-Vestiges-in-the-Adversarial-Examples-Against-Li-Shanghai/dfd291559716eaaff35ebd5d453a53499f4e944f. Pre-published (cit. on p. 5).

[14] K. Sudha Shruti Saravanan. "GPT-3 Powered System for Content Generation and Transformation". 2023. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9913526. Pre-published (cit. on pp. 20, 21).

[15] Benjamin Mann Tom B. Brown. "Language Models are Few-Shot Learners". 2020. URL: https://arxiv.org/abs/2005.14165. Pre-published (cit. on p. 6).

[16] Haomin Wang and Wei Li. "DDosTC: A Transformer-Based Network Attack Detection Hybrid Mechanism in SDN". 2021. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8348224/. Pre-published (cit. on p. 5).

[17] Yonghang Tai Xue Yuan Guanjun Lin and Jun Zhang. "Deep Neural Embedding for Software Vulnerability Discovery: Comparison and Optimization". 2022. URL: https://www.semanticscholar.org/paper/Deep-Neural-Embedding-for-Software-Vulnerability-Yuan-Lin/2a9bca4ac13c1fe3549a6c9efa36634a2fcc697e. Pre-published (cit. on p. 5).

[18] *ChatGPT-intro*. June 2023. URL: https://research.aimultiple.com/chatgpt-coding/ (visited on 04/12/2018) (cit. on p. 1).

[19] *Confusion matrix*. Sept. 2023. URL: https://www.wolframalpha.com/input?input=confusion+matrix (visited on 02/28/2019) (cit. on p. 14).

[20] *DiffChecker*. May 2023. URL: https://www.diffchecker.com (visited on 05/10/2023) (cit. on p. 17).

[21] *Gitlab-repo*. May 2023. URL: https://collaborating.tuhh.de/e-22/theses/bsc-yossef-al-buni/-/tree/master/Code (visited on 05/10/2023) (cit. on pp. 12, 13, 19, 25, 34, 36, 49).

[22] *GPT-4*. June 2023. URL: https://plainenglish.io/blog/beginners-guide-to-openai-s-gpt-3-5-turbo-model (visited on 04/12/2018) (cit. on pp. 2, 11).

[23] *Metrics*. July 2021. URL: https://www.analyticsvidhya.com/blog/2021/07/metrics-to-evaluate-your-classification-model-to-take-the-right-decisions/ (visited on 05/10/2023) (cit. on pp. 15, 16).

[24] *Mitre CWE library*. May 2023. URL: https://cwe.mitre.org/ (visited on 05/10/2023) (cit. on p. 10).

[25] *OpenAI-docu*. May 2023. URL: https://platform.openai.com/docs/guides/chat (visited on 05/10/2023) (cit. on pp. 1, 2, 11, 12, 20, 21).

[26] *sklearn DummyClassifier*. Aug. 2023. URL: https://scikit-learn.org/stable/modules/generated/sklearn.dummy.DummyClassifier.html (visited on 04/12/2007) (cit. on p. 7).

[27] *SpaCy*. May 2023. URL: https://spacy.io/usage/spacy-101 (visited on 05/10/2023) (cit. on p. 17).

[28] *SpaCy Text similarity calculation*. June 2023. URL: https://spacy.io/usage/vectors-similarity (visited on 04/12/2018) (cit. on pp. 17, 18).

[29] *SpaCy-vs-NLTK*. July 2021. URL: https://www.kaggle.com/general/299223# (visited on 05/10/2023) (cit. on p. 18).

[30] *Tenacity*. Apr. 2022. URL: https://tenacity.readthedocs.io/en/latest/# (visited on 04/12/2023) (cit. on p. 12).

[31] *TF-IDF*. May 2023. URL: https://arxiv.org/pdf/2002.11844.pdf (visited on 05/10/2023) (cit. on pp. 18, 19).

[32] *URL-URI*. June 2023. URL: https://www.javatpoint.com/uri-vs-url (visited on 04/12/2018) (cit. on p. 38).