Open Access License Notice

Assessing the Effectiveness and Security Implications of AI Code Generators

Maryam Taeb
Electrical & Computer Engineering
FAMU-FSU College of Engineering
Tallahassee, USA
0000-0001-9950-1953

Hongmei Chi
Computer Information & Sciences
Florida A&M University
Tallahassee, USA
0000-0003-4610-6479

Shonda Bernadin
Electrical & Computer Engineering
FAMU-FSU College of Engineering
Tallahassee, USA
0000-0002-5527-2279

***Abstract*—Students, especially those outside the field of cybersecurity, are increasingly turning to Large Language Model (LLM)-based generative AI tools for coding assistance. These AI code generators provide valuable support to developers by generating code based on provided input and instructions. However, the quality and accuracy of the generated code can vary, depending on factors such as task complexity, the clarity of instructions, and the model's familiarity with the programming language. Additionally, these generated codes may inadvertently utilize vulnerable built-in functions, potentially leading to source code vulnerabilities and exploits. This research undertakes an in-depth analysis and comparison of code generation, code completion, and security suggestions offered by prominent AI models, including OpenAI CodeX, CodeBert, and ChatGPT. The research aims to evaluate the effectiveness and security aspects of these tools in terms of their code generation, code completion capabilities, and their ability to enhance security. This analysis serves as a valuable resource for developers, enabling them to proactively avoid introducing security vulnerabilities in their projects. By doing so, developers can significantly reduce the need for extensive revisions and resource allocation, whether in the short or long term.**

Keywords—Source Code Vulnerability, Large Language Model, LLM, NLP, Foundation Models, CodeBert, GPT

I. INTRODUCTION

Natural language processing (NLP) models that can understand the intent of a query and search through large datasets of code snippets to find relevant matches have a wide range of potential applications in both education and industry.

Code generation models can improve programming education and provide personalized learning experiences based on individual needs and learning styles [1]. They can improve the efficiency and effectiveness of software development, improve software testing and quality assurance, source code vulnerability detection, and accessibility testing [2].

Although these models have been shown to substantially enhance code-authoring performance without compromising developers' ability to perform manual code-modification tasks [18], there are also drawbacks and concerns associated with their usage. One potential disadvantage of using them for code search and discovery is that developers may become over-reliant on the models. If users rely too heavily on the models to find solutions to coding problems, they may not develop the critical thinking skills and problem-solving abilities needed to become effective programmers. Furthermore, since the accuracy of these models can vary depending on the quality of the training data, there is a risk that developers may be provided with inaccurate or incomplete information [3]. Secure coding, also known as secure programming, involves writing robust code to prevent potential security vulnerabilities. Secure coding is not only about developing high-quality code. It also requires creating a secure environment and utilizing a secure platform. However, neither state-of-the-art technologies nor the education system emphasize the significance of source code vulnerability analysis or equips developers with practical exposure and adequate tools/techniques [4].

This research, by analyzing the privacy implications of code searching and generating AI models, aims to identify potential privacy risks and take steps to mitigate them. Examining the privacy implications of code snippet-generating AI models is vital to guarantee these technologies' secure and ethical utilization. It can also assist developers in gaining a more comprehensive understanding of the potential hazards associated with their implementation, especially when deciding how to integrate models into their development processes.

The rest of this paper is organized as follows. Section II presents an overview of the related work for AI/NLP code search and discovery models and their use cases. A detailed description of the security vulnerability tests in code snippets, methodology, and performed analysis is provided in Section III. Preliminary results and discussion of the findings are presented in Section IV. Finally, Section V provides the conclusion and future works.

II. RELATED WORK

Semantic code search and code-generating AI models have been an area of active research in recent years due to their potential to improve software development productivity and quality [5]. Code snippet search involves finding code snippets from a large corpus of code repositories that are relevant to a specific programming task. Furthermore, code-generating AI models are designed to automate the process of code generation by learning from existing code [6]. These tools can be used to help programmers find code snippets that

can be reused or modified for their specific task and generate code for various programming tasks, such as bug fixing, refactoring, and code synthesis, thus reducing development time and effort. One of the most common approaches for code snippet search is to use text-based search techniques, such as keyword matching or regular expressions, to find code snippets that match a given query [7]. CodeHow, is a code snippet search model that uses natural language queries and deep learning techniques to retrieve relevant code snippets using both semantic similarity and potential APIs on code search [8]. CodeHow is deployed as the backend of Microsoft Azure service and is available as a front-end service extension on Visual Studio. CodeHow achieves a Mean Reciprocal Rank (MRR) score of 0.86%.

One of the earliest works in code generation AI models is DeepCoder, proposed by Balog et al. [9]. DeepCoder is a system that automatically generates code from a high-level specification provided as an input-output example and has achieved an accuracy of 28.2%. Code2Vec, proposed by Alon et al. [10] is a neural network-based model that learns distributed representations of code snippets and uses them to generate code by mapping each snippet to a vector representation, achieving an accuracy of 72%. A team of researchers from OpenAI proposed GPT-2, a large-scale language model that can generate natural language text [11]. While not specifically designed for code generation, GPT-2 has been shown to be capable of generating high-quality code snippets given a natural language prompt and has achieved an accuracy of 0.83%. IntelliCode Compose was proposed as a general-purpose multilingual code completion tool that is capable of predicting sequences of code tokens of arbitrary types, generating up to entire lines of syntactically correct code, achieving an edit similarity of 86.7% [12].

Given the increasing importance of vulnerability-free source code in the software development life cycle and the growing trend of using AI technology in organizations, this study explores the potential applications and limitations of code-generating AI models for educational and academic purposes. This will be achieved through an in-depth analysis of the strengths and limitations of these tools and the improvement of the generated code using prompt engineering techniques.

III. METHODOLOGY

The approach taken in this work is partially based on our previous efforts in developing a personalized learning framework for software vulnerability detection and education [4]. Following the concept of Secure Software Development Life Cycle (SSDLC), in this work, we have examined and used CodeBERT, GPT 3.5, and CodeX to replicate real-life vulnerability scenarios that were raised from unpatched source codes that included CVE and NVD vulnerability examples. We then scanned both the AI replicated code and the previously designed unpatched source codes for vulnerabilities on GPT 3.5. This allowed us to understand the effectiveness and accuracy of the code-generating AIs, thus helping developers to learn essential skills to use such tools. Fig. 1 presents a general overview of our approach. The rest of this section will present an overview of the code generation models' behavior and the vulnerabilities tested on them.

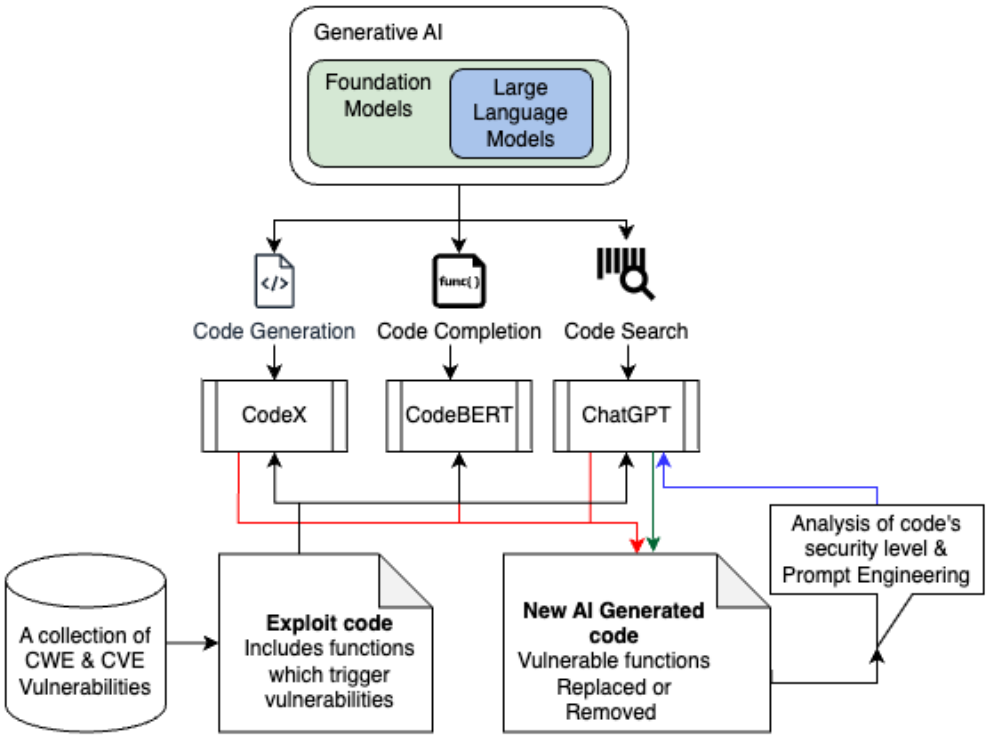

Fig. 1. Overview of the Approach

A. Experimental Setup

Our research is structured into three distinct experimental categories, each focusing on code analysis and enhancement aspects. In the first category, we leveraged the capabilities of CodeBERT, an advanced language model designed primarily for code retrieval and generation tasks. Within this category, we provided CodeBERT with code snippets exemplifying potential vulnerabilities as single-function C files, and we further enhanced the model's functionality by developing a utility function. This utility function was designed to predict masked functions or variables, enabling CodeBERT to generate recommendations for more secure built-in functions capable of mitigating potential vulnerabilities.

In the second category, we harnessed the power of CodeX, an advanced code generation tool, to create code snippets. These code snippets were intentionally designed to implement functionalities that exemplify vulnerabilities and security risks commonly encountered in software code. By utilizing CodeX's specialized capabilities, our objective was to assess the security of the generated code when incorporating the specified functionalities. The third category focused on assessing the security level of the generated code snippets. To accomplish this, we employed a combination of static analysis tools and GPT 3.5. While GPT 3.5 is not inherently designed for code-related tasks, it plays a crucial role in generating detailed and informative responses to queries about code. Additionally, GPT 3.5 offered valuable insights into code functionality and suggested ways to enhance its security. This multifaceted approach allowed us to comprehensively evaluate the security aspects of the generated code, providing a holistic perspective on code vulnerability and potential mitigation strategies.

Upon completing the experiments, we formulated a set of practical, hands-on labs to demonstrate the utilization of code-generating AI tools within the context of secure coding practices. In the subsequent section, we provide in-depth information regarding the vulnerabilities discovered and elaborate on the hands-on lab activities.

B. Tested Vulnerabilities & Hands-on Labs

In our previous work [4], we established a Personalized Learning Framework for Software Vulnerability Detection, which included a dataset comprising code snippets showcasing the most prevalent source code vulnerabilities, common CVEs, CWEs, NIST vulnerabilities, and OWASP's top 10 web application security risks. Utilizing these code snippets and their corresponding functionalities, we leveraged code-generating AI tools to introduce vulnerabilities like format string attacks, invalid string formats, and instances of undefined behavior due to unsequenced modification and variable access.

Subsequently, we integrated the outcomes of this new experiment into a fresh set of hands-on labs, maintaining continuity with the existing content. For the first category of the hands-on labs, we tasked CodeX with generating functions that would perform similar tasks to our code snippets, potentially leading to vulnerabilities. The second category of these labs employs CodeBERT to predict the names of built-in functions through the masking technique. Students are then introduced to the security implications of the predicted function and provided with alternative functions, each varying in terms of vulnerability. This approach provides them with a deeper insight into the inner workings of C's built-in functions and how they may inadvertently lead to vulnerabilities.

In the third category, we engaged GPT 3.5 to assess whether the generated code produced by CodeX, CodeBert, and the previously designed vulnerable code from the initial dataset contained any vulnerabilities. Our inquiry extended to soliciting GPT 3.5's recommendations for mitigation techniques. We then proceeded to compare these suggestions with the outcomes of static code analysis tools utilized in the initial labs, which included Clang-Tidy, FlawFinder, and VCG.

A comprehensive inventory of identified vulnerabilities, along with potential attack vectors that were tested and covered in the labs, includes the following: Format string attack (Tainted Data), Invalid String Format, Undefined Behavior Due to Unsequenced Modification and Access to Variables, Input Validation, Buffer Overflow Without User Input, Insufficient Input Sanitization, Memory Allocation (Errors & Leaks), SQL injection, Brute Force Attack, Cross-Site Scripting Attack, HTTP Flood, SYN Flood, Improper Restriction of Operations within the Buffer, NULL Pointer Deference, Use of Pointer Subtraction to Determine Size, and XSS.

IV. RESULTS & DISCUSSION

To illustrate how the analysis was carried out and what results were obtained, we have chosen two specific vulnerabilities - input validation/sensitization and format string vulnerability - that arise from the use of the "scanf" built-in function in the C programming language without setting any limits on the number of characters that the user can enter. This can result in undefined behavior and pose a potential security threat if an attacker is able to manipulate the input value. The code snippet below is an example of a vulnerable code from our initial dataset.

```
#include <stdio.h>
#include <stdlib.h>
int input;
scanf("%d", &input);
printf(input);
```

Note that the example code is lacking the following secure programming practices:

1) The printf function expects a format string as its first argument, but input is being passed as the argument. This can result in undefined behavior. To fix this, the code should use a format string to

specify the desired output format, like printf("%d", input).

2) There is no input validation or error checking on the scanf call. This can lead to undefined behavior when input is later used. To fix this, the code should ensure the return value of scanf is an integer from the input.

3) The input variable is declared as a global variable, which can make the code harder to understand and maintain. It is better practice to declare variables in the smallest scope possible.

4) There are no bounds checking on the input variable which could cause a buffer overflow vulnerability.

5) The code does not have a return statement, which can result in undefined behavior. The main function should always return an integer to indicate the exit status of the program.

A. CodeX

To evaluate the performance of these models, we began by tasking CodeX with replicating the exploit by generating a C function that takes user input and displays it as an output. The code snippet below demonstrates the generated code by CodeX.

```
#include <stdio.h>
int main( ){
   char input [100];
   printf("Please enter something:");
   scanf("%s", input);
   printf("You entered: %s\n", input);
   return0;
}
```

The generated code by CodeX covers most of the remediations to address the vulnerabilities and issues present in the original code however it is still missing the error checking.

B. CodeBERT

Next, we conducted training sessions with CodeBERT, employing the FORMAI dataset, a substantial repository comprising AI-generated C programs and their corresponding vulnerability classifications. Initially, the model featured 16 attention heads, with a maximum number of embeddings capped at 1024. However, the model's performance did not meet our expectations, achieving an accuracy rate of only 84%. Furthermore, it failed to provide secure built-in function suggestions as replacements for the masked functions. Below, we provide an illustration of the CodeBERT model's input and output using the same code snippet:

CodeBERT Input:

```
#include <stdio.h>
int main( ){
   char input [100];
   printf("Please enter something:");
   <mask> ("%s", input);
   printf("You entered: %s\n", input);
   return0;
}
```

CodeBERT Output:

```
#include <stdio.h>
int main( ){
   char input[100];
   printf("Please enter something:");
   =("%s", input);
   printf("You entered: %s\n", input);
   return0;
}
```

C. GPT 3.5

In the last category, we assigned GPT the following two tasks: First, to generate a code that replicates the functionality outlined in the original code snippet. Second, to analyze the original vulnerable code, offer security improvement recommendations, and implement fixes to transform the original code into a secure, non-vulnerable version. The generated code by GPT is as follows:

```
#include <stdio.h>
int main( ){
   char userInput[100];
   fgets(userInput, 100, stdin);
   printf("You entered: %s", userInput);
}
```

The code generated by GPT 3.5 successfully addresses several vulnerabilities, such as bound checking, input validation, global variable, and format string vulnerabilities. However, it fails to address the error checking and return statement issues. Interestingly, when provided with the exploit and asked to suggest security improvements, GPT addresses the return statement and error-checking concerns but overlooks the input validation, bound checking, global variable, and format string vulnerabilities. Moreover, when GPT 3.5 is presented with its non-exploit code suggestion, it indicates that there are no apparent vulnerabilities in the code. The improved original snippet generated by GPT is as follows:

```
#include <stdio.h>
#include <stdlib.h>
int main( ){
   int input;
   if(scanf (”%d”, &input) != 1){
      printf(”Error: invalid input \n”);
      return 1;
   }
   printf(”%d\n”, input);
   return 0;
}
```

Similar to the described method with the vulnerability example that was just described, we evaluated the performance of the three models on all vulnerability categories collected in our prior work that were described in Section III. The aforementioned vulnerability categories and the overall performance results of the models is presented in Fig. 2.

As [14] mentioned, user interactions with code generation AI models can be categorized into acceleration and exploration experiences. In acceleration mode, the programmer already knows what they want to do next, and the code generation model helps them get there quicker. In exploration mode, the programmer is not sure how to proceed and uses a code generation model to explore their options or get a starting point for the solution. Based on our experiment, both GPT 3.5 and CodeX are more suitable for developers in the acceleration mode as a debugging assistant [15] and the use of GPT 3.5 may be potent to security vulnerabilities if used by beginner-level developers in the exploration mode.

V. CONCLUSION & FUTURE WORK

In conclusion, this study aimed to compare and analyze the code generation capabilities and security measures taken for code generation in GPT 3.5, CodeBert and CodeX. The results showed that CodeX had the highest code generation capability, generating code that was accurate, secure, and privacy-preserving. GPT 3.5 had relatively lower code generation capabilities compared to CodeX, but it excelled in explaining potential vulnerabilities, commenting on the code, and analyzing log files, enabling students to gain a better understanding of HTTP requests. CodeBert also demonstrated a high code generation capability but was weaker in terms of security measures. It is essential to consider the complexity of the task, the clarity of the instructions, and the model's level of understanding of the programming language and relevant frameworks to ensure the quality and accuracy of the generated code. As suggested by [15] code, generating AIs cannot fully substitute for professionals whose responsibilities extend beyond mere coding. However, it offers a range of possibilities for individuals involved in coding by facilitating prompt engineering and improving coding skills. The findings of this study can provide a basis for future research on improving the security measures of AI code-generating models while enhancing their code-generation capabilities [16].

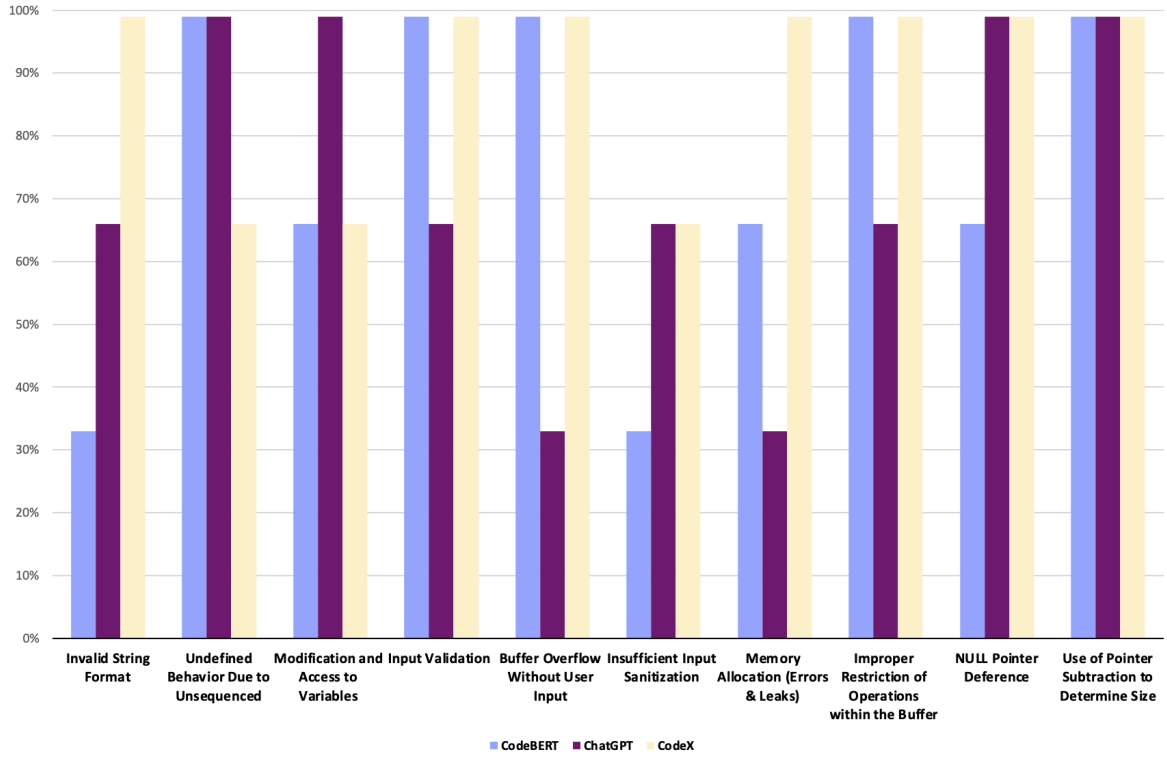

Fig. 2. Overall Performance of the 3 models

The area for future work could be exploring existing bias in these models when generating code [15]. Additionally, research can be conducted to analyze the effectiveness of various security measures and policies in preventing AI model attacks and ensuring the privacy and security of generated code in software engineering education [17]. The potential of generative AI to transform software engineering education by automating routine tasks, accelerating learning, personalizing instruction, and enhancing collaboration. It recognizes that while AI can play a pivotal role [20], educators remain central to nurturing the next generation of software engineers while emphasizing ethical considerations in integrating AI technologies. Generative AI has the potential to revolutionize software security and vulnerability management by automating tasks, improving threat detection, and enabling faster response to emerging threats. However, it also comes with challenges that need to be managed as this technology becomes more integrated into cybersecurity practices [21].

REFERENCES

[1] Parashar, Binayak, et al. ”Revolutionary transformations in twentieth century: making AI-assisted software development.” Computational Intelligence in Software Modeling 13.1 (2022).

[2] Rietz, Tim, and Alexander Maedche. ”Cody: An AI-based system to semi-automate coding for qualitative research.” Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems. 2021.

[3] Perry, Neil, et al. ”Do Users Write More Insecure Code with AI Assistants?.” arXiv preprint arXiv:2211.03622 (2022).

[4] Taeb, Maryam, and Hongmei Chi. ”A personalized learning framework for software vulnerability detection and education.” 2021 International Symposium on Computer Science and Intelligent Controls (ISCSIC). IEEE, 2021.

[5] Husain, H., Wu, H. H., Gazit, T., Allamanis, M., & Brockschmidt, M. Codesearchnet challenge: Evaluating the state of semantic code search. (2019)

[6] Yan, S., Yu, H., Chen, Y., Shen, B., & Jiang, L. (2020, February). Are the code snippets what we are searching for? a benchmark and an empirical study on code search with natural-language queries. In 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER) pp. 344–354

[7] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: code recommendation via structural code search. Proc. ACM Program. Lang. 3, OOPSLA, Article 152 (October 2019), 28 pages.

[8] Lv, F., Zhang, H., Lou, J. G., Wang, S., Zhang, D., & Zhao, J. (2015, November). Codehow: Effective code search based on api understanding and extended boolean model (e). In 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) pp. 260–270

[9] Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., & Tarlow, D. (2016). Deepcoder: Learning to write programs.

[10] Alon, U., Zilberstein, M., Levy, O., & Yahav, E. (2019). code2vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages, 3(POPL), 1-29

[11] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language models are unsupervised multitask learners. OpenAI blog, 1(8), 9

[12] Svyatkovskiy, A., Deng, S. K., Fu, S., & Sundaresan, N. (2020, November). Intellicode compose: Code generation using transformer. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp. 1433-1443)

[13] Hanif, Hazim, and Sergio Maffeis. ”Vulberta: Simplified source code pre-training for vulnerability detection.” 2022 International Joint Conference on Neural Networks (IJCNN). IEEE, 2022.

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

[15] Borji, Ali. ”A categorical archive of ChatGPT failures.” arXiv preprint arXiv:2302.03494 (2023).

[16] Kasneci, Enkelejda, et al. ”ChatGPT for good? On opportunities and challenges of large language models for education.” Learning and Individual Differences 103 (2023): 102274.

[17] Zhuo, Terry Yue, et al. ”Exploring AI ethics of ChatGPT: A diagnostic analysis.” arXiv preprint arXiv:2301.12867 (2023).

[18] Kazemitabaar, Majeed, et al. ”Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming.” Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems. 2023.

[19] Daun, Marian and Brings, Jennifer. ”How ChatGPT Will Change Software Engineering Education”, proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1,110– 116, 2023

[20] Happe, Andreas, and Ju¨rgen Cito. ”Getting pwn’d by AI: Penetration Testing with Large Language Models.” arXiv preprint arXiv:2308.00121 (2023).