information

Article

May the Source Be with You: On ChatGPT, Cybersecurity, and Secure Coding †

Tiago Espinha Gasiba [1,*], Andrei-Cristian Iosif [1,2], Ibrahim Kessba [1], Sathwik Amburi [3], Ulrike Lechner [2] and Maria Pinto-Albuquerque [4]

[1] Siemens AG, 81739 Munich, Germany; andrei-cristian.iosif@siemens.com or andrei.iosif@unibw.de (A.-C.I.); ibrahim.kessba@siemens.com (I.K.)
[2] Wirtschaftsinformatik, Institut für Schutz und Zuverlässigkeit, Universität der Bundeswehr München, 85579 Munich, Germany; ulrike.lechner@unibw.de
[3] TUM School of Computation, Information and Technology, Technische Universität München, 85748 Munich, Germany
[4] Instituto Universitário de Lisboa (ISCTE-IUL), ISTAR, 1600-189 Lisboa, Portugal; maria.albuquerque@iscte-iul.pt
* Correspondence: tiago.gasiba@siemens.com
† This article is a revised and expanded version of a paper entitled [I'm Sorry Dave, I'm Afraid I Can't Fix Your Code: On ChatGPT, CyberSecurity, and Secure Coding], which was presented at [the 4th International Computer Programming Education Conference (ICPEC 2023), Vila do Conde, Portugal, 26–28 June 2023].

Abstract: Software security is an important topic that is gaining more and more attention due to the rising number of publicly known cybersecurity incidents. Previous research has shown that one way to address software security is by means of a serious game, the CyberSecurity Challenges, which are designed to raise awareness of software developers of secure coding guidelines. This game, proven to be very successful in the industry, makes use of an artificial intelligence technique (laddering technique) to implement a chatbot for human–machine interaction. Recent advances in machine learning have led to a breakthrough, with the implementation and release of large language models, now freely available to the public. Such models are trained on a large amount of data and are capable of analyzing and interpreting not only natural language but also source code in different programming languages. With the advent of ChatGPT, and previous state-of-the-art research in secure software development, a natural question arises: to what extent can ChatGPT aid software developers in writing secure software? In this work, we draw on our experience in the industry, and also on extensive previous work to analyze and reflect on how to use ChatGPT to aid secure software development. Towards this, we conduct two experiments with large language models. Our engagements with ChatGPT and our experience in the field allow us to draw conclusions on the advantages, disadvantages, and limitations of the usage of this new technology.

Keywords: education; training; secure coding; industry; cybersecurity; capture the flag; game analysis; CyberSecurity Challenges

Citation: Espinha Gasiba, T.; Iosif, A.-C.; Kessba, I.; Amburi, S.; Lechner, U.; Pinto-Albuquerque, M. May the Source Be with You: On ChatGPT, Cybersecurity, and Secure Coding. *Information* **2024**, *15*, 572. https://doi.org/10.3390/info15090572

Academic Editor: Aneta Poniszewska-Maranda

Received: 24 July 2024
Revised: 16 August 2024
Accepted: 3 September 2024
Published: 18 September 2024

1. Introduction

According to ISO 25000 [1], one critical aspect of software development is security. Software security has gained significant attention over the last decade due to the increasing number of cybersecurity incidents resulting from poor software development practices. Consequently, industrial standards such as IEC 62443 [2] mandate the implementation of a secure software development lifecycle to address and reduce the number of vulnerabilities in products and services. The development of secure software is not only vital for the industry, particularly in critical infrastructures, but it is also a significant subject taught in many engineering and informatics courses at various universities.

There are several established methods to improve the quality of software. These include performing secure code reviews, using static application security testing (SAST),

and employing security testing techniques such as unit testing, penetration testing, and fuzzing. These methods generally rely on the principle that software should comply with secure coding guidelines, which are policies aimed at minimizing vulnerabilities and bugs in software. One way to ensure adherence to secure coding guidelines is through the use of static application security testing tools. However, not all secure coding guidelines are decidable [3]. This means that there exist certain secure coding guidelines for which no theoretical Turing machine can be constructed to determine compliance or non-compliance with the guidelines. This theoretical limitation raises an immediate issue: the full automation of secure coding is not possible. Therefore, software developers are ultimately responsible for the security of the code they write. In a 2019 survey of more than 4000 software developers from the industry, Patel [4] demonstrated that more than 50% of them cannot recognize vulnerabilities in source code.

To address this problem, raising awareness of secure coding among software developers is crucial. Similarly to Patel, Gasiba [5] has shown that industrial software developers lack awareness of secure coding guidelines. He extended the work by Hänsch et al. [6] to the field of secure coding, defining secure coding awareness in three dimensions: perception, protection, and behavior. A recent study by the Linux Foundation [7] also highlighted the importance and need to train software developers to develop secure software.

Artificial intelligence (AI) technology has garnered significant attention and traction in recent years, with integration into various fields, including cybersecurity. However, there are growing concerns about the security, privacy, and ethical implications associated with AI applications as indicated by the standardization efforts of governments in the EU and the USA [8,9]. In the industrial sector, security vulnerabilities can lead to severe consequences, ranging from financial losses to threats to human life. AI has experienced several cycles of hype and disillusionment, often referred to as 'AI winters' and 'AI summers', respectively [10]. We believe we are currently in an 'AI summer' due to the rampant integration experiments with generative AI (genAI). Furthermore, we believe that AI will play a major role in the field of secure software development as is already being demonstrated by several companies, products, and services emerging in this field.

However, previous studies that highlight the advantages, disadvantages, and expected performance of using AI for software development are scarce. Therefore, new techniques to assist software developers in writing secure code, along with a scientific evaluation, are a fruitful area of research. Gasiba et al. [11] in their work demonstrated that artificial intelligence could be used to raise awareness among software developers. The authors devised an intelligent coach using an AI technique known as the laddering technique, which is commonly used in chatbots [12]. The intelligent coach, facilitating human–machine interaction (HMi) in a controlled environment (the Sifu platform), was shown to be very effective in raising awareness of secure coding guidelines among software developers in the industry.

In this work, we extend the previous research by exploring the use of ChatGPT [13] as a means of HMi. ChatGPT, released in November 2022, is built on the GPT-3 family of large language models and was developed by the American research laboratory OpenAI. The language model has been fine-tuned with both supervised and reinforcement learning techniques.

Given the authors' experience, previous work, and the theoretical limitations inherent to the field of secure coding, this work aims to broaden the understanding of the extent to which ChatGPT can aid software developers in writing secure code. This research seeks to answer the following questions: RQ1. To what extent can ChatGPT recognize vulnerabilities in source code? RQ2. To what extent can ChatGPT rewrite code to eliminate present security vulnerabilities?

The authors have chosen ChatGPT for experimentation over other existing generative models, particularly those trained specifically for cybersecurity, because ChatGPT is widely available to the public and allows users to maintain a conversation with context, taking previous requests and answers into account. This study presented in this work is composed

of two parts: in the first part, the authors conduct interactions with ChatGPT based on five exercises from the serious game CyberSecurity Challenges (CSC) and an analysis of the responses in terms of secure coding; in the second part, the authors extend this work through additional and more extensive interactions, and using a more recent model of ChatGPT compared to the first experiment. The first experiment has been partially reported in previous work [14].

This work provides significant contributions to both academia and industry by offering a nuanced analysis of AI models in an industrial context and drawing on our extensive experience in teaching secure coding. For academia, our study highlights the practical advantages and limitations of AI, addressing often overlooked issues such as code maintainability and the prevention of undesired functionalities. These insights go beyond traditional statistical evaluations, presenting a comprehensive view that can inform future academic research into the role of AI in secure software development. For industry, our research facilitates a reflective assessment of AI integration, helping practitioners leverage tools like ChatGPT effectively while developing strategies to mitigate potential risks. By exploring the advantages, disadvantages, and limitations of human–machine interactions in raising secure coding awareness, we enrich the discourse on secure coding practices and AI applications. This work also highlights a new and rich field of research: using machine learning algorithms and Generative AI to enhance secure coding awareness through human–machine interactions, fostering a deeper understanding and improved implementation strategies in both domains.

The rest of this paper is organized as follows: Section 2 discusses previous work that is either related to or served as inspiration for our study. Section 3 briefly discusses the experiment setup followed in this work to address the research questions. In Section 4, we provide a summary of our results, and in Section 5, we conduct a critical discussion of these results. Finally, in Section 6, we conclude our work and outline future research.

2. Related Work

The use of AI in secure software development is an emerging area of research that has garnered significant attention in recent years. This section reviews the relevant standards, frameworks, and practical implementations that inform our understanding of how AI can be leveraged to enhance software security. We also examine studies and case studies that highlight both the potential and the limitations of AI tools in this context and explore the role of AI in secure software development, to illustrate the current state of the field.

The industry fosters the creation and adoption of standardization efforts because they ensure quality assurance as well as the facilitation of regulatory compliance. In the following, we will mention the standards most relevant to our work, which concern themselves with information security in the context of the software development lifecycle.

The IEC 62443 standard by the International Electrotechnical Commission provides guidelines for securing industrial automation and control systems (IACSs) [2]. It encompasses system security, risk assessment, and management, as well as secure development and lifecycle management of components. While the standard outlines processes that could benefit from AI, such as vulnerability mitigation and dynamic security measures, it is important to underscore that, at the time of writing, the standard has not yet been updated to address the fulminant advances in Generative Artificial Intelligence (GenAI). As such, AI should be considered one component of a broader, multifaceted approach to cybersecurity when considering IEC 62443 compliance.

Governmental bodies are adapting to the GenAI revolution as well. The European Union Artificial Intelligence Act focuses on regulating AI systems to align with EU values and fundamental rights, classifying AI systems into risk categories and setting requirements for high-risk categories to ensure transparency and data governance [8]. The U.S. initiative, AI.gov, serves as a central resource for federal AI activities, promoting AI innovation and public trust through coordination across various agencies, focusing on policy, research, and education [9]. Furthermore, The National Institute of Standards and Technology (NIST)

has released four draft publications intended to help improve the safety, security and trustworthiness of AI System, together with an AI Risk Management Framework [15,16]. These resources aim to guide organizations in the ethical and technical considerations of AI systems.

ISO/IEC 27001 [17] and ISO/IEC 27002 [18] are internationally recognized standards for information security management [17]. ISO/IEC 27001 provides a framework for establishing, implementing, maintaining, and continuously improving an information security management system (ISMS). ISO/IEC 27002 offers guidelines and best practices for initiating, implementing, and maintaining information security management. As such, these standards ensure that organizations can effectively manage and protect their information assets, which is essential when integrating AI into secure software development processes.

ISO/IEC 25000 [1], also known as the Software Product Quality Requirements and Evaluation (SQuARE) series, focuses on software quality. It provides a comprehensive framework for evaluating the quality of software products and includes standards such as ISO/IEC 25010, which defines quality models for software and systems. These models include characteristics like security, reliability, and maintainability, which are essential for assessing the quality of code, regardless of whether they are AI generated or not. By adhering to the ISO/IEC 25000 standards, developers can ensure that AI tools contribute positively to software quality and security.

At the technical level of secure software development, several frameworks and standards guide developers toward safer software development practices. For example, the MITRE Corporation's Common Weakness Enumeration (CWE) releases a secure coding standard containing more than 1200 secure coding guidelines for different programming languages. MITRE also releases the CWE Top 25, which highlights twenty-five guidelines out of the entire catalog which are considered very important in addressing security in software, i.e., it ranks the top 25 "most dangerous software weaknesses" that could lead to serious vulnerabilities if left unaddressed. This list helps developers prioritize their security efforts based on the potential risks and impacts of these weaknesses across different types of software development projects.

Similarly, the OWASP (Open Worldwide Application Security Project) offers various Top 10 lists, updated periodically, across different domains. With the rise in popularity of large language models (LLMs), OWASP has expanded its scope by releasing a Top 10 specifically for LLMs. This resource aims to educate developers, designers, architects, managers, and organizations on the potential security risks associated with deploying and managing LLMs [19]. MITRE, through the ATLAS Matrix project [20], has also adapted its ATT&CK matrix, a resource on attacker tactics and techniques that covers machine learning (ML) techniques.

Furthermore, the ISO/IEC TR 24772-1 [21] provides a taxonomy of software vulnerabilities, offering guidance on avoiding common mistakes in a variety of programming languages. This technical report is part of a series that aids developers in understanding how to implement secure coding practices effectively. It covers vulnerabilities related to language-specific issues and provides mitigation strategies that are essential for developing robust, secure applications.

Driven by innovations in machine learning and deep learning, GenAI models are designed to generate content, such as text and images. Notable examples include generative adversarial networks (GANs) and transformer-based models like GPT (Generative Pre-trained Transformer) [22]. The latter has demonstrated capabilities in natural language understanding and generation, facilitating tasks ranging from automated content creation to complex problem-solving. Next, we will mention the most notable models available today.

ChatGPT, part of the GPT series by OpenAI, has gained widespread attention for its conversational abilities and versatility in handling diverse queries. Its application ranges from customer service to educational tools and beyond [23]. In the context of software development, ChatGPT's ability to understand and generate human-like text provides a unique opportunity to assist developers. Another model that is gaining traction both in

industry and also academia is *LLaMA* (Large Language Model-based Automated Assistant), which its parent company, Meta, has made openly available [24]. However, since these models are probabilistic in nature, their usage and performance in the field of secure software development still lacks understanding. In contrast, our work contributes to the understanding of using AI as a means to assist software developers in writing secure code.

Russel et al. [25] have explored the potential of AI to revolutionize industries by enhancing efficiency, accuracy, and innovation. However, they also highlighted critical challenges such as ethical considerations, security risks, and the need for robust validation mechanisms. Shen et al. [26] also showed that, in software development, AI tools show promise in automating repetitive tasks, improving code quality, and identifying vulnerabilities. Yet, these studies also caution about the over-reliance on AI without proper oversight, which can lead to unintended consequences, including the introduction of new security flaws.

Fu et al. [27] introduced LineVul, a Transformer-based method for line-level vulnerability prediction, improving upon the IVDetect approach of Li et al. [28]. In their study with over 188,000 C/C++ functions, LineVul significantly outperformed existing methods, notably achieving up to 379% better F1-measure for function-level predictions and reducing effort by up to 53% for achieving 20% recall. Notably, concerning the CWE Top-25, LineVul is very accurate (75–100%) for predicting vulnerable functions. This highlights AI tools' potential to become more efficient and effective in vulnerability detection in real-world applications, similar to how SAST tools are integrated today in the development lifecycle.

GitHub CoPilot, developed in collaboration with OpenAI, is a developer productivity tool. It utilizes AI to provide real-time code suggestions and auto-completions within integrated development environments (IDEs). CoPilot can predict and generate code snippets, thereby accelerating the coding process [29]. However, concerns have been raised about its potential to inadvertently propagate insecure coding practices and vulnerabilities, necessitating a closer examination of its impact on secure software development. Not only that, concerns have also been raised about privacy and intellectual property, with GitHub Copilot having been found to leak secrets in the past [30].

Although increasingly explored as a tool to improve secure software development practices, Perry et al. [31] conducted a comprehensive user study on how individuals use an AI Code assistant for security tasks in various programming languages. Their findings revealed that participants using the AI, specifically OpenAI's codex-davinci-002 model, generally wrote less secure code than those who did not use the AI. Furthermore, those with AI access often overestimated the security of their code. Notably, the study found that participants who were more critical of the AI and adjusted their prompts produced code with fewer vulnerabilities. Similarly, Pearce et al. (2022) [32] systematically investigated the prevalence and conditions that can cause GitHub Copilot, a popular AI coding assistant, to recommend insecure code. The authors analyzed code generated in Python, C, and Verilog with CodeQL [33] and through manual inspection, focusing on the MITRE CWE Top-25 [34]. According to their findings, roughly 40% of the programs produced in their experiments were found to be vulnerable.

In the domain of software development, AI is being explored in the Test Driven Development (TDD) style, where AI generates code based on predefined tests, improving coding efficiency and adhering to the principles of TDD [35].

3. Experiment

In this section, we describe two experiments designed to explore the effectiveness of large language models (LLMs) in identifying and mitigating software vulnerabilities, which were carried out in two steps. The Preliminary Study with GPT-3 served as an exploratory phase, conducted during the initial release of ChatGPT, to assess the basic capabilities of the GPT-3-based version in detecting and addressing code vulnerabilities. This work was initially published in [14]. The present work builds and extends on this foundation through an additional analysis using the GPT-4 model. Our extended study allows us to

evaluate the improvements in code analysis and vulnerability mitigation achieved with the GPT-4 model.

3.1. Preliminary Study with GPT-3 Model

The Preliminary Study aimed to assess the capabilities of GPT-3 in identifying and mitigating common software vulnerabilities. To achieve this, we selected five distinct challenges from the CyberSecurity Challenges available on the Sifu platform, focusing on C/C++ vulnerabilities. These challenges were chosen based on their prevalence in real-world scenarios and their alignment with the authors' extensive experience in teaching cybersecurity. The study was conducted using the version of ChatGPT available on 13 January 2023, which is based on GPT-3. Table 1 shows a summary of the selected challenges and the corresponding CWE identifier.

Table 1. Selected challenges from Sifu platform, according to CWE ID.

ID	Vulnerability	Description
C1	CWE-121	Stack-Based Buffer Overflow
C2	CWE-758	Reliance on Undefined, Unspecified, or Implementation-Defined Behavior
C3	CWE-242	Use of Inherently Dangerous Function
C4	CWE-190	Integer Overflow or Wraparound
C5	CWE-208	Observable Timing Discrepancy

The first challenge involved a C function that contains a stack-based buffer overflow vulnerability, which was evident through the use of the *strcpy* function. The second challenge, written in C++, included a vulnerability based on undefined behavior, which could produce different results depending on the compiler. In the third challenge, the code utilized the deprecated and unsafe *gets* function, which has been removed in the C11 standard due to its risks. The fourth challenge presented an integer overflow vulnerability, which could be triggered by passing large integer values to the function. The fifth and final challenge focused on a side-channel leakage vulnerability, where the function's runtime depends on its input values, allowing for the potential leakage of sensitive information based on execution time. The last chosen challenge is more typical in embedded systems.

Listing 1 shows the source code corresponding to the fifth challenge. The issue with the code is that the function's runtime depends on its inputs, potentially leaking side-channel information. Specifically, the *for* loop will break depending on the contents of the input *a* and input *b*. If a difference is found, the loop breaks and the function returns 1. If both arrays contain the same values, the loop will take the longest time to run, dependent on the length of the arrays. The Sifu platform also provides information that inputs *a* and *b* are of the same length, equal to *len*.

Listing 1. Vulnerable Code Snippet Containing CWE-208 (C5).

```
int is_equal(const char* a, const char* b, size_t len) {
    for (size_t i = 0; i < len ; i++) {
        if (a[i] != b[i])
            return 1;
    }
    return 0;
}
```

For this example, the desired answer from the player corresponds to the code shown in Listing 2. In this listing, the function does not return immediately when the first unequal values are observed. The runtime of the function is constant, depending only on the length of the arrays. Since the comparison's returned value does not depend on the contents of the input arrays *a* and *b* but only on their length, no information is leaked through the

algorithm's execution time. Thus, an attacker manipulating one of the inputs cannot gain information about the other input based on the time the algorithm takes to run.

Listing 2. Desired Challenge Solution for C5.

```
int is_equal(const char* a, const char* b, size_t len) {
    if (a == NULL || b == NULL) return -1;

    int result = 0;
    for (size_t i = 0; i < len; i++) {
        result |= a[i] ^ b[i];
    }
    return result;
}
```

Figure 1 illustrates the process of interacting with ChatGPT to evaluate its ability to identify and correct software vulnerabilities. For each of the five selected challenges, a different code snippet S containing a specific vulnerability was provided. To assess the model's (LLM) effectiveness, we simulated multiple interactions with ChatGPT, testing its performance in recognizing the vulnerability, providing an accurate diagnosis, and proposing a suitable fix for each code snippet S.

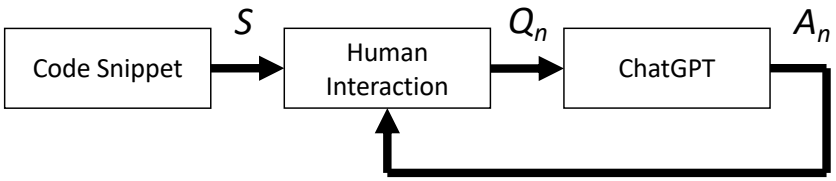

Figure 1. Interaction with ChatGPT.

Interactions with ChatGPT were conducted through multiple sessions, each involving a sequence of questions (Q_n) and corresponding answers (A_n) as shown in Figure 1. These interactions aimed to evaluate GPT-3's ability to correctly identify, explain, and rectify the presented vulnerabilities Table 2.

Table 2. Human interactions with ChatGPT (GPT-3).

Nr.	Question	Expected Answer
1	What is the vulnerability present in the following code snippet S	Correct vulnerability identification
2	What is the corresponding CWE number?	CWE number according to Table 1
3	Please fix the code	Correct fix of the code
4 . . . 15	There is still a vulnerability in the code, please fix it	Improved code
>15	The code contains vulnerability XXX, please fix it	Improved code

The strategy for asking questions was as follows: in the first question, we asked ChatGPT to identify the vulnerability by name. Given ChatGPT's verbosity, we expected it to provide a detailed description of the problem. In the second question, we asked for the corresponding CWE number to see if it matched our design. The purpose of these questions was to evaluate how well ChatGPT could support a software developer in identifying and understanding secure code problems. The questions and expected answers are detailed in Table 2.

In the next phase (starting with Question 3), we asked ChatGPT to fix the code based on its previous answers. Our expectation was that the fixed code would correctly address the challenge's vulnerability. We also posed additional questions (4 . . . 15), claiming that there were still vulnerabilities present and requesting ChatGPT to fix them. The goal was to

see how well the model could detect further issues and iteratively improve the code. Finally, in the sixteenth question, we claimed that the code contained the intended vulnerability and asked ChatGPT to fix it.

The experiments were conducted through the online interface of ChatGPT on 15 January 2023. The study involved a total of 43 interactions with the ChatGPT user interface, comprising 5 for CWE-121, 5 for CWE-758, 5 for CWE-242, 11 for CWE-190, and 17 for CWE-208. The version of ChatGPT used was "ChatGPT 9 Jan Version".

3.2. Extended Analysis with GPT-4 Model

This section extends the initial experiment by using the GPT-4 version of the LLM employed by ChatGPT. This additional work aims to further understand how LLMs perform in identifying and mitigating vulnerabilities within code snippets, building on the foundational insights gained from the GPT-3-based Preliminary Study. This study was conducted in May of 2024. This phase of the experiment was conducted in two steps. In the first step, we evaluated vulnerable code snippets by extending the Preliminary Study. In the second step, we looked at all the obtained results and analyzed them further. During the first step, we used two sets of code snippets as detailed in the following:

1. **Set 1**: *SANS Top 25 Snippets*—These snippets include vulnerabilities from the SANS Top 25, representing some of the most common and critical security flaws found in software. This set serves as a benchmark for evaluating the model's ability to recognize well-documented and frequently occurring issues.
2. **Set 2**: *Curated Code Snippet*—A single code snippet that covers a range of common vulnerabilities, aiming to test the models' generalization abilities across a broader spectrum of potential security issues. This Curated Code Snippet consists of 18 vulnerabilities and 2 secure coding malpractices.

The SANS Top 25 snippets were obtained by scrapping the MITRE website [36] for good and bad code examples for the different vulnerabilities. Each code snippet corresponds to one CWE from the SANS Top 25. A list of the used CWEs is present in Table 6. Listing 3 shows the additional Curated Code Snippet that was used in the experiment. The authors developed this code snippet based on their experience in the industry and in teaching secure coding to industrial software developers. The reasoning behind the development of this Curated Code Snippet is for it to represent several potential vulnerabilities present in industrial code all together in a single snippet, thus with the intent to stress the LLM in the identification of vulnerabilities.

Listing 3. Curated Code Snippet Issues.

```
import sqlite3, random
from flask import Flask, abort, request, jsonify
from flask_cors import CORS

app = Flask(__name__)  #!! Bad Initialization Order
CORS(app)
database = './login.db'  #!! Hardcoded database path

def create_response(message):  #!! Use type annotations
    response = jsonify({'message': message})
    response.headers.add('Access-Control-Allow-Origin', '*')
    #!! Avoid wildcard CORS
    # TODO Ticket: id91263  #!! Remove TODO comments from public code
    return response  #!! Use make_response for compliance

@app.route('/setup', methods=['POST'])
def setup():
    connection = sqlite3.connect(database)  #!! No exception handling
    SECRET_PASSWORD = "letMeIn!";  #!! Hardcoded credentials
    THIS_IS_A_VARIABLE = "WBneKJw1fHch8Qd3XFUS";  #!! Unused variable
    print("Super␣Secret␣Password␣SSH␣Server␣Password␣to␣10.10.10.1:22:␣"
          + SECRET_PASSWORD)  #!! Avoid logging secrets
```

```
    connection.executescript('CREATE␣TABLE␣IF␣NOT␣EXISTS␣login(username␣
        TEXT␣'
                             'NOT␣NULL␣UNIQUE,␣password␣TEXT␣NOT␣NULL);')
    #!! No field limits
    connection.executescript('INSERT␣OR␣IGNORE␣INTO␣login␣'
                             'VALUES("user_1","123456");')
    #!! Hardcoded credentials
    return create_response('Setup␣done!')

@app.route('/login', methods=['POST'])
def login():
    username = request.json['username']
    password = request.json['password']  #!! Password in plaintext
    connection = sqlite3.connect(database)  #!! No exception handling
    cursor = connection.cursor()
    cursor.execute('SELECT␣*␣FROM␣login␣WHERE␣username␣=␣"%s"␣'
                   'AND␣password␣=␣"%s"' % (username, password))
    #!! SQL injection risk
    user = cursor.fetchone()
    if user:
        response = create_response('Login␣successful!')
        response.set_cookie('SESSIONID',
                            str(random.randint(1,99999999999999999999)),
                            httponly=False, secure=False) #!! Insecure
                                cookies
        response.set_cookie('TESTID1', str("TESTSTRING1"),
                            httponly=True, secure=True)
        response.set_cookie('TESTID2', str("TESTSTRING2"))
        #!! Insecure defaults
        return response  #!! Use jsonify
    else:
        response = create_response('Login␣failed!')
        response.delete_cookie('username')  #!! Nonexistent cookie
        return response, 401

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=8080, debug=True)
    #!! Debug mode in production, listening on all interfaces
```

In this phase of our experiment, we used 26 code snippets as compared to 5 in the Preliminary Study. Due to the number of code snippets used in Set 1, we decided to make a small adaptation of our setup to enable the automatic querying of the OpenAI interface through the Azure service. Figure 2 details the setup that was used to evaluate the Set 1 code snippets for the SANS Top 25 vulnerabilities. Each code snippet is processed sequentially through a series of prompts, and responses are automatically recorded for evaluation.

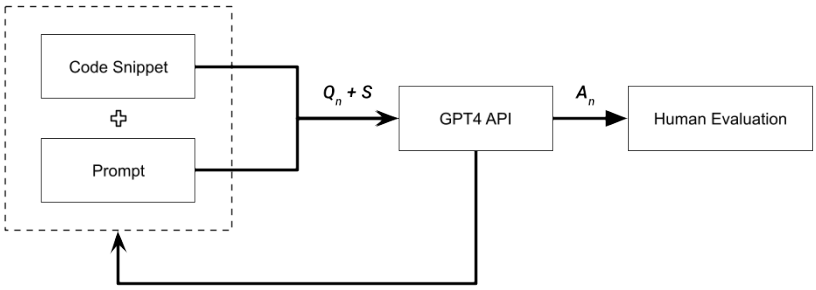

Figure 2. Experiment setup used in the extended analysis of the SANS Top 25 vulnerabilities.

While we used automation for the code snippets in Set 1 and for the Curated Code Snippet in Set 2, we decided to use the same setup as in the Preliminary study, which is detailed in Figure 1. The reason for this is that the authors wanted a fine and manual control of the experiment. Note that the code snippet used for the evaluation of the LLM did not include the comments as detailed in Listing 3, except for the `TODO Ticket :`

`id91263` comment. This was done in order not to give any hits to the LLM where the code vulnerabilities were located. However, leaving the `TODO` comment was done intentionally to understand if the LLM would identify this as a coding issue.

In this Extended Analysis, we also extended the original prompts as shown in Table 3. The reason for modifying the prompts was not only to obtain the information obtained in the Preliminary Study but also to obtain additional information about the analysis provided by the LLM. Therefore, these prompts not only mirror and extend the structure used in the Preliminary Study with GPT-3 but also allow for a direct comparison of the results.

Table 3. Prompts for the additional experiment (using GPT-4).

Prompt Nr	Prompt
P1	Please explain what this code does.
P2	Is there a vulnerability in this code? Yes or No?
P3	What is the vulnerability in this code? Please provide the CWE number if possible.
P4	Please suggest a fix for this code.
P5	Are there any other vulnerabilities in this code? If yes, please explain and suggest fixes.

The different prompts were designed based on our experience. The aim of these prompts, which were used to query the LLM, was to (P1) understand code functionality, (P2) detect vulnerabilities, (P3) obtain details on the vulnerability, (P4) suggest a fix for the vulnerable code, and (P5) comprehensive vulnerability detection. Table 4 outlines prompts together with the design goals and evaluation criteria for individual prompts.

Table 4. Evaluation based on prompts (GPT-4).

Nr.	Prompt	Evaluation Criteria and Details
P1	Please explain what this code does.	**Understanding Code Functionality:** • Correctness and depth of the models' explanations. • Clarity of explanation.
P2	Is there a vulnerability in this code? Yes or No?	**Initial Vulnerability Detection:** • Identification of presence or absence of vulnerabilities. • Consistency across different code snippets.
P3	What is the vulnerability in this code? Please provide the CWE number if possible.	**Specificity and Detail:** • Specificity and Correctness in identifying and describing vulnerabilities. • Diagnostic accuracy.
P4	Please suggest a fix for this code.	**Relevance and Feasibility of Proposed Fixes:** • Applicability and practicality of the fixes suggested. • Effectiveness of the fixes.
P5	Are there any other vulnerabilities in this code? If yes, please explain and suggest fixes.	**Comprehensive Vulnerability Detection:** • Ability to detect multiple vulnerabilities. • Depth of analysis.

While evaluating the results for each prompt, we used our experience to determine if the answers provided by the LLM fulfilled or did not fulfill the prompt criteria as defined in Table 4.

Furthermore, we conducted a second step with the aim of having a more holistic understanding of the usage of the LLMs in secure software development. This analysis was conducted based on three different aspects: correctness, completeness, and relevance. We have defined these aspects as follows:

- **Correctness**: Does the LLM correctly identify and describe vulnerabilities, focusing on the accuracy and relevance of the identified issues?

- **Completeness**: Does the LLM have the ability to identify all relevant security issues within the code?
- **Relevance**: Are the suggested fixes proposed by the LLM applicable and practical for an industrial environment?

The analysis of these aspects was conducted manually and through discussions with five additional industrial cybersecurity experts. Our results entail a report on our view of these aspects. Analysis of the *correctness* aspect was evaluated through P1, P2, and P3. The analysis of the *completeness* was based on P3 and P5. Finally, the analysis of the *relevance* aspect was based on P4 and P5. This analysis was conducted for Set 1 and Set 2 of the experiment.

4. Results

In this section, we provide the results that were obtained during the Preliminary Study and the Extended Analysis Study as detailed in Section 3.

4.1. Results for Preliminary Study with GPT-3

Table 5 summarizes the challenges (C1 to C5) and the vulnerabilities identified by ChatGPT. The model used by ChatGPT for these challenges is the GPT-3 model as discussed in the experiment section. We compared these findings with the vulnerabilities described in the challenges, noting that the CWE categories identified by ChatGPT did not always match exactly with those in the challenges. Instead, we assessed the closeness of ChatGPT's answers to the correct vulnerabilities.

For the first three challenges, we found that while the challenge's CWE described a more general vulnerability, ChatGPT identified a more specific type of the same vulnerability. This means ChatGPT's answers were precise and acceptable, as they effectively pinpointed the vulnerabilities in more detail.

Table 5. Vulnerabilities identified by ChatGPT (GPT-3).

ID	Designed Challenge Vulnerability	ChatGPT Identified Vulnerability	Assessment Is Acceptable?	Description
1	CWE-121	CWE-120	YES	Correctly identified that buffer overflow occurs due to copying string
2	CWE-758	CWE-835	YES	Correctly identified an infinite loop due to undefined behavior
3	CWE-242	CWE-121	YES	Technically correct assessment, but failed to identify that gets is deprecated
4	CWE-190	CWE-835	NO	Assessment of problem misses the major issue, in favor of smaller detail
5	CWE-208	CWE-121	*NO*	While technically correct, the assessment is wrong due to missing context

For the fourth and fifth challenges (CWE-190 and CWE-121), ChatGPT did not correctly identify the problem with the code. For the CWE-190 challenge, ChatGPT found another problem, which, although correct, was not the major problem with the code. Since the major problem with the code was not identified, we decided not to accept ChatGPT's solution. For the CWE-208 challenge, ChatGPT completely missed the problem. Even though we asked several times (4 . . . 15) to find and fix vulnerabilities in the code, ChatGPT could not identify it. In the 16th iteration, we told ChatGPT that the code had the CWE-208 vulnerability and that ChatGPT should solve the code.

Due to the interesting nature of this special case, we present the solution from ChatGPT for Question 16. Listing 4 shows the solution presented by ChatGPT. In this case, ChatGPT used the `CRYPTO_memcmp` from OpenSSL to solve the problem.

Listing 4. ChatGPT Solution to CWE-208 Challenge (using OpenSSL).

```
#include <openssl/crypto.h>

int is_equal(const char* a, const char* b, size_t len) {
```

```
    if(a == NULL || b == NULL) return -1;
    size_t a_len = strnlen(a, len);
    size_t b_len = strnlen(b, len);
    if (len == 0 || len > a_len || len > b_len) {
        len = a_len < b_len ? a_len : b_len;
    }
    if (a[a_len] != '\0' || b[b_len] != '\0') {
        return -1;
    }
    return CRYPTO_memcmp(a, b, len);
}
```

Since we were not expecting that ChatGPT would use an external library, on iteration 17, we asked ChatGPT to provide us with a solution without using OpenSSL. The answer from ChatGPT is presented in Listing 5.

Listing 5. ChatGPT Solution to CWE-208 Challenge (without using OpenSSL).

```
int is_equal(const char* a, const char* b, size_t len) {
    if(a == NULL || b == NULL) return -1;
    size_t a_len = strnlen(a, len);
    size_t b_len = strnlen(b, len);
    if (len == 0 || len > a_len || len > b_len) {
        len = a_len < b_len ? a_len : blen;
    }
    if (a[a_len] != '\0' || b[b_len] != '\0') {
        return -1;
    }
    int result = 0;
    for(size_t i = 0; i < len; i++) {
        result |= a[i] ^ b[i];
    }
    return result;
}
```

In this listing, we can see that the desired solution (see Listing 2) is present in the code. However, additional tests are performed in relation to the length of the input arrays, which is unnecessary given the context in which the function is used. Furthermore, ChatGPT has assumed that both inputs represent a string and, therefore, tests for the case that the inputs are null terminated. We note that these modifications to the source code introduced by ChatGPT change the semantics of the function.

4.2. Results for Extended Analysis with GPT-4

In this section, we cover the analysis of LLM responses for GPT-4, which were conducted in the Extended Analysis experiment. We provide the results on the code snippets of used in Set 1 and Set 2. Note that while the individual results from the SANS Top 25 (Set 1) also relate to individual CWEs, the Curated Code Snippet used in Set 2 relates to twenty individual CWEs.

Table 6 presents the results of the Set 1 code snippets based on the automated prompts. As detailed in Section 3, the results are based on the fulfillment or non-fulfillment of the individual-defined evaluation criteria. Additionally, we provide an evaluation of the coverage of each prompt toward its criteria in the last row of the table.

Our results show that the LLM fulfills the criteria of code understanding (P1) with 100%. In terms of vulnerability detection (P2), our results show a coverage of 88%. The same percentage value is also obtained for code fix suggestions (P4) and comprehensive vulnerability detection (P5). In terms of the obtainment of details on the vulnerability, our results show a coverage of 56%.

Table 6. Results for SANS Top 25 CWE (Set 1) for GPT-4 experiment.

CWE ID	Short Description	P1	P2	P3	P4	P5
CWE-787	Out-of-bounds Write	✔	✔	✔	✔	✔
CWE-79	Cross-site Scripting	✔	✔	✔	✔	✔
CWE-89	SQL Injection	✔	✔	✔	✔	✔
CWE-416	Use After Free	✔	✔	✔	✔	✔
CWE-78	OS Command Injection	✔	✔	-	✔	✔
CWE-20	Improper Input Validation	✔	✔	-	✔	✔
CWE-125	Out-of-bounds Read	✔	✔	✔	✔	✔
CWE-22	Path Traversal	✔	✔	✔	✔	✔
CWE-352	Cross-Site Request Forgery	✔	✔	✔	✔	✔
CWE-434	Unrestricted Dangerous File Upload	✔	✔	✔	✔	✔
CWE-862	Missing Authorization	✔	-	-	-	-
CWE-476	NULL Pointer Dereference	✔	✔	✔	✔	✔
CWE-287	Improper Authentication	✔	✔	✔	✔	✔
CWE-190	Integer Overflow or Wraparound	✔	✔	-	✔	✔
CWE-502	Deserialization of Untrusted Data	✔	✔	✔	✔	✔
CWE-77	Command Injection	✔	✔	-	✔	✔
CWE-119	Buffer Overflow	✔	✔	-	✔	✔
CWE-798	Use of Hard-coded Credentials	✔	✔	-	✔	✔
CWE-918	Server-Side Request Forgery	✔	✔	✔	✔	✔
CWE-306	Missing Critical Function Authentication	✔	-	-	-	-
CWE-362	Race Condition	✔	-	✔	-	✔
CWE-269	Improper Privilege Management	✔	✔	-	✔	✔
CWE-94	Code Injection	✔	✔	-	✔	✔
CWE-863	Incorrect Authorization	✔	✔	-	✔	✔
CWE-276	Incorrect Default Permissions	✔	-	-	-	-
	Coverage	100%	88%	56%	88%	88%

As described in the experiment section, we also evaluated the overall *correctness* of the results. In P1, GPT-4 consistently provided clear and accurate explanations of the code functionality across all 25 code snippets, demonstrating a robust understanding of diverse coding constructs. The model's explanations were generally precise, highlighting its capability to interpret and articulate code operations effectively. For P2, which inquired about the presence of vulnerabilities, GPT-4 accurately identified vulnerabilities in 22 out of 25 cases, reflecting an 88% success rate in initial vulnerability detection. Notably, it successfully flagged critical vulnerabilities such as CWE-787 (Out-of-bounds Write), CWE-79 (Cross-site Scripting), and CWE-89 (SQL Injection). However, the model exhibited deficiencies in accurately identifying vulnerabilities in cases like CWE-862 (Missing Authorization) and CWE-276 (Incorrect Default Permissions), suggesting areas where the model's training data might need refinement or where the inherent complexity of these vulnerabilities poses challenges (such as lack of context). For P3, which required specifying the vulnerability and providing the CWE number, GPT-4 correctly identified and described the vulnerabilities, including the correct CWE number, in 14 out of 25 cases (56%). This demonstrates a limitation in specificity and detail, as the model often struggled with assigning the correct CWE numbers, particularly for vulnerabilities such as CWE-78 (OS Command Injection) and CWE-20 (Improper Input Validation).

In terms of *completeness*, we observed that GPT-4's performance in identifying multiple vulnerabilities was variable. For 56% of the code snippets, the LLM successfully detected additional vulnerabilities beyond the primary one. For instance, in cases like CWE-125 (Out-of-bounds Read) and CWE-434 (Unrestricted Dangerous File Upload), the model

provided comprehensive analyses, indicating a thorough understanding of these vulnerabilities and their potential impacts. However, it often missed secondary vulnerabilities in more complex scenarios, such as CWE-862 (Missing Authorization), CWE-190 (Integer Overflow or Wraparound), and CWE-77 (Command Injection). The ability to detect multiple vulnerabilities is crucial for comprehensive code security analysis, and GPT-4's 56% coverage rate indicates significant room for improvement in this area.

In terms of the *relevance* aspect, our results show that GPT-4 provided relevant and practical fixes for many identified vulnerabilities. As an example, the suggested fixes for SQL injection (CWE-89), cross-site scripting (CWE-79), and use after free (CWE-416) were appropriate and effective, directly addressing the identified issues. These suggestions were typically clear and implementable, showcasing GPT-4's potential as a useful tool for developers. However, there were instances where the proposed fixes were less practical or introduced unnecessary complexity. In the case of CWE-78 (OS Command Injection) and CWE-20 (Improper Input Validation), the suggested fixes sometimes failed to fully address the issues or introduced new complexities, which could potentially lead to further vulnerabilities. Overall, the *relevance* of fixes showed coverage of 88%, indicating that while GPT-4 is generally capable of suggesting practical and effective fixes, there is still a need for improvement to ensure all proposed solutions are fully relevant and comprehensive.

Table 7 shows the results pertaining to prompts P1 through to P5 obtained for the Set 2 code snippet, i.e., the Curated Code Snippet.

Table 7. Results for Curated Code Snippet (Set 2) vs. GPT-4 experiment.

CWE ID	Short Description	P1	P2	P3	P4	P5
Multiple CWE's	Curated Code Snippet	✔	✔	-	-	-

When evaluating the GPT-4 model's response for *correctness* on the Curated Code Snippet, the results were less impressive compared to its performance on the SANS Top 25 snippets (Set 1). Out of the twenty issues present in the snippet, GPT-4 identified only eight issues. This result indicates that GPT-4 had issues identifying less common vulnerabilities, particularly when requiring a nuanced understanding of context and secure coding practices. When prompted about the identification of vulnerabilities (P3), the GPT-4 LLM correctly identified a single vulnerability, namely, the SQL injection vulnerability, which is present in the Curated Code Snippet. The GPT-4 LLM also failed to correctly identify issues that, according to the authors' experience, are important to address in the industrial context in P3, namely, hardcoded credentials and relative database file paths. The LLM model also overlooked critical problems like declaring the Flask app globally, as this issue can lead to unpredictable initialization. However, when asked if there were any additional vulnerabilities (P5), the model identified 7 additional issues out of the remaining 19. Additionally, we noted that the model generated false positives in P5 by identifying vulnerabilities that did not actually exist. This problem is also known as hallucination. In this regard, our experiment also showed that the model hallucinated random CWE numbers and descriptions when prompted with P3, only giving correct CWEs 56% of the time. Furthermore, we note that the LLM failed to identify 12 out of the 20 vulnerabilities present in the script.

The *completeness* of GPT-4's responses for the results obtained in Set 2 in identifying vulnerabilities was inconsistent. While the model managed to identify several critical vulnerabilities, the overall identification rate was lower than for Set 1. In particular, we again highlight that, out of the 20 issues present in the code snippet, GPT-4 identified only 8, thus having a negative impact on *completeness*. In particular, the GPT-4 failed to identify the risk of disabling Cross-Origin Resource Sharing (CORS) in responses, which can increase the risk of cross-site scripting (XSS) attacks and information leakage. We also observe that, although the LLM correctly identified the SQL injection vulnerability in Set 2, it missed critical issues such as declaring the Flask app globally, using hardcoded database paths. It also overlooked improper exception handling for database operations, which can lead

to application crashes or inconsistent database states. Additionally, it missed identifying the use of insecure random number generation for session IDs, which can make session hijacking attacks more likely.

In terms of the *relevance* aspect for Set 2, the results varied considerably. Our results show that the GPT-4 model correctly suggested securing cookies by setting the *httponly* and *secure* flags, mitigating the risk of session hijacking. Upon further prompting (P5 and more), the GPT-4 also identified the need to fix hardcoded credentials and suggested removing sensitive information from console logs to prevent exposure. Additionally, we observed incomplete proposals for fixes of identified vulnerabilities, e.g., the advice on remediation for SQL injection and insecure cookie handling vulnerabilities. In this case, the LLM failed to address issues such as improper initialization and the use of hardcoded database paths. The authors' opinion is that these fixes are crucial for ensuring the security and robustness of the application; however, they were not identified or suggested by the GPT-4 model.

5. Discussion

In this section, we provide a critical discussion of the results obtained in the Preliminary Study and the Extended Analysis study. The focus of our discussion is on the two guiding research questions of the present work, namely, the following: *RQ1. To what extent can ChatGPT recognize vulnerabilities in source code? RQ2. To what extent can ChatGPT rewrite code to eliminate present security vulnerabilities?*. We also draw conclusions on the practical implications of using LLMs for secure software development in an industrial context. Additionally, we provide an authors' view on the perceived advantages and disadvantages of using LLMs for secure software development and provide practical recommendations for industrial practitioners.

5.1. Critical Discussion of the Results

In terms of the capability to recognize vulnerabilities (RQ1), our Preliminary Study of GPT-3 revealed notable potential in identifying security vulnerabilities and providing remediation, especially in smaller code snippets (under 40 lines). GPT-3 successfully pinpointed underlying issues and suggested appropriate fixes in over 60% of cases despite the inherent challenges in secure coding. This result is encouraging, particularly given the non-decidable nature of many secure coding problems.

Additionally, GPT-3 demonstrated a capacity to explain its reasoning, offering accurate descriptions of vulnerabilities in about three-fifths of the analyzed cases. This suggests that LLMs like ChatGPT could be a valuable teaching tool for secure coding, helping developers understand not only what needs to be fixed but also why it needs to be fixed.

Despite these strengths, several limitations became evident in the Preliminary Study. GPT-3 often struggled to maintain the correct context of the code, leading to unnecessary corrections or changes that altered the intended semantics (RQ2). This is particularly concerning for safety-critical systems, where even subtle changes can have serious repercussions. Additionally, GPT-3 sometimes generated unnecessarily complex code compared to the original, potentially introducing performance inefficiencies and maintainability issues.

Building upon the foundation established by GPT-3, we carried out an Extended Analysis Study with a newer LLM model, namely, GPT-4. Our results show significant improvements in terms of the usability of the tool as a means to assist the development of secure software in the industry. The ability to detect vulnerabilities by GPT-4 was better than GPT-3. In particular, we observed that our Set 2 of SANS Top 25 vulnerabilities achieved an overall accuracy of 88%, compared to 60% obtained with GPT-3. While this result shows an improvement in the detection capability, it also shows that about one in ten vulnerabilities are not identified. This means that vulnerability handling in secure software development cannot be fully covered by this tool. Nevertheless, the LLM model identified common vulnerabilities such as SQL injection (CWE-89), cross-site scripting (CWE-79), and out-of-bounds write (CWE-787). We believe that this showcases the models' effectiveness in handling well-known and well-documented security issues. The authors believe that the

results highlight the usage of LLM as a valuable tool for practitioners to detect frequently encountered vulnerability issues in code.

However, GPT-4's performance on Set 2 (Curated Code Snippet) was less consistent in the identification of code vulnerabilities compared to Set 1 (SANS Top 25), resulting in a lower detection rate in Set 2 compared to Set 1. We attribute this discrepancy to the fact that the identification of code vulnerabilities in Set 2 requires additional contextual knowledge compared to Set 1. This result suggests that while GPT-4 excels with well-known vulnerabilities that have ample training data, it can struggle when faced with less common or context-dependent issues. This also highlights a potential critical challenge for the model and researchers alike: expanding and diversifying the models' training data is essential to improving its generalizability and robustness across a broader range of vulnerability types.

Moreover, GPT-4, like GPT-3, showed limitations in reliably identifying and classifying vulnerabilities in the Curated Code Snippet. We observed identification mistakes, such as misidentifying issues or generating false claims about vulnerabilities that do not actually exist, a problem known as "hallucinations". Furthermore, our results hint that the model can exhibit issues when connecting vulnerabilities to CWE numbers. We note that, due to the nature of CWEs, this result was expected. However, these issues contribute to reducing the model's reliability for practical usage. The results also highlight the need for well-trained software developers in secure coding who can cover the gaps in the LLM, i.e., to aid in the LLM gaps in accuracy and usefulness, human experts need to be involved. Integrating GPT-4 into security workflows, where people can review, interpret, and correct their findings, has the potential to significantly enhance its reliability and effectiveness.

The differences in performance between the Set 1 vulnerabilities and those of Set 2 further underscore GPT-4's limitations. While our engineered Curated Code Snippet contains 18 vulnerabilities and 2 security practice issues, the GPT-4 model only identified 8 issues. This result highlights that LLMs can produce less reliable results on software vulnerabilities that appear less in practice compared to higher reliable results for well-known and more frequent vulnerabilities found in practice. It is the authors' understanding that this result can also be tied to the context-specific nature of security flaws; however, this requires further study. Our results show that the LLM failed to reliably handle vulnerabilities such as SQL injection, particularly in its recommendations on how to fix the code. These kinds of omissions are likely dependent on the models' comprehensiveness of their training data. This result also highlights the importance of the continuous refinement of the training of the model.

Despite these challenges, we observed a notable improvement in the results offered by GPT-4 compared to those of GPT-3 in terms of accuracy and vulnerability identification. This improvement likely stems not only from the structured nature of the SANS Top 25 snippets but also their likely inclusion in the model's training data. According to recent research results, this effect can indeed likely be tied to the quality and diversity of its training data [37] as well as to the higher number of model parameters of GPT-4 compared to GPT-3. Nevertheless, the authors noticed reliability issues of GPT-4 in more complex and context-specific scenarios (Set 2). The authors also noticed false-positive suggestions by the LLM model. Thus, we believe that integrating GPT-4 into secure coding practices requires careful knowledge, understanding, and consideration of its limitations. In particular, we advocate for the need for human validation of its findings to avoid potential misidentifications or oversights.

Another limitation of these models is the potential reliance on outdated training data. This may prevent the model from recognizing emerging threats and vulnerabilities or provide outdated solutions for code fixes. This also highlights the essential need for continuous updates and refinement to ensure the model remains relevant in the face of evolving security challenges. Additionally, industry practice has shown that the model's capacity to learn from user interactions introduces additional risks in industrial environments. Not only could incorrect data lead to flawed recommendations but the leakage of intellectual property could also become a serious issue. We believe that implementing safeguards to pre-

vent the model from learning inaccurate information and leaking intellectual information is crucial, particularly in critical infrastructure settings.

Both GPT-3 and GPT-4 demonstrated potential in suggesting fixes for identified vulnerabilities, though with varying effectiveness (RQ2). Some proposed solutions were creative and well suited to the problems, such as securing cookies by setting the 'httponly' and 'secure' flags. However, the models also introduced overly complex fixes that could negatively impact code maintainability and performance—a critical aspect for industrial software. Inconsistencies in addressing security concerns such as hardcoded database paths or improper initialization highlight the need for further development to ensure that remediation suggestions are both accurate and comprehensive.

Our experiment shows that GPT-4 effectively suggested securing cookies and fixing hardcoded credentials; however, it failed to suggest the restriction of the network interface exposure. This lapse by the LLM can result in an increased risk of unauthorized system access.

Additionally, GPT-4 failed to identify the presence of hardcoded credentials, which not only are problematic due to their nature of being hardcoded but could also be easily guessed or brute-forced by an attacker. This result reveals additional gaps in the analysis of the vulnerabilities by the LLM.

Our view is that while both GPT-3 and GPT-4 show potential as tools for improving existing secure coding practices, their usage still has additional considerations that need attention. Our experience shows that the GPT-3 model excels in providing creative solutions and clear explanations for small code snippets and that it is a valuable tool to teach software developers about secure coding vulnerabilities. However, its lack of context and inability to provide consistently semantic-preserving fixes limit its practicality in industrial settings.

However, our results show inconsistencies with less common issues and the models' susceptibility to hallucinations. While GPT-4 demonstrates considerable potential in aiding software developers with secure coding practices, its limitations in detecting multiple vulnerabilities and assigning correct CWE numbers indicate areas where further refinement is needed by the research community. Ensuring that all suggested solutions are fully comprehensive and do not introduce new complexities also remains a challenge, underscoring the importance of ongoing improvements in AI models like GPT-4 to enhance their effectiveness in supporting secure software development.

5.2. Advantages and Disadvantages

In this section, we discuss the authors' view on the advantages and disadvantages of using LLM for secure software development as a result of our own experience and discussions with additional five industrial experts in the field and resulting from the context in which this study was conducted. The authors believe it is crucial to highlight and raise awareness about both the advantages and disadvantages of using LLMs like ChatGPT in secure software development, as these will be used more and more in the future.

The results of the discussions are summarized in two tables, describing the advantages and disadvantages, respectively. Table 8 provides details on the authors' perceived advantages of using LLMs to assist in secure software development. Our results include *automation*, *reproducability*, *lower barrier*, *rapid prototyping*, *creativity*, *ease of use*, *lowered workload*, and *increased efficiency*. Table 9 provides details based on the authors' perceived disadvantages of using LLMs to assist in secure software development. Our results include *skill*, *over-reliance*, *code quality*, *insecure code generation*, *context*, *non-decidability*, *copyright*, *privacy*, *semantics*, and *bias*.

Table 8. Advantages of LLM.

Advantage	Description
Automation	Automation in software development and cybersecurity has advanced significantly with the integration of LLMs. These models streamline vulnerability identification and mitigation, reducing manual workload and human error. By automating the detection of both subtle and complex vulnerabilities, LLMs enhance development efficiency and allow teams to concentrate on strategic tasks like designing security architectures and developing innovative solutions. This leads to a more secure software development lifecycle with fewer overlooked vulnerabilities [38–42].
Reproducible Results	An advantage of using LLMs in software development is their consistency and reproducibility. Unlike human developers, who may produce variable results due to fatigue, context switching, and subjective judgment, LLMs offer a standardized approach to vulnerability detection and code correction. This consistency is crucial for maintaining high software quality and security standards and facilitates easier audits and compliance checks. It builds trust among stakeholders by ensuring uniformity and dependability in security measures.
Lowers Barrier	The use of LLMs in software development democratizes the field, allowing less-skilled developers to contribute effectively by providing real-time suggestions and best practices [43,44]. This capability helps bridge the skill gap by accelerating the learning curve for novice developers and fosters innovation by integrating diverse perspectives into the field.
Rapid Prototyping	LLMs facilitate rapid prototyping, a crucial aspect of modern software development. They help developers quickly generate and refine code snippets, speeding up the creation and testing of functional product versions. This accelerates the development cycle, allowing teams to experiment with multiple solutions and incorporate feedback promptly. The speed and flexibility provided by LLMs are essential for staying competitive in fast-paced technology markets [45].
Encourages Creativity	LLMs serve as a sparring partner for experienced developers, boosting creativity and innovation during code reviews and problem-solving. By suggesting alternative approaches and highlighting potential improvements, LLMs encourage developers to think creatively and explore new methodologies. This interaction fosters a collaborative environment, challenges assumptions, and inspires developers to enhance their skills and adopt advanced practices, ultimately improving the quality and security of software products.
Ease of Use	The advent of ChatGPT has made it much easier to access LLMs. OpenAI has simplified the process, so these technologies are now available with minimal setup. Additionally, new tools are being created that are simple to use and can easily be integrated into popular development environments like GitHub Co-Pilot. GitHub Co-Pilot, for example, enables 55% faster task completion and improves quality in many areas [46]. From our experience and what we see in the industry, this ease of use is likely to significantly enhance the software development process, making advanced tools more accessible and valuable to developers.

Table 8. *Cont.*

Advantage	Description
Lowering Workload	We expect that the usage of LLMs in secure software development will considerably lower software developers' workload in writing code. LLMs lend themselves very well to not only generating boilerplate code but also aiding in rewriting existing code, e.g., through the introduction of additional security checks. While no prominent studies have been carried out to study the maintainability of code generated by LLM, our practical experience clearly indicates that developers can write secure code faster than without using LLM. We note that additional studies are needed to determine the impact and effect of software developers' security-related stress.
Increased Efficiency	The use of LLMs can help experienced developers write secure code more efficiently. Our experience shows that LLMs can accelerate code development and improve functionality implementation. Evaluation of GPT-4's performance on the Curated Code Snippet reveals that while it effectively identifies and fixes well-documented vulnerabilities, it struggles with less common or context-dependent issues.

Table 9. Disadvantages of LLM.

Disadvantage	Description
Lack of Skill	Inexperienced developers might generate poor software that can slip through checks. While LLMs provide valuable assistance, they cannot replace a solid understanding of secure coding principles and can be misused by inexperienced users, which may result in insecure or suboptimal code. Over-reliance on AI suggestions without grasping the security implications can result in vulnerabilities that are not immediately apparent [47–49]. Proper training is essential to use AI tools effectively as a supplement to developer expertise.
Over-reliance	Excessive dependence on AI without understanding secure coding principles can lead to incorrect results and missed vulnerabilities [48]. AI systems are not infallible and may produce erroneous outputs. Developers must critically evaluate AI suggestions to avoid accepting flawed solutions and ensure that subtle vulnerabilities are not overlooked.
Poor Code Quality	AI-generated code may suffer from issues related to maintainability, readability, and performance [48,49]. Although syntactically correct, such code can be poorly structured or complex, making it hard for developers to understand and maintain. This can lead to long-term technical debt and performance issues, especially in resource-constrained environments.
Insecure Code Generation	AI-generated code can sometimes introduce security vulnerabilities if the model generates code snippets that are insecure or non-compliant with best practices. This can occur because AI models might lack awareness of the latest security standards or fail to recognize context-specific vulnerabilities. GitHub Copilot has been reported to sometimes suggest insecure coding practices. According to a report by Snyk [47,50]. AI models can propagate and amplify security flaws if not carefully monitored. Developers must remain vigilant and continuously review AI-generated code to ensure it adheres to best security practices [47,48].
Lack of Context	AI lacks detailed context about the code, such as internal APIs and processes, leading to incomplete or inaccurate solutions. LLMs generate code based on patterns from their training data, which may not align with the specific project's architecture or standards, requiring additional human intervention to adapt and correct the code.

Table 9. *Cont.*

Disadvantage	Description
Non-decidable Problems	Some cybersecurity problems are non-decidable, meaning they cannot be solved universally by algorithms. While AI can offer heuristics, it cannot provide complete solutions for all security issues due to fundamental theoretical limitations. Developers must use AI as a tool to complement their efforts, not as a definitive answer to all security challenges.
Copyrighted Code	AI might inadvertently use or generate copyrighted code, raising legal concerns. Training datasets may include copyrighted material, so developers need to be cautious about the legal implications of AI-generated code and ensure they do not infringe on intellectual property rights.
Privacy Issues	There is a risk of information leakage when using AI tools, as highlighted by incidents like the Samsung data leak [51]. AI tools can expose sensitive information if not properly secured. Implementing stringent data protection measures and ensuring compliance with privacy regulations is crucial to safeguarding sensitive data.
Change in Semantics	AI models can exhibit biases based on their training data, leading to skewed detection and remediation suggestions. This bias can be intentional, such as focusing a model on specific topics or training it with company-internal code, or the poisoning of the model by an attacker. However, it can also arise inadvertently from unbalanced real-world data. Such biases can result in suboptimal or unfair recommendations and potentially overlook certain vulnerabilities. The careful curation of training data and continuous monitoring are necessary to mitigate these biases and ensure balanced and effective AI-assisted security practices.
Bias in Detection and Recommendations	AI-generated code may alter the behavior of the original code, introducing new issues. Changes made by AI tools can subtly affect functionality or introduce bugs. Thorough testing and review are necessary to ensure that AI-generated code preserves the intended semantics and does not create new vulnerabilities or regressions.

As a result of the discussions on the advantages and disadvantages of using LLM for secure software development, the following additional notes and observations have emerged related to reproducibility:

- Reproducibility can be difficult due to the model being provided by an external company, which prevents local running and testing and proper access to versioned releases.
- Results can vary based on stochastic variables, such as the "temperature" parameter, making it necessary to conduct extensive studies for comprehensive insights.
- The future availability of models is uncertain, as OpenAI might not make them accessible in the long term, impacting the ability to reproduce results over time.
- The evolution of LLMs over time may also affect reproducibility, as advancements and changes in models could alter results.

We would like to further highlight the fact that these discussion results present the authors' view on the subject matter, and their experience in the industrial context. Additional future research is needed to further explore and validate these findings individually.

5.3. Further Considerations and Recommendations

The increasing generation of code by AI and the subsequent training of new LLMs on this AI-generated code can lead to what is known as *model collapse* or an *AI feedback loop* [52]. In such scenarios, the performance of these models may incur degradation over time. This effect might be influenced by the following factors: (1) loss of human nuance and creativity in code, (2) propagation and amplification of errors, (3) homogenization of coding styles and solutions, and (4) decreased adaptability to new programming paradigms.

Firstly, the excessive use of AI for code generation can potentially lead to a loss of human nuance and creativity when developing new software. Secondly, training and retraining LLMs on insecure code can lead to the amplification of vulnerabilities. Third, the perceived generalization of LLM-produced code can potentially impact coding styles and solutions, which, by itself, might lead to a negative impact on code review and lead to a corresponding increase in vulnerabilities. Fourth, training LLMs on LLM-generated code rather than up-to-date programming languages and paradigms would lead to decreased adaptability. These consequences can lead to a significant impact in the usability of LLM for secure software development.

Human creativity remains essential in software development to prevent the collapse of LLM models. Therefore, it is crucial to balance leveraging AI for efficiency and ensuring human oversight and creativity in the development process.

To minimize the limitations and effects of model collapse or AI feedback loops, a multifaceted approach is necessary. First, organizations should implement robust human oversight mechanisms to ensure that AI-generated code undergoes thorough review by experienced developers. Second, diversifying AI training data with high-quality, human-written code is crucial for maintaining variety and best practices. Third, continuous validation processes, including rigorous testing and security audits, should be standard practice. Fourth, investing in developer education on AI limitations and ethical use is essential. Finally, regularly updating AI models with carefully curated datasets that incorporate the latest human-developed coding practices will help maintain the relevance and quality of AI-assisted development.

The authors believe that, by adopting these measures, the software industry can harness the benefits of AI while addressing the risks of over-reliance and quality degradation. The authors also recommend the following general actions as highlighted in Table 10.

Table 10. Recommendations.

Recommendation	Description
Quality-Gate	Ensure that the output of AI-generated code is reviewed by experienced developers. Establishing a quality-gate process, where AI-generated code undergoes thorough review by seasoned developers is crucial. These reviews can catch potential security flaws, inefficiencies, and logical errors that AI may miss. By incorporating human oversight, organizations can maintain high standards of code quality and security, ensuring that AI serves as a beneficial tool rather than a source of vulnerabilities.
Unit Tests	Use comprehensive unit tests (not developed by AI) to validate AI-generated code. Relying on unit tests is essential to verify the correctness and functionality of AI-generated code. Experienced developers should create these tests to ensure they cover a wide range of scenarios and edge cases. Comprehensive testing helps identify defects early in the development process, reducing the risk of deploying flawed software. It also provides a safety net that can detect unintended changes introduced by AI modifications.
Formal Verification	Apply formal verification methods to check the output from AI. Formal verification involves mathematically proving the correctness of algorithms with respect to a certain formal specification. By applying these rigorous methods to AI-generated code, developers can ensure that the code meets specified requirements and behaves as intended. This is particularly important for critical systems where errors can have severe consequences. Formal verification adds an extra layer of assurance beyond traditional testing techniques.
Isolated/Local LLMs	Use local LLMs to prevent internal data leakage. To mitigate the risk of sensitive information leakage, it is advisable to deploy local instances of LLMs. These local environments should be kept separate from broader networks and other systems to prevent unintended data sharing. By using local LLMs, organizations can better control the data that these models access and generate, thereby safeguarding proprietary and confidential information.

Table 10. *Cont.*

Recommendation	Description
Restricted Usage	Limit the use of AI to generating boilerplate code until more is understood about its implications. Until the broader implications of AI-generated code are fully understood, it is prudent to restrict its use to generating boilerplate and routine code segments. Boilerplate code, which often involves repetitive and standard tasks, is less likely to introduce complex vulnerabilities. This cautious approach allows developers to leverage AI's efficiency while minimizing the risks associated with its broader application.
Threat and Risk Modeling	Implement threat and risk modeling to evaluate AI-generated code. Incorporating threat and risk modeling into the development process helps identify potential security risks AI-generated code poses. This proactive approach involves assessing how AI-generated components might be exploited and determining the potential impact of such exploits. By understanding and mitigating these risks early, developers can enhance the security posture of their applications.
Developer Awareness	Raise awareness among developers about AI, including prompt engineering and understanding AI limitations. It is essential to educate developers about the strengths and limitations of AI tools. Training should cover topics such as prompt engineering, which involves crafting effective inputs for AI systems, and recognizing the boundaries of AI capabilities. By fostering a deep understanding of AI, developers can use these tools more effectively and responsibly, ensuring they augment rather than replace human expertise.
Community Engagement	Highlight the risks of using AI within the security community. Engaging with the broader security community to discuss and address the risks associated with AI is vital. By sharing knowledge and experiences, security professionals can collectively develop best practices and guidelines for the safe use of AI in software development. This collaborative effort can lead to more robust defenses against the unique challenges posed by AI technologies [19].
Regulation	Advocate for laws and standards to ensure due diligence in using AI. The rapid advancement of AI technologies necessitates the development of regulatory frameworks to ensure their responsible use. Advocacy for laws and standards can help establish clear guidelines for AI deployment, ensuring that organizations exercise due diligence in managing risks. Regulatory oversight can also promote transparency and accountability, fostering public trust in AI applications.

5.4. Threats to Validity

The authors recognize the limitations in the present study, which may impact its validity. One such limitation is the small sample size and simplified code snippets used in the experiments. This limitation underscores the need for further extensive research to fully assess ChatGPT's utility in both instructional and practical contexts. However, the authors are confident in the presented results, as they match their own experience in the field.

Another limitation is that the Set 2 evaluation utilized code snippets from the well-known SANS Top 25 CWE list. Since this is highly available information on the internet, it is reasonable to think that these code snippets might have been included in GPT-4's training data. The effect of this is the potential influence and inflation of the model's performance. To counter this effect, the authors utilized Set 2 and observed, as expected, lower performance. Therefore, vulnerability identification highly depends on whether the code snippets used in the evaluation have also been used in the training set—this is not only a threat to the current work but to any work in this field.

As LLMs are rapidly evolving, some of our results might not be reflected in later versions of ChatGPT. This could potentially limit or partially invalidate the conclusions obtained in the present study. While the authors believe that generative AI technology will experience significant advancements, potentially leading to better results than those presented here, we also contend that there are fundamental theoretical limitations that will impose constraints on its usefulness and practical applicability.

6. Conclusions

This study provides a comprehensive evaluation of ChatGPT's capabilities in identifying and mitigating software vulnerabilities, with a particular focus on its performance with the SANS Top 25 and a Curated Code Snippet. The results demonstrate that ChatGPT version GPT-4 significantly improves upon the GPT-3 earlier model, especially in its ability to detect common vulnerabilities. Specifically, ChatGPT achieved an accuracy rate of 88% when identifying vulnerabilities from the SANS Top 25 list, effectively recognizing issues such as SQL injection, cross-site scripting, and out-of-bounds write errors. This result underscores ChatGPT's potential as a valuable tool in automated vulnerability detection.

Software security is not only a critical aspect of software development but has also gained increasing attention in recent years due. This increase in attention is due to the observed growing number of cybersecurity incidents, which have significant consequences for society in general. Poor coding practices are often the root cause of these incidents, highlighting the importance of teaching and employing best practices in software development. Traditionally, these practices are taught in academic settings or through industry training programs. In this context, ChatGPT has the potential to play a dual role: assisting software developers in writing secure code and raising awareness of secure coding practices.

Addressing RQ1, the extent to which ChatGPT can recognize vulnerabilities in source code is substantial when dealing with well-known and frequently occurring issues. However, the study highlighted critical limitations, particularly when ChatGPT was confronted with random code snippets or more obscure and context-specific vulnerabilities. In the curated code snippet analysis, which included 18 vulnerabilities and 2 security practice issues, ChatGPT was only able to identify 8 issues. This indicates that while ChatGPT is effective in handling common vulnerabilities, it remains less reliable in detecting less frequent or complex security flaws and is prone to generating false positives or "hallucinations".

Regarding RQ2, ChatGPT has shown potential in rewriting code to eliminate present security vulnerabilities, but its success is largely dependent on the complexity of the vulnerabilities. For well-known vulnerabilities identified in the SANS Top 25, ChatGPT was able to provide accurate and relevant fixes, improving the security of the code. However, when dealing with less common vulnerabilities or those requiring deep contextual understanding, ChatGPT's rewrites were less effective, sometimes failing to address the root cause of the issue. Furthermore, we have identified additional issues not necessarily related to the code security but which are crucial for industrial code, such as issues with code maintainability.

The present work also investigates and reflects on the advantages and disadvantages of using generative AI for secure software development, based on the authors' opinion rooted in their experience in the field. While ChatGPT and similar models show clear potential as aids in software development, there are practical and theoretical limitations to their use. One of the major contributions of this work is an in-depth discussion on the use of ChatGPT. Based on these discussions, we provide insights relevant to both academia and industry practitioners, highlighting not only potential future research avenues but also practical advice on the use of large language models (LLMs).

In conclusion, while ChatGPT represents a significant advancement in the use of LLMs for software security, its limitations necessitate careful application and continued refinement. The model's strong performance with the SANS Top 25 vulnerabilities demonstrates its potential to assist in automated security processes, but the gaps in its detection and code rewriting capabilities, particularly with more complex or context-specific issues, indicate that ChatGPT should be used in conjunction with traditional security practices and expert oversight. Future research should explore the extent to which ChatGPT and similar models can be used to evaluate software quality and assist in code review, as well as addressing the validity of the advantages and disadvantages identified in this study.

Author Contributions: T.E.G., A.-C.I., I.K. and S.A. wrote the manuscript. U.L. and M.P.-A. reviewed the manuscript and contributed with valuable discussions on the content. T.E.G. and S.A. conducted the experiments with ChatGPT, collected the results and analyzed them. T.E.G., A.-C.I. and I.K. conducted internal discussions with colleagues that are experts in the field to enhance their understanding and validate their own reported experiences. All authors have read and agreed to the published version of the manuscript.

Funding: This research task was partially supported by Fundação para a Ciência e a Tecnologia, I.P. (FCT) [ISTAR Projects: UIDB/04466/2020 and UIDP/04466/2020]. Ulrike Lechner acknowledges funding by dtec.bw for project LIONS and dtec.bw is funded by the European Union—NextGenerationEU and for project CONTAIN by the Bundesministerium für Bildung und Forschung (FKZ 13N16581). Tiago Gasiba and Andrei-Cristian Iosif acknowledge the funding provided by the Bundesministerium für Bildung und Forschung (BMBF) for the project CONTAIN with the number 13N16585.

Informed Consent Statement: Informed consent was obtained from all cybersecurity experts involved in discussions that lead to the conclusions of our study.

Data Availability Statement: Most of the data is contained within the article. Additional data can be found in the following Github repository: https://github.com/Sathwik-Amburi/mdpi-llm-evaluation, accessed on 1 September 2024.

Acknowledgments: The authors would like to thank their work colleagues who took time from their busy work schedules to discuss and criticize the conclusions that the authors present in this work. The authors would also like to thank the anonymous reviewers who provided suggestions which led to a significant improvement of the manuscript.

Conflicts of Interest: The authors declare that this study received funding from Bundesministerium für Bildung und Forschung and Fundação para a Ciência e a Tecnologia. The funder had the following involvement with the study: Tiago Gasiba, Andrei-Cristian Iosif, Ulrike Lechner and Maria Pinto-Albuquerque.

Abbreviations

The following common abbreviations are used in this manuscript:

GenAI	Generative Artificial Intelligence
CORS	Cross-Origin Resource Sharing
HTTP	Hypertext Transfer Protocol
httponly	HTTP Only
LLM	Large Language Model
SAST	Static Application Security Testing
CWE	Common Weakness Enumeration
OWASP	Open Worldwide Application Security Project
LLMs	Large Language Models
GANs	Generative Adversarial Networks
GPT	Generative Pre-trained Transformer
HMi	Human–Machine Interaction
IACS	Industrial Automation and Control Systems
NIST	National Institute of Standards and Technology
ISMS	Information Security Management System
SQUARE	Software Product Quality Requirements and Evaluation
TDD	Test Driven Development
IDEs	Integrated Development Environments
XSS	Cross-Site Scripting
CORS	Cross-Origin Resource Sharing

References

1. *ISO/IEC 25000:2014*; Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—Guide to SQuaRE. International Organization for Standardization: Geneva, Switzerland, 2014.
2. *DIN EN/IEC 62443-4-1:2018*; Security for Industrial Automation and Control Systems—Part 4-1: Secure Product Development Lifecycle Requirements. International Electrotechnical Commission: Geneva, Switzerland, 2018.
3. Bagnara, R.; Bagnara, A.; Hill, P.M. Coding Guidelines and Undecidability. *arXiv* **2022**, arXiv:2212.13933.

4. Patel, S. 2019 Global Developer Report: DevSecOps Finds Security Roadblocks Divide Teams. Available online: https://about.gitlab.com/blog/2019/07/15/global-developer-report/ (accessed on 18 July 2020).
5. Gasiba, T.E. Raising Awareness on Secure Coding in the Industry through CyberSecurity Challenges. Ph.D. Thesis, Universität der Bundeswehr München, Neubiberg, Germany, 2021.
6. Hänsch, N.; Benenson, Z. Specifying IT Security Awareness. In Proceedings of the 25th International Workshop on Database and Expert Systems Applications, Munich, Germany, 1–5 September 2014; pp. 326–330. [CrossRef]
7. Linux Fountation. Secure Software Development Education 2024 Survey. Available online: https://www.linuxfoundation.org/research/software-security-education-study (accessed on 24 July 2024).
8. EU Artificial Intelligence Act. Available online: https://artificialintelligenceact.eu/ (accessed on 8 May 2023).
9. AI.gov: Making AI Work for the American People. Available online: https://ai.gov/ (accessed on 8 May 2023).
10. Toosi, A.; Bottino, A.G.; Saboury, B.; Siegel, E.; Rahmim, A. A Brief History of AI: How to Prevent Another Winter (A Critical Review). *PET Clin.* **2021**, *16*, 449–469. [CrossRef] [PubMed]
11. Gasiba, T.; Lechner, U.; Pinto-Albuquerque, M. Sifu—A CyberSecurity Awareness Platform with Challenge Assessment and Intelligent Coach. *Cybersecurity* **2020**, *3*, 24. [CrossRef]
12. Rietz, T.; Maedche, A. LadderBot: A Requirements Self-Elicitation System. In Proceedings of the 2019 IEEE 27th International Requirements Engineering Conference (RE), Jeju, Republic of Korea, 23–27 September 2019; pp. 357–362. [CrossRef]
13. OpenAI LP. ChatGPT. Available online: https://chat.openai.com/ (accessed on 23 January 2023).
14. Espinha Gasiba, T.; Oguzhan, K.; Kessba, I.; Lechner, U.; Pinto-Albuquerque, M. I'm Sorry Dave, I'm Afraid I Can't Fix Your Code: On ChatGPT, CyberSecurity, and Secure Coding. In Proceedings of the 4th International Computer Programming Education Conference (ICPEC 2023), Vila do Conde, Portugal, 26–28 June 2023; Peixoto de Queirós, R.A., Teixeira Pinto, M.P., Eds.; Dagstuhl: Wadern, Germany, 2023; Volume 112, pp. 2:1–2:12. [CrossRef]
15. Artificial Intelligence at NIST. Available online: https://www.nist.gov/artificial-intelligence (accessed on 8 May 2024).
16. AI Risk Management Framework. Available online: https://www.nist.gov/itl/ai-risk-management-framework (accessed on 8 May 2024).
17. *ISO/IEC 27001:2013*; Information Technology—Security Techniques—Information Security Management Systems—Requirements. International Organization for Standardization: Geneva, Switzerland, 2013.
18. *ISO/IEC 27002:2022*; Information Security, Cybersecurity and Privacy Protection—Information Security Controls. International Organization for Standardization: Geneva, Switzerland, 2022.
19. OWASP Foundation. OWASP Top 10 for LLMs. 2021. Available online: https://owasp.org/www-project-top-10-for-large-language-model-applications (accessed on 24 May 2024).
20. MITRE Corporation. ATLAS Matrix. 2023. Available online: https://atlas.mitre.org/matrices/ATLAS (accessed on 24 July 2024).
21. *ISO/IEC 24772-1:2019*; Programming Languages—Guidance to Avoiding Vulnerabilities in Programming Languages—Part 1: Language-Independent Guidance. International Organization for Standardization: Geneva, Switzerland, 2019.
22. Radford, A.; Narasimhan, K. Improving Language Understanding by Generative Pre-Training. In Proceedings of the Improving Language Understanding by Generative Pre-Training, Pre-Print 2018, Available online: https://hayate-lab.com/wp-content/uploads/2023/05/43372bfa750340059ad87ac8e538c53b.pdf (accessed on 6 September 2024).
23. Brown, T.B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language models are few-shot learners. In Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS '20), Red Hook, NY, USA, 6–12 December 2020.
24. Meta. LLaMA: Large Language Model-Based Automated Assistant. *J. AI Res.* **2022** Available online: https://llama.meta.com (accessed on 3 August 2024).
25. Russell, S.; Norvig, P. *Artificial Intelligence: A Modern Approach, Global Edition*; Pearson Education: London, UK, 2021.
26. Shen, Z.; Chen, S. A Survey of Automatic Software Vulnerability Detection, Program Repair, and Defect Prediction Techniques. *Secur. Commun. Netw.* **2020**, *2020*, 8858010. [CrossRef]
27. Fu, M.; Tantithamthavorn, C. LineVul: A transformer-based line-level vulnerability prediction. In Proceedings of the 19th International Conference on Mining Software Repositories, Pittsburgh, PA, USA, 23–24 May 2022; pp. 608–620. [CrossRef]
28. Li, Y.; Wang, S.; Nguyen, T.N. Vulnerability detection with fine-grained interpretations. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021), New York, NY, USA, 23–28 August 2021; pp. 292–303. [CrossRef]
29. GitHub. GitHub Copilot. 2021. Available online: https://copilot.github.com/ (accessed on 3 August 2024).
30. Niu, L.; Mirza, S.; Maradni, Z.; Pöpper, C. {CodexLeaks}: Privacy leaks from code generation language models in {GitHub} copilot. In Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23), Anaheim, CA, USA, 9–11 August 2023; pp. 2133–2150.
31. Perry, N.; Srivastava, M.; Kumar, D.; Boneh, D. Do Users Write More Insecure Code with AI Assistants? In Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), Melbourne, Australia, 5–9 June 2023. [CrossRef]
32. Pearce, H.; Ahmad, B.; Tan, B.; Dolan-Gavitt, B.; Karri, R. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 22–26 May 2022; pp. 754–768. [CrossRef]

33. GitHub. CodeQL. 2024. Available online: https://codeql.github.com/ (accessed on 7 July 2024).
34. MITRE Corporation. CWE Top 25 Most Dangerous Software Weaknesses. 2023. Available online: https://cwe.mitre.org/top25/ (accessed on 24 July 2023).
35. AI TDD: You Write Tests, AI Generates Code. Available online: https://wonderwhy-er.medium.com/ai-tdd-you-write-tests-ai-generates-code-c8ad41813c0a (accessed on 8 May 2024).
36. MITRE. Common Weakness Enumeration. Available online: https://cwe.mitre.org/ (accessed on 4 February 2020).
37. Badshah, S.; Sajjad, H. Quantifying the Capabilities of LLMs across Scale and Precision. *arXiv* **2024**, arXiv:2405.03146
38. Omar, M. Detecting software vulnerabilities using Language Models. *arXiv* **2023**, arXiv:2302.11773.
39. Shestov, A.; Levichev, R.; Mussabayev, R.; Maslov, E.; Cheshkov, A.; Zadorozhny, P. Finetuning Large Language Models for Vulnerability Detection. *arXiv* **2024**, arXiv:2401.17010.
40. Jensen, R.I.T.; Tawosi, V.; Alamir, S. Software Vulnerability and Functionality Assessment using LLMs. *arXiv* **2024**, arXiv:2403.08429.
41. Li, Z.; Dutta, S.; Naik, M. LLM-Assisted Static Analysis for Detecting Security Vulnerabilities. *arXiv* **2024**, arXiv:2405.17238.
42. Tamberg, K.; Bahsi, H. Harnessing Large Language Models for Software Vulnerability Detection: A Comprehensive Benchmarking Study. *arXiv* **2024**, arXiv:2405.15614.
43. Tarassow, A. The potential of LLMs for coding with low-resource and domain-specific programming languages. *arXiv* **2023**, arXiv:2307.13018.
44. Jalil, S. The Transformative Influence of Large Language Models on Software Development. *arXiv* **2023**, arXiv:2311.16429.
45. Hou, X.; Zhao, Y.; Liu, Y.; Yang, Z.; Wang, K.; Li, L.; Luo, X.; Lo, D.; Grundy, J.; Wang, H. Large Language Models for Software Engineering: A Systematic Literature Review. *arXiv* **2024**, arXiv:2308.10620.
46. GitHub. Measuring the Impact of GitHub Copilot. 2024. Available online: https://resources.github.com/learn/pathways/copilot/essentials/measuring-the-impact-of-github-copilot/ (accessed on 16 August 2024).
47. Pearce, H.; Ahmad, B.; Tan, B.; Dolan-Gavitt, B.; Karri, R. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. *arXiv* **2021**, arXiv:2108.09293.
48. Tambon, F.; Dakhel, A.M.; Nikanjam, A.; Khomh, F.; Desmarais, M.C.; Antoniol, G. Bugs in Large Language Models Generated Code: An Empirical Study. *arXiv* **2024**, arXiv:2403.08937.
49. Fang, C.; Miao, N.; Srivastav, S.; Liu, J.; Zhang, R.; Fang, R.; Asmita; Tsang, R.; Nazari, N.; Wang, H.; et al. Large Language Models for Code Analysis: Do LLMs Really Do Their Job? *arXiv* **2024**, arXiv:2310.12357.
50. Degges, R. Copilot Amplifies Insecure Codebases by Replicating Vulnerabilities in Your Projects. *Snyk Blog* **2024** . Available online: https://snyk.io/blog/copilot-amplifies-insecure-codebases-by-replicating-vulnerabilities/ (accessed on 13 August 2024).
51. Sawers, P. Samsung Bans Use of Generative AI Tools Like ChatGPT after April Internal Data Leak. *TechCrunch* **2023** . Available online: https://techcrunch.com/2023/05/02/samsung-bans-use-of-generative-ai-tools-like-chatgpt-after-april-internal-data-leak/ (accessed on 13 August 2024).
52. Franzen, C. The AI Feedback Loop: Researchers Warn of 'Model Collapse' as AI Trains on AI-Generated Content. *VentureBeat* **2023**. Available online: https://venturebeat.com/ai/the-ai-feedback-loop-researchers-warn-of-model-collapse-as-ai-trains-on-ai-generated-content/ (accessed on 16 August 2024).