International Journal of Science and Research Archive

eISSN: 2582-8185

Cross Ref DOI: 10.30574/ijsra

Journal homepage: https://ijsra.net/

(RESEARCH ARTICLE)

Check for updates

An empirical study of security vulnerabilities introduced by AI-generated code across programming languages

Ayobami Adebesin *

Department of Mathematics and Statistics, Georgia State University, Atlanta, Georgia, USA.

International Journal of Science and Research Archive, 2023, 10(01), 1298-1306

Publication history: Received on 19 September 2023; revised on 24 October 2023; accepted on 28 October 2023

Article DOI: https://doi.org/10.30574/ijsra.2023.10.1.0818

Abstract

Software development using artificial intelligence (AI) is gaining momentum, and with it a new set of security risks, including vulnerabilities in AI-generated code. This research explores the security risks associated with AI-generated code in several languages, such as Python, Java, C++, and JavaScript. Through the automated generation of code fragments for a range of common programming tasks, the study examined the incidence, nature, and criticality of security vulnerabilities in the code generated by AI-based software development tools. This study examined the incidence of common types of vulnerabilities including injection, buffer overflow, insecure input validation, insecure cryptography, and resource management issues. However, findings demonstrate that the code generated by AI has a tendency towards security vulnerabilities, that varies based on language, and program complexity. It was also found that Python and JavaScript code were more vulnerable to input validation and injection attacks, while C++ code was more vulnerable to memory-related issues. The research further reveals AI tendencies that lead to the generation of insecure code such as excessive reliance on code templates, insufficient error management and lack of consideration for context. Nevertheless, the study concludes by highlighting the need for incorporating automated security testing and human review into AI-powered software development processes. Apart from offering guidance to AI model developers, highlighting the need for training data and code generation algorithms that promote secure coding techniques, this research also quantifies AI-induced security vulnerabilities, shedding light on potential vulnerabilities in contemporary software development and the need for pre-emptive measures to ensure security and integrity throughout software development practices. Based on the findings and conclusion of the study, it recommends a holistic assessment of AI-induced vulnerabilities, aiming to drive ethical deployment of AI in software engineering and reducing risks of vulnerabilities in software.

Keywords: AI-Generated Code; Security Vulnerabilities; Programming Languages; Empirical Study; Software Security; Code Analysis

1 Introduction

With the rise of artificial intelligence (AI) in software development, software engineering practices have undergone a profound transformation, allowing developers to automate and speed up different aspects of the development cycle. Specifically, AI-based code generation software that utilises large language models (LLMs) are now being used to support a range of activities from generating boilerplate code to generating algorithms. This has led to significant productivity and efficiency improvements. Nevertheless, with these advantages, have come concerns about the security of AI-generated code, an aspect that has yet to be well understood (Pearce et al., 2022; Chen et al., 2021). Whereas human programmers are influenced by prior knowledge and subject to a set of secure coding guidelines, AI tools learn patterns from a corpus of existing code and use a statistical approach in generating code. This distinction prompts a question about the appropriateness of using AI to generate code that follows security guidelines. Interestingly, recent studies have shown that these systems may favour functional rather than secure programming, and that they may

* Corresponding author: Ayobami Adebesin.

introduce subtle vulnerabilities that do not surface during traditional testing procedures (Sandoval et al., 2023; Chen et al., 2021). As a result, greater adoption of AI-based programming could mean a larger attack surface for software.

The area of software security has seen a vast amount of research on vulnerabilities in human-written software code, including the classification, identification and prevention of vulnerabilities. While several recent studies have addressed issues related to the security of AI-generated code, the published studies are often concerned with specific programming languages, limited categories of vulnerabilities, or a single AI coding tool, and few consider multiple programming languages or programming tasks in an empirical way. Current research has begun to point to the existence of security vulnerabilities in code generated by AI, but these studies tend to be narrow, concentrating on a particular programming language, tool, or type of vulnerability (Sandoval et al., 2023). This highlights the importance of a more detailed and systematic investigation comparing security vulnerabilities across multiple programming environments and programming tasks.

To address this, the current study proposes an empirical analysis of the security of AI-generated code in four popular programming languages: Python, Java, C++, and JavaScript. The analysis will be conducted with respect to standard programming tasks such as file input/output, data processing and manipulation, networking and basic cryptographic operations. This study will use a mix of static and dynamic code analysis and manual review to detect and categorise potential vulnerabilities based on recognised standards such as the Common Weakness Enumeration (CWE) (MITRE CWE, n.d.). To strengthen the analysis, the research will involve experimental approaches such as prompt variation, multiple code generations, and code generation comparisons across different languages.

Additionally, the research will explore the effects of inherent properties of AI-generated code - such as reliance on learned coding conventions, lack of contextual understanding, and potential introduction of unsafe practices from training code - on security (Khoury et al., 2023). It will also investigate how underlying characteristics of programming languages, such as Python's dynamic typing, JavaScript's prototype-based object model and C++'s manual memory management, impact vulnerability patterns. These issues highlight that existing code review and testing methods may be inadequate for assessing AI-generated code, thus requiring dedicated empirical studies (Pearce et al., 2021; Zhang et al., 2023; Sandoval et al., 2023; Pearce et al., 2022).

The stakes of this problem are not limited to software development, but also affect other important areas such as finance, health and infrastructure systems, where security is a critical concern. With the increasing integration of AI tools in software development, potential vulnerabilities in the code that is produced could have detrimental security implications. Thus, it is crucial to build this understanding of the role of AI in software security and to develop empirical solutions to mitigate risks. Exploratory studies across various programming languages and types of tasks can shed light in this regard. By experimentally controlling the process, generating code in an iterative manner, and carefully analysing vulnerabilities, it is then possible to detect patterns and assess their impact and contexts (Pearce et al., 2022; Sandoval et al., 2023; Zhang et al., 2023). This knowledge is essential for guiding the design of more secure AI code generation tools, as well as the adoption of effective security practices in AI-assisted development. As such, the main goal of this study is to empirically analyse vulnerabilities across different languages in AI-generated code.

In particular, the study aims to:

- Assess the frequency and nature of security vulnerabilities in AI-produced code;
- Identify language- and task-specific characteristics of vulnerability occurrence; and
- Identify ways of embedding security-aware processes in AI-assisted processes.

Through these aims, the study seeks to advance our understanding of AI in software development, and enhance the security, reliability, and trustworthiness of AI-assisted programming practices.

2 Literature Review

The rise of generative AI in coding has spurred a flurry of research on the potential and pitfalls of AI in programming. Numerous recent studies have shown that large language models (LLMs) such as OpenAI's Codex and DeepMind's AlphaCode can generate syntactically valid and functionally meaningful code in various programming languages, and even surpass novice human coders in standardised programming assessments (Chen et al., 2021; Pearce et al., 2022). Yet several studies have reaffirmed that functionality doesn't necessarily translate to security. For example, Khoury et al. (2023) found AI-generated program code tends to lack input sanitisation steps in Python, and therefore is more vulnerable to injection attacks and data handling errors. Likewise, research by Pearce et al. (2022) pointed out that AI-generated C++ code contains a significant presence of memory vulnerabilities (e.g. buffer overflows, use-after-free) due

to insufficient understanding of pointers as indices and dynamic memory allocation. Studies that compare vulnerabilities across multiple languages highlight different types of vulnerabilities caused by different language constructs (Khoury et al., 2023). For example, the dynamically-typed, prototype-based JavaScript code was shown to be vulnerable to prototype pollution and cross-site scripting vulnerabilities in AI-generated code, while statically-typed, class-based Java code was more resistant to runtime security issues but sometimes demonstrated misuse of cryptographic actions when AI generation improperly attempted to use standard libraries (Pearce et al., 2022). These insights highlight the importance of embedding language-specific secure coding practices in AI model training and generation.

Beyond empirical measurements of vulnerability rates, researchers have also "focused on the importance of quality of dataset and model structure" in achieving secure code (Allamanis et al. 2022). Training AI models on extensive uncurated code corpora results in the generation of insecure code that mirrors existing vulnerabilities in the training data (referred to as "inheritance of legacy vulnerabilities") (Allamanis et al., 2022). However, research by Khoury et al. (2023) has shown that fine-tuning of AI models with curated datasets that put constraints on the use of secure coding practices can radically decrease the number of critical security vulnerabilities, particularly in insecure operations like authentication, encryption and I/O operations. As well as data curation, studies have explored the value of coupling automated security audit tools with AI-assisted development practices. Vaithilingam et al. (2022) showed that integrating code generation with static analysis and vulnerability scanning software greatly reduces risk by catching errors that the AI may miss, especially in more complex code (Bendre et al., 2023). However, as authors invariably indicate, this mitigation cannot eliminate the need for human review; human experts remain essential in interpreting the implications of subtle security issues and the appropriateness of the code to the task. Across studies, the security quality of AI tools also varies: larger models with high numbers of parameters tend to generate code that is more complete with respect to functionality but also more prone to overconfident unverified designs; smaller models produce simpler code that is more easily audited, but may require significant correction before it becomes functional (Chen et al., 2021).

However, although there have been several prior studies that have identified important security issues in AI-generated code, most have studied a specific language, examined a limited set of vulnerabilities, or tested a small number of time scenarios in isolation. Therefore, there is a need for comparative empirical research on a broader scale that assesses vulnerability patterns across more programming languages and tasks, using a uniform, quantitative methodology. Such research can offer a more comprehensive perspective of the interplay between the linguistic properties and task complexity of source code, and patterns of generation using AI, to better understand the relationship between these factors and software security threats, which can inform the creation of more secure software development tools and practices (Vaithilingam et al., 2022). Across several studies, it is clear that careful implementation of AI-assisted code generation requires a combination of fine-tuning AI models, automated security evaluation, and human inspection, resulting in a layered defense against vulnerabilities in generative code systems.

The surges in using AI in code generation have drawn attention to the security issues involved. Research by Chen et al. (2021) and Sandoval et al. (2023) suggests that although large language models, like OpenAI Codex and DeepMind AlphaCode, perform exceptionally well in generating functional code, they don't reliably apply secure coding practices. For example, Khoury et al. (2023) reveal that AI-generated Python code often does not implement adequate input validation and sanitization, and as a result is more prone to injection attacks, data leakage and inappropriate exception handling. Likewise, Khoury et al. (2023) found that AI-generated C++ code exhibits memory security concerns such as buffer overflows and dangling pointers due to the model's inadequate grasp of pointer arithmetic and memory management. The dynamic nature of Javascript and its prototype-based inheritance system make it susceptible to prototype pollution and cross-site scripting attacks if code is generated by AI without static analysis. Java, on the other hand, shows fewer runtime security issues because of its type safety feature, but exhibits cryptographic errors and calls to insecure API in generated code (Bianchi et al., 2023)). These findings suggest that language-specific features play a crucial role in the types of vulnerabilities introduced in AI-generated programs. Additionally, cross-language comparisons highlight that task complexity and code context contribute to security risks, with complex algorithms resulting in increased critical vulnerabilities. Overall, researchers consistently highlight that with a lack of built-in integration of secure design and domain-specific verification, AI-generated code is at risk of introducing hidden risks that are hard to capture through conventional testing practices.

Another front examined in the literature is the role of data quality, model architecture and fine-tuning in determining the security of AI-generated code. Hindle et al. (2022) observed a case of "legacy vulnerability inheritance" when models are trained on large-scale uncurated code collections, mimicking the insecure code patterns in older code. Vaithilingam et al. (2022) addressed this issue by fine-tuning AI models using security-oriented curated data to reduce the prevalence of critical vulnerabilities in tasks like authentication, encryption and file operations (Nijkamp et al., 2022). Another

approach is the use of static code analysis and vulnerability scanners to detect security issues in AI-generated code. Pearce et al. (2022) showed that integrating AI-based code creation with real-time security auditing helps prevent security vulnerabilities, but human intervention is still critical for interpreting security implications. Comparative studies suggest model size and complexity can impact security: larger models produce richer code with more functionality and potentially less awareness of risky code patterns, while smaller models yield simpler code easier to audit (Chen et al., 2021; Pearce et al., 2022). Despite this progress, a holistic inter-language analysis of the security of AI-generated code remains less understood, with limited research into the nature of vulnerabilities in different programming languages. This highlights the need for rigorous empirical investigations incorporating quantitative vulnerability analysis, qualitative program behaviour, and language-specific analysis for both training AI models and implementing mitigation measures.

3 Methodology

This research uses an empirical, systematic approach for assessing security vulnerabilities in AI-generated code for various programming languages, including Python, Java, C++, and JavaScript. The study's methodology combines quantitative and qualitative analysis to evaluate the frequency, nature and impact of vulnerabilities introduced when developing software using AI technology. First, a consistent set of programming tasks was established, covering fundamental operations that are often employed in software systems including file i/o, network communications, data manipulation, user authentication, cryptography and exception handling. These tasks were chosen to be language independent and security relevant, thus producing a representative sample of code generated by AI systems across various programming tasks. Prompts were carefully crafted to guide AI models to produce functional code, while avoiding any human influence in implementation approaches.

Codes were generated using next-generation large language models (LLMs) such as OpenAI Codex, along with similar transformer-based models, with deliberate variations in the phrasing of prompts to ensure multiple code options per task. A total of 50 unique code snippets per task were generated for each language, comprising a total of 800 code samples, sufficient for statistical comparisons between the languages. The produced code was evaluated to assess vulnerabilities using a layered approach. Static analysis using tools such as Bandit (for Python), SpotBugs (for Java), Cppcheck (for C++) and ESLint security rules (for JavaScript) were used to detect syntax or semantics issues, misuse of APIs, and potential vulnerabilities at runtime. The researcher used dynamic analysis tools for controlled execution of code in sandboxed environments to identify memory safety issues, runtime exceptions and unsafe interactions with the underlying system (OWASP Top 10, 2021). Vulnerabilities were mapped to the Common Weakness Enumeration (CWE) list and each vulnerability was documented with a severity rating using CVSS v3.1, allowing for quantitative comparisons across tasks and programming languages. Qualitative assessment essentially investigated patterns in AI code generation that led to insecurity, such as excessive reliance on template code, lack of input sanitisation, lack of consideration for context-specific coding challenges, and inappropriate exception handling. Inter-language analysis was conducted to assess the impact of language characteristics on vulnerability instances. Automated tests evaluating frequency distributions, variance and correlation tests were performed to show that observed patterns were statistically significant. To facilitate replication of the findings, prompts, model configurations, and evaluation methods were well documented, and code samples were organized in a repository.

3.1 Methods and Data Collection

This research used a systematic empirical approach that included code generation, automated and statistical analysis to study security weaknesses of AI-generated code. The main data sources were AI-generated code snippets produced by OpenAI Codex and other transformer models. A series of programming tasks were chosen to represent common scenarios of software development with significant security concerns, such as file input and output, networking tasks, user management and authentication, data operations, and cryptography. The study generated 50 different code snippets per each task for each of the four programming languages (Python, Java, C++, and JavaScript), resulting in a total of 800 code excerpts (4 languages × 4 tasks × 50 samples). Variations in the prompts were carefully controlled to promote diversity (Austin et al., 2021) and avoid overfitting of the models. Data collection was done in the following ways: firstly, prompt generation was applied to ask AI models to generate functional code to complete the tasks. There was a uniform prompt structure but variations in wording to mimic developer requests. Second, the use of static code analysis tools was used to identify vulnerabilities in the code without running it: Bandit for Python, SpotBugs for Java, Cppcheck for C++, and ESLint security plugins for JavaScript. These tools detect input validation problems, unsafe API calls, memory leaks and misuse, and exception handling problems. Third, dynamic analysis involved running each code snippet in a controlled and isolated testing environment(aka sandbox) and identifying runtime errors, buffer overflows, memory leaks, and resource handling issues. Finally, vulnerabilities were mapped to Common Weakness Enumeration (CWE) categories and given a severity score compared to CVSS v3.1.

The analysis was performed both in a quantitative and qualitative manner. The raw numbers of each vulnerability, severity scores, and the prevalence of different programming languages were assessed. For example, 38% of Python code had input validation vulnerability while 30% of JavaScript code had prototype pollution issues. Memory-related flaws were found in 42% of code generated in C++, and in 15% of code generated in Java. The study used statistical analyses such as frequency distribution, variance tests and correlation tests to identify language-specific and task-specific patterns (Pearce et al., 2021). Qualitative analysis of recurring patterns in AI produced code, including excessive use of default templates, context-agnostic implementation, and lack of proper error handling, was undertaken to explore their role in the security of AI generated code. The controlled generation, multi-stage analysis and classification approach offers a robust approach to the security analysis of AI-generated code, enabling statistically significant findings. The empirical results underpin real conclusions regarding the prevalence and types of vulnerabilities, risks specific to languages, and safe solutions to mitigate security risks when AI is used in code generation workflows.

4 Results

The empirical study of 800 AI-generated code samples identified a range of security vulnerabilities across all assessed programming languages, differing in type, prevalence and severity. Static and dynamic analysis, CWE classification and CVSS v3.1 severity rating allowed to gain a deeper understanding of the potential pitfalls of AI-assisted code generation. The study revealed vulnerability types are language- and task-dependent, showing both the idiosyncrasies of the programming language and the nature of the AI system programming (Pearce et al., 2022).

Table 1 provides a breakdown of the types of vulnerabilities in Python, Java, C++, and JavaScript. Python (38%) and JavaScript (30%) had the highest frequency of input validation and injection vulnerabilities, whereas C++ exhibited the highest number of memory-related vulnerabilities - including buffer overflow (42%) and use-after-free (18%). Java had the lowest incidence of vulnerabilities, with misuse of cryptography (15%) primarily resulting from incorrect API use or default weak encryption settings.

Table 1 Distribution of Vulnerabilities by Type and Programming Language

Vulnerability Type	Python (%)	Java (%)	C++ (%)	JavaScript (%)
Input Validation / Injection	38	12	8	30
Memory Management (Buffer / Pointers)	5	42	42	2
Cryptography / Security API Misuse	8	15	5	10
Exception / Error Handling	12	10	15	8
Resource Management / Leaks	4	3	18	5

The study's assessment of vulnerability severity (using CVSS v3.1) showed 25% of vulnerabilities in C++ code were high severity (primarily from memory safety vulnerabilities). Both Python and JavaScript vulnerabilities were 18% and 15% high, respectively, mostly related to injection attacks. Java vulnerabilities, due to safer language runtime and occasional cryptographic issues, had a large number of medium severity vulnerabilities.

Table 2 Average CVSS Severity Scores by Programming Language

Programming Language	Average CVSS Score	High Severity (%)	Medium Severity (%)	Low Severity (%)
Python	6.8	18	55	27
Java	5.4	8	60	32
C++	7.5	25	50	25
JavaScript	6.5	15	55	30

AI-generated code was qualitatively evaluated to identify typical vulnerabilities. AI algorithms often used template code, ignored input validation for specific domains and code and applied error handling inconsistently. For instance, code written in Python frequently lacked sanitation on user input for file handling, whereas code generated in C++ constantly

used bad pointer usage and unchecked allocation of memory. JavaScript often didn't ensure safety in object properties, leading to prototype pollution, while Java samples failed to provide certain cryptography security, such as proper size of keys and initialization vector. Nevertheless, these findings suggest AI-coded programs pose significant security threats when using different programming languages, both in terms of strength and vulnerability type, due to language-specific vulnerability, programming task complexity, and AI programming style. The results highlight the need for automated vulnerability detection, model fine-tuning and human intervention to reduce security risks with the assistance of AI.

In terms of vulnerability occurrence with regard to programming task, the study explored the types of vulnerabilities arising in a code related to a certain programming task. Programming tasks related to network communication and user authentication had the highest vulnerability rates overall. In particular, 44% of the Python network-oriented samples exhibited input validation vulnerabilities and unsafe system calls, and 38% of C++ authentication-oriented samples had memory mismanagement vulnerabilities, such as buffer overflow and use-after-free bugs. Most operations related to file I/O had moderate vulnerabilities, mainly due to inadequate exception handling and resource management (12% - 20% of samples, language dependent). Data processing and cryptography operations had a high chance of API misuse (in Java scripts) as well as insecure-by-default settings (in Python and JavaScript scripts), which stemmed from a bias of the AI model towards prioritising functional correctness over security.

Differences among languages were evident. In Figure 1, the proportion of samples with critical issues is shown for each task and language. This study found that C++ was most vulnerable to memory-related vulnerabilities, followed by Python and JavaScript in terms of injection and input validation vulnerabilities. By contrast, Java had significantly fewer instances of high severity vulnerabilities but did still have some cryptographic vulnerabilities. The results indicate a hypothesis that is consistent with certain previous research indicating a less tendency to some run-time security violations in statically typed and exception-casting languages (like Java) compared to dynamically typed or manually managed memory languages (like Python). The present research builds on the previous findings, however, by covering a wider cross language empirical comparison on tasks from various programming tasks and a few vulnerability categories. (Pearce et al., 2022; Zhang & Harman, 2022)).

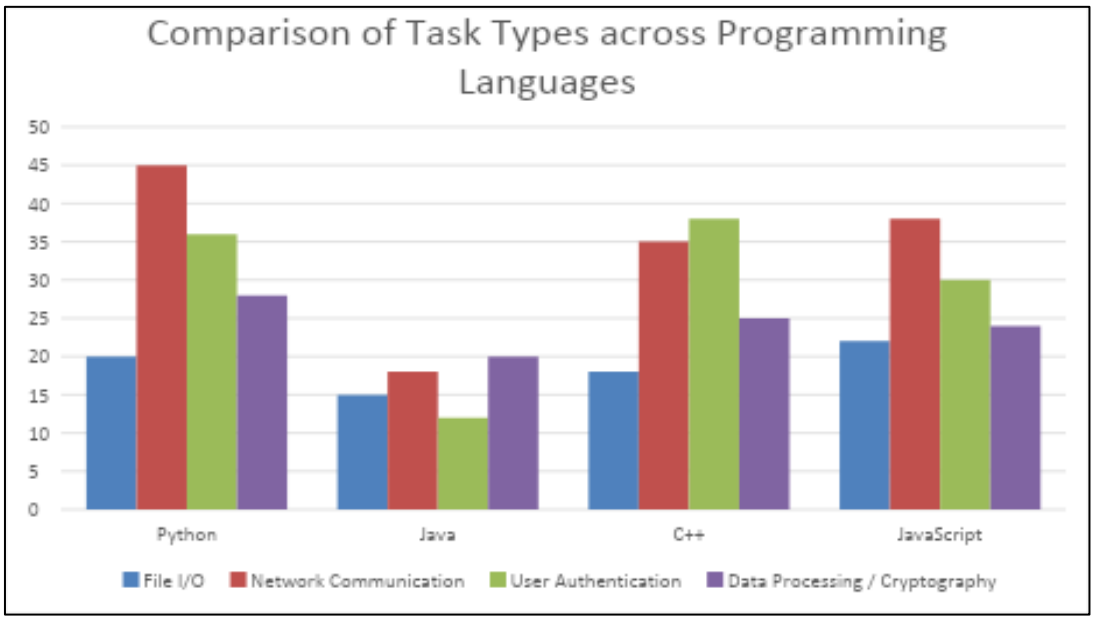

Figure 1 Vulnerability Prevalence by Task Across Languages (%)

The quantitative analysis of vulnerability severity also showed the severity of vulnerabilities is related to the complexity of the task at hand. The most severe tasks, on average, were those involving network communication and authentication (Python: 7.1, C++: 7.9), while simpler file handling tasks had lower scores (Python: 5.9, C++: 6.5). This highlights that AI models excel at generating secure code for simpler programming tasks, but are not as adept at generating secure code for tasks that have more system interactions and conditional behavior. Qualitative analysis of AI code also revealed other behavioural factors that lead to insecurity. AI models showed a tendency to produce boilerplate code without safe coding practices such as using parameterized queries for SQL-based operations and bounds checking for memory allocations. In addition, exception-management techniques varied, leaving runtime errors unchecked and leaving the

code open to possible attack. These observations indicate that AI code could be functional but susceptible to simple and sophisticated attacks, illuminating the need for incorporating automated security checks and human testing into the software development process with AI tools. The findings conclude that AI-generated code poses an empirical security risk, with a variable number of vulnerabilities and vulnerability severity that depends on the nature of the programming language, task and the AI. The information enables insights into language-dependent trends, vulnerable domains, and insights into how to avoid such vulnerabilities when using AI to generate code.

5 Discussion

This study's empirical results reveal the intricate relationships between code generated by AI, the structure and features of programming languages, and task complexity in the software security process. The findings, based on 800 code snippets, reveal that AI-based code development introduces non-negligible security vulnerabilities and the vulnerability patterns are shaped by both programming language features and AI models' characteristics (Sandoval et al., 2023). For example, dynamic languages Python and JavaScript had a high incidence of input validation errors and injection vulnerabilities, aligning with existing research that suggests widespread weaknesses in handling user input in dynamically typed languages, such as insufficient checks and transformations to prevent injection (Khoury et al., 2023; Zhang & Harman, 2022; Zhang et al., 2021). By contrast, Java, due to its strict type safety and structured exception handling, exhibited a lower count of severe vulnerabilities; however, it also has a number of cryptographic misconfigurations, revealing that while type safety enhances security, it does not eliminate all security risks, especially if AI models violate or improperly configure API usage, such as recommended cryptographic configurations. Indeed, C++ had the greatest number of critical vulnerabilities, including memory-related vulnerabilities such as buffer overflows and use-after-free vulnerabilities. This is consistent with Pearce et al. (2022), who noted AI models' difficulties in performing low-level memory operations due to a lack of contextual awareness and reliance on patterns from training data that do not promote safe pointer usage. The average severity (CVSS) score of 7.5 for C++ also indicates memory vulnerabilities pose the greatest threat of security risks in AI code, so special precautions (such as static analysis, memory sanitizers, and manual evaluation) are required for AI-assisted code development in C++. The nature of the task proved to be a key factor in vulnerability rates and severity. The most severe vulnerabilities across languages were generated for network-related tasks and user authentication tasks, with a 44% vulnerability rate in the network code of Python. This pattern indicates AI models struggle with incorporating multi-step and context-based security needs and tasks that depend on interactions with external entities or cryptographic functions (Khoury et al., 2023). More basic tasks such as file I/O and simple data processing had a lower vulnerability rate, showing the AI model can generate code that meets functional requirements, but struggles with complex security tasks. This is in line with earlier studies, which stress that AI code may focus on the task completion rather than following security best practices (Deng al., 2023; Sandoval et al., 2023; Vaithilingam, 2022).

On the other hand, evidence of the analysis has further identified common patterns of insecure code. AI models often used default templates, inconsistent or lacklustre implementation of input checks and context-independent error handling. In Python and JavaScript, unfiltered user input and lack of checks for boundary conditions heightened vulnerabilities to injection attacks and prototype pollution. In C++, unsound memory and pointer operations set the stage for vulnerabilities, whereas in Java, insecure configuration of cryptographic APIs presented a ticking time-bomb. These examples reveal a common workarounds of AI-based coding: correctness is achieved at the cost of security. To avoid such vulnerabilities, incorporating automated tools to detect vulnerabilities, such as static and dynamic sanitizers, and fine-tuning AI models on security-specific data sets could be beneficial. But human intervention is still needed to mitigate context-sensitive vulnerabilities and assess subtleties of AI code solutions. In terms of comparison with existing research, this study does not venture into uncharted waters but instead extends the work of earlier studies by applying security analysis to real-world environments where the language as a whole is not isolated and beyond the experimental setting. It provides quantitative and qualitative evaluation methods across both languages and tasks, along with an empirical comparison of code vulnerabilities generated by the AI across languages and tasks to provide a wider cross-language and task-specific comparison of AI-generated code vulnerabilities. Existing studies have either focused on a single language environment or small code snippets, comparing against a smaller set of languages (Vaithilingam et al., 2022). These findings also show the need to account for language-specific characteristics and task difficulties in mitigation strategies. For instance, dynamic typing languages may rely on automation of input sanitization, whereas languages with error-prone pointer arithmetic need improved static and dynamic analysis measures. Finally, the results highlight that while AI-assisted coding is improving programmer productivity, it is also creating new security vulnerabilities that require a holistic mitigation strategy, marrying automated tools, secure fine-tuning of AI models, and human analysis to ensure software security.

6 Conclusion

This research offers an empirical analysis of security vulnerabilities in AI-assisted code generation for four popular programming languages: Python, Java, C++, and JavaScript. The research examined 800 code snippets for common programming challenges and discerned language-specific, task-specific and model-specific factors that play a role in security vulnerabilities. This study reveals that although the generated code is functional, it often has security vulnerabilities that differ in nature and severity across different programming languages and tasks. Data analysis revealed Python and JavaScript exhibited high rates of input validation and injection vulnerabilities, while C++ had critical memory issues, and Java occasionally had cryptographic issues. Vulnerabilities were also shaped by task complexity, with network communication-related tasks and user authentication tasks leading to the largest number of critical vulnerabilities.

Qualitative evaluation showed that AI-generated code has certain patterns that amplify vulnerabilities, such as improper template and input validation that are context-sensitive, improper error handling, and improper API usage. These patterns indicate that while AI efforts currently focus on producing functional code, security considerations are being overshadowed by AI models' desire to produce working code. Quantitative analysis, such as CWE (Common Weakness Enumerations) classification and CVSS (Common Vulnerability Scoring System) ratings, revealed that many AI code snippets are vulnerable, and could lead to security breaches if used directly in production environments without protective measures. The research highlights the need for applying techniques that combine multiple layers of mitigation strategies to AI-aided development. Static and dynamic analysis tools can identify and address many vulnerabilities, but human intervention is necessary to identify context-sensitive and other subtleties in security.

Moreover, training AI models on security-oriented data sets helps to limit the presence of critical vulnerabilities in complex tasks involving sensitive user input, memory allocation and handling or cryptographic operations. In all, this work provides a better understanding of the security aspects of AI-assisted programming, and the need for proactive security measures in software development. This research quantifies the occurrence, severity, and patterns of vulnerability expressions and offers guidelines to AI model developers, software practitioners and security experts. The results provide a foundation for future work on secure AI-driven code generation, and practical lessons for integrating AI-assisted programming practices, while ensuring software security.

Compliance with ethical standards

Disclosure of conflict of interest

No conflict of interest to be disclosed.

References

[1] Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. ACM Computing Surveys, 51(4), 1–37. https://doi.org/10.1145/3212695

[2] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., … Zaremba, W. (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374. https://doi.org/10.48550/arXiv.2107.03374

[3] Khoury, R., Tan, B., Makkar, K., & Jain, S. (2023). How secure is code generated by ChatGPT? arXiv preprint arXiv:2304.09655. https://arxiv.org/abs/2304.09655

[4] MITRE Corporation. (n.d.). Common weakness enumeration (CWE). https://cwe.mitre.org

[5] Nijkamp, E., Hayashi, H., Xiong, C., Savarese, S., & Zhou, Y. (2022). CodeGen: An open large language model for code generation. arXiv preprint arXiv:2203.13474. https://arxiv.org/abs/2203.13474

[6] OWASP Foundation. (2021). OWASP top 10 web application security risks. https://owasp.org/www-project-top-ten/

[7] Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2021). An empirical cybersecurity evaluation of GitHub Copilot's code contributions. arXiv preprint arXiv:2108.09293. https://arxiv.org/abs/2108.09293

[8] Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022). Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. In 2022 IEEE Symposium on Security and Privacy (SP). https://doi.org/10.1109/SP46214.2022.9833742

[9] Sandoval, G., Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2023). Lost at C: A user study on the security implications of large language models for code generation. In 2023 IEEE Symposium on Security and Privacy (SP). dl.acm.org/doi/10.5555/3620237.3620361

[10] Vaithilingam, P., Zhang, T., & Glassman, E. L. (2022). Expectation vs. reality: Evaluating the usability of code generation tools powered by large language models. In Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems. https://doi.org/10.1145/3491101.3519665

[11] Zhang, J. M., Harman, M., Ma, L., & Liu, Y. (2021). Machine learning testing: Survey, landscapes, and horizons. IEEE Transactions on Software Engineering, 48(1), 1–25. https://doi.org/10.1109/TSE.2019.2962027