A Causal-Comparative Study of Security Vulnerabilities in AI-Generated versus Human-Generated Source Code from GitHub

Dissertation Manuscript

Submitted to National University

School of Technology and Engineering

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY IN TECHNOLOGY MANAGEMENT

by

BRENDA CAROL PALMER

San Diego, California

February 2026

Abstract

In this causal-comparative quantitative study, security vulnerabilities of AI-generated code versus human-generated code were analyzed in publicly available software projects. With this growing usage of large language models to create source code, there is still uncertainty on whether AI-generated code presents a higher security risk compared to human-written code. This is a problem that affects software developers, security practitioners, and organizational leaders charged with the responsibility of secure software engineering and technology governing. The theory that informed the study was the Intellectual Capital theory and informed by structured security risk management principles. A code analysis tool tested source code of over 134 publicly available GitHub repositories based on three programming languages. Three research questions were answered by using descriptive statistics, t-tests of independent samples, and chi-square to determine the severity and distribution of security vulnerabilities depending on their code of origin (AI or Human). The outcomes showed that there is a statistically significant difference in the average severity scores, as the human-generated code had a higher average severity score than the AI-generated code. The results indicate that AI-generated code is not more severe than human-generated code yet both need thorough security verification. The research conducted adds new empirical data to the growing body of literature on AI-assisted software development and highlights the necessity of disciplined risk management, secure software engineering practices, and governance irrespective of the source of the code's origin. Future studies need to increase repository samples, consider other programming languages and include complementary security analysis methods.

Acknowledgments

My deepest gratitude extends to my children, Darryl Tyler Palmer (Beloved Builder) and Victoria Carol Palmer (Victorious Joy), whose love, patience, and unwavering support have been my greatest source of strength and joy throughout this journey. Balancing the demands of working full-time while pursuing a Ph.D. has been an immense challenge, and it would not have been possible without their understanding and encouragement. Their ability to bring laughter and light to even the most stressful days has been a constant reminder of what truly matters in life. Their resilience and adaptability in the face of my demanding schedule have continually inspired me. This achievement is as much theirs as it is mine, and I am immeasurably grateful for the light they bring into my life.

I also wish to express my deepest appreciation to Karl Werner, Assistant Director of Enterprise Software and Applications (ESA) at University Corporation of Atmospheric Research (UCAR). Karl generously took time to read each and every one of my dissertation drafts, observing their evolution into the final manuscript. He consistently listened to my triumphs, offered encouragement, and lent an ear when I felt overwhelmed and uncertain of the path ahead. Knowing that someone as intelligent, experienced, and respected as Karl was reading my work in his free time gave me the confidence to believe I truly had what it takes to earn a Ph.D. There were many moments when I doubted myself, but Karl's steady support served as a powerful motivator to keep going, because of Karl Werner, I am now Dr. Brenda Carol Palmer - also known as Dr. Bamboo. I will remain forever grateful to him.

I am also profoundly grateful to my dissertation committee (Chair: Dr. Ben Ayed, Subject Matter Expert (SME): Dr. James Webb, and Academic Reader: Dr. Jason Pittman) for their guidance and thoughtful feedback throughout this process. This was an incredibly rigorous

academic journey, one of the toughest challenges I have ever faced, and I am certain I could not have achieved this without your direction. Thank you for helping me shape this dissertation into the strongest, most complete version it could possibly be.

A heartfelt thank you as well to the Academic Success Center (ASC), and especially to Dr. Brian Aunkst, whose coaching and support were truly lifesaving at times. I worked with many of you during this process, and your dedication, patience, and encouragement made an extraordinary difference. I could not have completed this without you. Thank you!

Lastly, I want to thank UCAR for sponsoring this educational pursuit. UCAR provided the financial support that made this degree possible, and I am sincerely grateful for their investment in my education and professional growth. Above all, I want to thank God for granting me the strength, perseverance, and clarity to use the mind He gave me to earn this doctorate, even while navigating the challenges of perimenopause. Without God, none of this would have been possible.

Table of Contents

Chapter 1: Introduction 1

Statement of the Problem 4
Purpose of the Study 5
Introduction to Framework 5
Introduction to Research Methodology and Design 8
Research Questions 10
Hypotheses 11
Significance of the Study 11
Definitions of Key Terms 12
Summary 14

Chapter 2: Literature Review 16

Theoretical Framework 17
Understanding Software Development 24
Understanding AI 28
Understanding AI in Software Development (SD) 32
Comparing Human-Generated Code with AI-Generated Code 43
Summary 50

Chapter 3: Research Method 52

Research Methodology and Design (Nature of the Study) 52
Population and Sample 54
Materials or Instrumentation 55
Operational Definitions of Variables 56
Study Procedures 59
Data Analysis 61
Assumptions 62
Limitations 63
Delimitations 64
Ethical Assurances 64
Summary 66

Chapter 4: Findings 67

Validity and Reliability of the Data 67
Results 73
Comparison of Results to the Literature Review 79
Summary 83

Chapter 5: Discussion, Recommendations, and Study Summary 85

Discussion 85

Recommendations for Practice 88
Recommendations for Future Research 89
Study Summary 90

References 91

Appendix A Search Terms 107

Appendix B CodeQL Code 110

Appendix C Delimitations Table 136

Appendix D PowerShell Commands 137

Appendix E IRB Approval Letter 138

Appendix F Public GitHub Repository Licenses 140

List of Tables

Table 1 *Contrast of Theories* 24

Table 2 *Discovered AI-Generated Code Security Vulnerabilities* 37

Table 3 *CodeQL Key Features* 39

Table 4 *CodeQL's Workflow* 39

Table 5 *Potential Weaknesses of Security Analysis Tools* 42

Table 6 *Research Insights on Code Generation Security Vulnerabilities* 48

Table 7 *Comparison of Common Security Vulnerabilities* 49

Table 8 *Operational Table of Variables* 58

Table 9 *Gantt Chart* 61

Table 10 *Descriptive Statistics for CVSS-Like Severity Scores by Code Origin* 71

Table 11 *Repositories with and without Detected Security Vulnerabilities by Code Origin* 72

Table 12 *Refined Delimitations* 131

List of Figures

Figure 1 *NIST AI RMF Core Principles* 23

Figure 2 *Diverse Range of AI Users and Stakeholders* 31

Chapter 1: Introduction

Artificial intelligence (AI) is ubiquitous, meaning AI-generated code is steadily making its way into a different array of electronic devices, such as vehicles, washing machines, phones, watches, just to name a few material items used in everyday life. Similarly, GitHub, a widely used platform for software development, enables developers to utilize AI tools to generate code efficiently. AI-generated code helps to enhance productivity by providing software developers with code that can easily be integrated into their software projects (Majdinasab et al., 2024). As the utilization of AI-generated code is growing at what seems to be an exponential rate due to popularity (Suneja et al., 2021), developers, organizations, and the whole technology field must comprehend the effects of AI on software security. Therefore, being mindful of the software security vulnerabilities that could be introduced into software projects when using AI-generated code is even more essential (Okey et al., 2023).

AI-generated source code security vulnerabilities occur when there is incorrect or suboptimal code in large language models (LLMs) that are trained with datasets that do not adhere to proper security practices (Cotroneo et al., 2025). This results in opportunities for AI-generated source code security vulnerabilities, which are easy to exploit in the hands of an attacker (Idrisov & Schlippe, 2024). Another well-known security vulnerability in AI-generated code is where the LLMs do not have contextual sense (Ogundare et al., 2022). LLMs can generate functional code; however, AI cannot realize what the project should or should not perform as defined in the specified requirements for the project (Ogundare et al., 2022). For instance, software developers may rely on AI to understand that certain code relies on its software environment, which AI cannot do; hence, when such a code is integrated without code review it results in security vulnerabilities (Clark et al., 2024).

AI is inclined to repeat other decidedly insecure coding patterns derived from the training data it has been fed (Sindhwad et al., 2024). The LLMs learn from different repositories hosted in GitHub, and some of the code may contain insecure or outdated coding practices, such as using weak cryptographic algorithms, insecure API integration, or improper session management (Sindhwad et al., 2024). This issue is especially prevalent when software developers are not assiduous in reviewing the code generated by the AI as they tend to automatically believe that the AI-generated code is safe or secure (Idrisov & Schlippe, 2024). Currently, there is no guarantee that insecure patterns are not spread by AI across the various projects causing a general insecurity throughout the systems incorporating AI-generated code (Sindhwad et al., 2024).

Other security vulnerabilities in AI-generated code:

- Missing security features: This is so because some securities such as input validation, encryption and authentication may not be put in place if the code is written by AI (Żywiołek, 2024).
- Lack of comprehensive testing: AI can build self-generated code on its own and compile it as well, most probably none of which was subjected to tests for security vulnerabilities (Alqaradaghi & Kozsik, 2024).
- Dependency on external libraries: AI-generated code requires references from libraries that may be out of the system, and this can be disastrous since it may use unsuitable and insecure versions (Imtiaz & Williams, 2023).
- Inadequate error handling: Another problem is that error handling will not be very strict when the application is coded with the help of AI, and these can be used to find more information about the system when an error occurs (Xchain Analytics Ltd, 2023).

- Over-reliance on AI for security decisions: The developers trust created code produced by AI without reviewing it in detail leading to overlooking existing security vulnerabilities (Gillard et al., 2023).
- Inconsistent code quality: When using AI during development, it is capable of developing code with various levels of security as the quality of code obtained from the use of AI might not be of high quality (Yetistiren et al., 2022).
- Lack of regular updates: First, the software code created with the help of AI will not be able to be updated to the contemporary standards of protection against threats and is vulnerable to new forms of threats (Li, K. et al., 2023).

Software developers and organizations who use AI to generate code can experience significant problems concerning security vulnerabilities, such as compromised software integrity (Liu et al., 2024). Significant security vulnerability in a software application can be manipulated by a hacker, which would result in a security breach (Ampel et al., 2020). Data poisoning, the result of security vulnerabilities introduced into LLMs during its training process, affects all LLMs, making them an easy target for malicious attacks that source security vulnerabilities in AI-generated code (Cotroneo et al., 2025). It has been discovered that AI-generated code can introduce security vulnerability due to complex coding tasks where ambiguity exists in the textual request by the user (Tao et al., 2024). Cybersecurity research has discussed the difficulties of defending software projects against sophisticated threats, but the differences of security vulnerabilities between AI-generated code and human-generated code has not been well-researched (Negri-Ribalta et al., 2024).

Statement of the Problem

The problem addressed in this study is generating source code using AI increases the risk of security vulnerabilities stemming from data poisoning, compared to traditional human-generated code. Although AI-generated coding tools have significantly advanced, there remains limited empirical research explicitly comparing the security vulnerabilities between AI-generated and human-generated code (Lertbanjongngam et al., 2022). Additionally, existing cybersecurity frameworks, such as the NIST Risk Management Framework (RMF), offer only general guidance without specifically addressing or adapting to the unique security assessment needs posed by AI-generated code (Negri-Ribalta et al., 2024).

Lately, there has been a noticeable increase in the application of AI technologies, especially automated code generation, which has dramatically influenced the software development process across different programming language platforms, where programs using the LLMs generated code contain security vulnerabilities (Majdinasab et al., 2024). In addition, Hamer et al. (2024) stated that AI-generated code affects the software supply chain, and software developers must balance the benefits and risks. Tosi (2024) noted that ChatGPT was checked for quality verification with human metrics, and most of the time, the quality validations passed, but did have noticeable issues of syntax errors, incompatible types, and missing libraries for functions.

There is limited knowledge with AI-generated code compared to human-generated code introducing security vulnerabilities into software projects (Patel. A. et al., 2024). When using AI-generated code and what these security vulnerabilities consist of, to determine the severity and frequency of a security vulnerability. Making this study essential for software developers and organizations who have adopted LLMs into their software development practice.

Purpose of the Study

The purpose of this quantitative causal-comparative research paper aimed at investigating the security vulnerabilities between AI-generated and human-generated source code. In this study, the analysis of security vulnerabilities are based on the use of GitHub repositories and CodeQL security scans to determine the patterns of the vulnerabilities and suggest the security recommendations to enhance the code safety.

To carry out this research, security scans of a sample of 132 publicly available GitHub repositories was done using CodeQL. This tool has demonstrated promising outcomes in identifying security vulnerability especially in AI-generated code, thus, it is appropriate in identifying trends in software security vulnerabilities. This study offered an insight into whether AI-generated code creates more security weaknesses by comparing the security health of the AI-generated code and human-generated code.

The causal-comparative design was suitable in this study, as it would allow discussing pre-existing groups (AI-generated and human-generated code) without any experimental manipulations. The data was gathered and analyzed in a systematic manner so that valid comparisons to represent the real-world software development practices. The researcher also examined ways in which the theory of Intellectual Capital (ICT) and the current frameworks, including the NIST Risk Management Framework (RMF) and the NIST AI RMF, can be modified to eliminate the risk posed by AI-generated code. The results will help an organization deal with software security vulnerabilities more efficiently.

Introduction to Framework

This research was anchored on the theory of Intellectual Capital (ICT), which holds that organizational value and performance depends on the effective use of human, structural, and

relational capital (Wohlin et al., 2015). In the context of software development, human capital refers to the knowledge and competency of the developers; structural capital refers to processes, structures and tools such as the LLMs, static analyzers and risk management structures and relational capital refer to the trust and confidence of the users and stakeholders in safe, dependable systems (Wohlin et al., 2015). Security vulnerabilities are dangerous to all three dimensions of intellectual capital: it undermines human knowledge, destroys organizational frameworks, and reduces confidence with LLMs (Kamaja et al., 2016; Sallos et al., 2019). To operationalize this viewpoint, the study used the National Institute of Standards and Technology (NIST) Risk Management Framework (RMF) and the NIST Artificial Intelligence Risk Management Framework (AI RMF). These models provided a structured approach to the detection, evaluation and reduction of security vulnerabilities in both AI- and human-generated code. As a result, ICT offered the theoretical background that explains the attention paid to the knowledge assets and trust protection, but the NIST RMF and AI RMF offered the operational framework of security vulnerability assessment and reduction (Ng, 2006; NIST, 2028; NIST, 2024). Collectively, they formed a strong and extensive theoretical foundation of studying the differences between AI- and human-generated code in terms of security vulnerability (Sallos et al., 2019).

The foundation of this study is the National Institute of Standards and Technology (NIST; 2018) risk management framework (RMF) and NIST AI RMF (Piper, 2023). The NIST RMF provided a structured approach to identifying, assessing, and mitigating security vulnerabilities in software systems. This study applied the RMF to compare AI-generated and human-generated code, focusing on security vulnerability detection using CodeQL. Through the RMF and AI RMF, a holistic methodology is presented for addressing risks in the entire software development

lifecycle, making it especially suitable for studying the security vulnerabilities introduced by AI-generated code and human-generated code in public GitHub repositories. The NIST RMF helps organizations control system security and privacy risks through its structured processes, but the NIST AI RMF specifically handles AI system vulnerabilities such as bias and trust in addition to explainability; both frameworks apply to AI-generated code through their secure development guidance and risk assessment methods suitable for AI-driven software (NIST, 2018; Tabassi, 2023). The NIST Risk Management Framework (RMF) outlined three steps relevant to this study: risk identification, risk assessment, and risk mitigation (NIST, 2018). In this context, risk identification involves pinpointing security vulnerabilities in AI-generated and human-generated source code. Risk assessment evaluated the likelihood and potential impact of these security vulnerabilities, while risk mitigation involves strategies to reduce or eliminate security vulnerabilities before code deployment (NIST, 2018).

Based on the idea of proactive and systematic risk management, the RMF assumed that the possibility of a security breach can be minimized by the National Institute of Standards and Technology (NIST, 2018). When a security vulnerability is found, severity of the risks and the recommendation to mitigate the risks to the security of software projects was influenced by NIST AI RMF (Piper, 2023). The focus on proactive risk management aligns well with the objectives of this study, which explored whether AI-generated code increases the security vulnerability probability and identify which project characteristics could raise or lower the risk of those security vulnerabilities.

Derived from National Institute of Standards and Technology (NIST, 2018) RMF's structured and systematic approach to dealing with security vulnerabilities in AI-generated code (Piper, 2023), the problem statement was developed. This informed the production of the

purpose statement, which is to investigate the casual-comparative between AI-generated code and security vulnerabilities and human-generated code, where the research questions were shaped by the proactive use of NIST RMF. The NIST (2018) RMF is aligned with the problem statement, purpose statement, and proposed research question(s), as well as the objective of proposing a proactive approach to managing security risk on AI-generated code. Secure coding practices recommended by NIST AI RMF and automated tools to detect security vulnerabilities prior to exporting code in production environments will be adopted by mitigation strategies within the already existing context of software development (Tabassi, 2023).

Introduction to Research Methodology and Design

This study employed a quantitative method suitable for this type of research since systematic collection and analysis of numerical data determined the differences between security vulnerabilities in AI-generated code compared to human-generated code and the overall security posture of software projects. Calder (2018) provides extensive guidelines on how to design and conduct a study using the quantitative method for this study. The independent variables (security vulnerabilities in human-generated code and AI-generated code caused by data poisoning as stated by Majdinasab et al. (2024)) and the dependent variable (overall security posture of software projects). This design guided the researcher in identifying well-suited research questions without manipulating the variables. Cohen et al. (2013) provided explanations and examples of how to apply these causal relationship techniques, which strengthens this study by exploring the potential security vulnerabilities of AI-generated code and the impact on the security posture of software projects.

Automated code analyzers was used to assert the correctness of identified security vulnerabilities in code, which is produced by an AI algorithm. Enumeration tools for code

analysis automatically analyze the code to find or categorize security weaknesses. Neumann et al. (2010) explained the role using the enumerations tools in the analysis of security vulnerabilities in software projects, since past failures and research are typically ignored. Security posture metrics where issue reports and security patches were utilized to determine the general security status of the projects. These metrics offered a clear overview of how a project is likely to deal with the threats concerning security issues, a factor that is central to the study's goals (Tabassi, 2023). These instruments are selected based on the dependability and credibility of instruments used to quantify security threats and security status in an organization so that the data collected contains no inherent flaws and bears relevance to the research.

A causal-comparative design is appropriate for this study as it allows for a comparison of security vulnerabilities in AI-generated and human-generated code without manipulating the independent variable. Unlike experimental studies, which require controlled environments, this approach enables analysis of real-world GitHub repositories, which are not controlled in a dynamic environment. Qualitative methods were considered; however, it would be rejected due to the collecting and analyzing numeric (interval or ratio) data and qualitative methodology only works for nominal (categorical). T-tests and chi-square tests were used in this study to evaluate the casual-comparison between independent variables referring to security vulnerabilities within AI-generated code together with human-generated code and the dependent variable that represents software projects' overall security posture. It is noteworthy that this analysis is crucial for finding some relationships that may be helpful to improve security strategies concerning AI-written code in the future (Tabassi, 2023).

This study design entailed choosing a purposive sample of 67 repositories (group one) that used AI-generated code and 67 repositories (group two) that used human-generated code to

ensure statistical power for conducting t-tests and chi-square tests out of the entire population of ~154,600,000 repositories determined using a priori power analysis using G*Power 3.1.9.7. To find and quantify these relations, the researcher used t-tests and chi-square tests, to discover associations between independent variables (e.g. type of code generation (AI versus human), the programming language used or the project size) and the dependent variable (the presence, and severity, of security vulnerabilities).

Research Questions

The research questions and their respective hypotheses were used to determine whether the code generated using AI or human-generated code injects security vulnerabilities into the source code. The research questions asked, along with the hypotheses testing were used to determine whether the research questions are significant in this study.

RQ1

What is the statistical significant difference in the mean CVSS scores of security vulnerabilities between AI-generated code and human-generated code when analyzed with CodeQL?

RQ1a

What types of security vulnerabilities may be more common in AI-generated code than in human-generated code?

RQ1b

What security vulnerabilities may be common to both AI-generated code and human-generated code?

Hypotheses

$H1_0$

There is no significant difference in the mean CVSS scores between the security vulnerabilities in AI-generated code than that of human-generated code.

$H1_a$

AI-generated code exhibit significantly different mean CVSS scores security vulnerabilities compared to human-generated code.

$H1a_0$

There is no significant difference in the security vulnerabilities between AI-generated and human-generated code.

$H1a_a$

AI-generated code will exhibit significantly more security vulnerabilities than human-generated code when analyzed using CodeQL.

$H1b_0$

There is no significant commonality of security vulnerabilities (e.g., input validation errors) of both AI- and human-generated code.

$H1b_a$

Specific security vulnerabilities (e.g., input validation errors) will be more prevalent in AI-generated code compared to human-generated code.

Significance of the Study

The study for security vulnerabilities in AI-generated code versus human-generated code is of academic and industrial importance, owing to the empirical evidence it provided regarding lack of security in AI-generated code compared to human-generated code. As creating source

code through LLMs becomes a more common approach, it is important to understand the security vulnerabilities associated with the said source code. In this regard, the findings of this research may inform adjustments to existing AI governance frameworks (e.g., the NIST AI Risk Management Framework (AI RMF)) in addressing the specific security vulnerabilities posed by generative models, and in certain respects where existing guidance falls short. Results from the study could be used to guide the enhancing rule coverage of static analysis tools, like CodeQL, for detecting patterns of security vulnerability in AI-generated code. This research may be used by industry platforms such as GitHub to update security policies, improve code review practices, and encourage a safer codebase among software developers. Lastly, this work could possibly shed some light on areas for future academic research and policy making to think about safety in current AI-generated code and offer scoped objectives for safe development of software in the future of generative AI.

Definitions of Key Terms

AI-Generated Code

AI-generated code is software code that is automatically generated or suggested by using artificial intelligent tools or systems, such as GitHub Copilot. With this help, it generates code snippets, automates routine coding tasks or provides suggestions based on input from users (Kapakos & Fulk, 2024).

CodeQL

A security analysis tool available in GitHub to scan source code for security vulnerabilities, where the source code is viewed as data, as in a database, and the user writes custom queries to uncover potential security vulnerabilities (Fröberg, 2023).

Data Poisoning

Altering the training data of an artificial intelligence model with malicious data so that faulty responses or outputs are generated (Aghakhani et al., 2024).

GitHub Repository

It is an online software source code management system and code storage platform that could be public or private, to provide collaborative software project development (Hou et al., 2023).

Human-Generated Code

Human-generated code is where software projects are created by professional software engineers (Idrisov & Schlippe, 2024).

Large Language Models (LLMs)

Large Language Models (LLMs) are used to automate programming tasks through the understanding of natural language to create code snippets and help software engineers (Hou et al., 2023).

Open-Source

Software is freely available for anyone to use, modify, and distribute, which is what is known as open-source software (Liu et al., 2024).

Risk Assessment

Analyzing the AI-generated code and human-generated code for security vulnerabilities and determining the potential security vulnerabilities that might occur during software development National Institute of Standards and Technology (NIST, 2018).

Risk Identification

Identifying risk is the process of identifying and documenting potential security vulnerabilities of a software system (Sllame et al., 2024).

Risk Mitigation

Before security vulnerabilities occur, strategies are put in place to potentially prevent security vulnerabilities from occurring in the software National Institute of Standards and Technology (NIST, 2018).

Security Posture

The term security posture can refer to a system's overall strength in resisting security threats (Edkrantz et al., 2015). For instance, a system with tools like multi factor authentication and constant security checks is likely to resist attacks.

Security Vulnerability

There is a flaw or weakness in a system’s software that could lead to damage or disruption of the system’s function, or unauthorized access to the system by attackers, technically known as hackers (Edkrantz et al., 2015).

Summary

This chapter introduced the relevance of evaluating security vulnerabilities in AI-generated versus human-generated code, grounded in the Intellectual Capital (ICT) theory and NIST RMF and AI RMF frameworks. The problem and purpose statements, along with the research questions, set the foundation for a quantitative causal-comparative design using CodeQL to scan open-source GitHub repositories. This chapter also introduced the potential impact on AI security governance and industry practices. Chapter 2 builds on this by exploring current research on AI coding tools, security vulnerability patterns, and static analysis

techniques. The literature review will help identify what is known, what remains unclear, and how this study addresses existing gaps.

Chapter 2: Literature Review

The purpose of this quantitative causal-comparative research paper aimed at investigating the security vulnerabilities between AI-generated and human-generated source code. The problem addressed in this study is generating source code using AI increases the risk of security vulnerabilities stemming from data poisoning, compared to traditional human-generated code. This chapter begins with an examination of the theoretical framework that supports this study, including its background and current, relevant usage in AI-generated code. The chapter continues with a review of the pertinent literature as related to the security vulnerabilities of AI-generated code.

As part of the literature review the following will be examined in detail: Understanding Software Development; Understanding AI; Understanding AI in Software Development (SD); Comparing Human-Generated Code with AI-Generated Code.

Relevant literature review appears in this chapter for providing theoretical support to the studies research method. The theoretical framework section forms the first part of the review which merges AI Risk Management Framework (RMF) with the National Institute of Standards and Technology (NIST) RMF. Previous researchers have used these frameworks to integrate risk mitigation strategies for the betterment of humankind (Lahiri & Saltz, 2024; Swaminathan & Danks, 2024). The section progresses to a discussion about AI-generated code which evaluates its software development function and reviews its benefits alongside its security consequences. Then human-generated code is evaluated in its software development function to review its security vulnerabilities that continue to occur in software projects, along with its consequences. Security vulnerabilities from both AI-generated code and human-generated code receive a comparison treatment in the subsequent discussion. This chapter provides a review of static

analysis tools where it examines how automated security scanners, particularly CodeQL can identify security vulnerabilities in software projects. The chapter contains through identification of research gaps that demonstrate the necessity for more work in AI-generated code security vulnerability awareness.

Search terms and their combinations are listed in Appendix A.

The researcher used EBSCO Databases and Google Scholar to locate peer-reviewed articles in addition to conference proceedings and technical reports about AI-generated code and human-generated code security. The researcher performed literature research through keywords which combined topics about AI-generated code and human-generated code security vulnerabilities with LLMs and security vulnerabilities and CodeQL static analysis tool and comparative security vulnerabilities in software development. The main restriction was to use materials from 2024 and 2025, but the researcher included essential references published earlier than 2024. The medium-long duration for books to get published usually leads to obsolete content so they were typically eliminated from consideration. In some cases the researcher had to utilize an internet platform: https://www.cve.org to get the latest information of publicly known security vulnerabilities in software projects to help guide the researcher on what security vulnerabilities tend to be the most frequent in code; however, it does not state whether the software project was developed using AI-generated code, human-generated code or hybrid code. The examined studies present detailed information about both the security vulnerabilities facing AI-generated code and ways to evaluate these risks through available methods.

Theoretical Framework

Security vulnerabilities concerning AI-generated code must have a robust theoretical framework to effectively understand, classify and resolve relevant emerging threats. The main

theoretical framework of this research is Intellectual Capital Theory (ICT) that explains the way organizations generate and maintain value because of three types of intangible resources: human capital, structural capital, and relational capital. In software development terms, human capital refers to expertise and imagination of software developers; structural capital entails the mechanisms, frameworks, and tools that facilitate safe development (such as AI-generated code and CodeQL the framework of RMF offered by NIST); relational capital refers to trust and confidence end-users, clients and stakeholders place in a software system in terms of its security and reliability (Sallos et al., 2019; Wohlin et al., 2015). Security vulnerabilities that are added to code, be it through human error or AI-generated patterns, undermine all the three types of intellectual capital, by undermining expertise, weakening organizational structures, and reducing trust (Ng, 2006). As a result, the issue of the security vulnerabilities of AI-generated code as compared to the human-generated code is identified not only as a technical matter but also as perceived as an intellectual capital issue (Ng, 2006). Due to compromised training data, AI-generated code is likely to replicate unsafe trends, which consequently jeopardizes organizational structural capital by introducing security vulnerabilities into software projects (Sallos et al., 2019). Likewise, the code produced by humans might be an indicator of the shortcomings of human capital like cognitive overload, lack of knowledge or imperfect judgment (Kamaja et al., 2016). The two situations threaten relational capital, where weaknesses cause a decrease in trust in the organizational systems and may destroy the relationship with long-term stakeholders. ICT, therefore, explains the logic of this research: to safeguard intellectual capital, security vulnerabilities in different modes of code production need to be identified, evaluated, and reduced (Sallos et al., 2019).

Other operationalization of ICT in this study includes the use of National Institute of Standards and Technology (NIST) Risk Management Framework (RMF) and NIST Artificial Intelligence Risk Management Framework (AI RMF). Compared to the AI RMF, the NIST RMF provides a systematic approach to risk identification, evaluation, and reduction across the software lifecycle, and the AI RMF applies analogous values to AI-generated code issues like explainability, bias and trustworthiness (Wohlin et al., 2015). Collectively, these models provide the technical and procedural means with which to protect intellectual capital in modern software development (Sallos et al., 2019). By way of illustration, the risk identification and mitigation measures based on NIST are directly related to the capacity to preserve structural capital, and the focus on transparency and trustworthiness offered by the AI RMF on relational capital. The combination of the ICT and these risk management frameworks produce a two-layered theoretical basis. ICT emphasizes the intangible resources that organizations need to retain to ensure a competitive edge, but NIST RMF and AI RMF offer more tangible means of securing such resources in practice (NIST, 2018; NIST, 2024; Wohlin et al., 2015). This synthesis allows the study not only to evaluate the difference between AI- and human-generated code security vulnerabilities but also to see the consequences of the security vulnerabilities on the organizational knowledge, structure, and trust in general. The fact that the analysis is anchored by the ICT highlights the fact that the notions of cybersecurity cannot exist without the management of the intellectual capital, thus making the ICT theory the primary tool in the understanding of the matter of secure code in organizational sustainability (Sallos et al., 2019).

The NIST RMF is a comprehensive, repeatable, flexible process for managing security and privacy risks of AI-generated code (NIST, 2018). The NIST RMF is appropriate for analyzing AI-generated security vulnerabilities because it systematically manages risks across

the entire software lifecycle, addressing unique challenges like data poisoning and insecure coding patterns specific to AI-generated software. It integrates security, privacy and supply chain risk management activities into the software development lifecycle to enable identification, assessment and mitigation of security vulnerability risks (NIST, 2018). NIST RMF is intended to be tailorable, so that the application is appropriate to an organization's specific needs and requirements (NIST, 2018).

While the NIST RMF provides a robust structure for managing security vulnerabilities, the design is for traditional IT systems rather than AI-specific challenges. In contrast, the NIST AI RMF expands this foundation by addressing emerging concerns such as AI bias, explainability, and trustworthiness. The flexibility of the AI RMF allows it to accommodate the rapid evolution of LLMs and their unique vulnerability profiles. Compared to ISO 27001, which offers a generalized and adaptable cybersecurity approach, the NIST RMFs emphasize rigid security control baselines that may be better suited for structured security vulnerability analysis. The EU AI Act provides a legal framework for AI governance and risk classification; it is regulatory and less focused on the technical detection and mitigation of security vulnerabilities within AI-generated source code. This difference justifies the application of the NIST RMF and AI RMF, where organized, technical, and cybersecurity assessment methodologies are necessary. The NIST (2018) showed how to assess and mitigate risks in software development projects by ensuring security controls are put into place and effectively implemented and maintained throughout the software's lifecycle, especially in pertaining to AI-generated code (Chowdhury et al., 2023).

This study also draws upon the theoretical base of the National Institute of Standards and Technology's Artificial Intelligence Risk Management Framework (NIST AI RMF) as developed

by Tabassi (2023) and NIST (2024). By using this framework, to assess LLMs in a structured and organized manner, it was possible to quantitatively estimate security vulnerabilities in AI-generated code compared to human-generated code. Although the NIST AI RMF provides valuable guidelines for managing general AI risks such as bias and explainability, it currently lacks specific, actionable recommendations tailored to identifying and mitigating security vulnerabilities unique to AI-generated source code. The increased complexity and interweaving of LLMs into diverse organizations, the NIST AI RMF was created to assist organizations in a comprehensive approach to recognizing, evaluating, and addressing security vulnerabilities particular to the employment of LLMs (NIST, 2024). Piper (2023) explained how this framework is timely and important as the number of solutions driven by LLMs has increased exponentially in every domain, and LLMs presents a novel risk profile, requiring its own dedicated guidance. NIST AI RMF is particularly designed within software engineering as well as cybersecurity to identify the LLM specific security vulnerabilities (NIST, 2024).

Tabassi (2023) theorized that these categories of identifications, assessments, management, and governance are essential for the robust conceptualization of the security vulnerabilities in AI-generated code. The NIST AI RMF's advantage is in emphasizing systematic, cyclic risk identification, assessment, mitigation, and continuous monitoring processes, which lends researchers a systematic and quantitative approach for making risk calculations concerning security vulnerabilities on addressing the causal-comparative research questions effectively (NIST, 2024). AI-generated code presents greater security vulnerabilities compared to human-generated code, the NIST AI RMF offers a theoretical basis for a holistic approach to these security vulnerabilities. Previous research mentioned by Ambati et al. (2024), Fu et al. (2024), and Cotroneo et al. (2025) stated how AI-specific phenomena, such as LLM

data poisoning, data contamination, and context misunderstanding, produce unique security vulnerabilities that get introduced into software projects.

The NIST AI RMF states the need to perform security vulnerability assessments continuously, across an LLMs full life cycle, from design, training, to deployment, and finally, post-deployment (NIST, 2024). Calder (2018), Roy (2020), and Lahiri and Saltz (2024) stated how cybersecurity best practices should skip the reactive and isolated assessments and instead adopt a proactive lifecycle integrated risk management practice. Alenezi and Akour (2025), Bannon and Laplante (2024), and Balogh et al. (2024) explicitly asked for structured theoretical frameworks such as the NIST AI RMF to find a route through the complexity of incorporating LLMs in software development. The NIST AI RMF also includes key elements needed to ensure transparency, explainability, accountability, and trustworthiness of an LLM, all vital elements in mitigating security vulnerabilities in AI-generated code. According to Milovidov (2024) and Żywiołek (2024), trust and transparency remain key issues in the deployment of AI-generated code that will be in direct contact with cybersecurity postures. However, the security vulnerabilities in the underlying AI-generated code are often rooted in decision making of the underlying LLMs, which is usually opaque, making security vulnerability identification and remediation transparent a very challenging problem (Cotroneo et al., 2025).

Employing both the NIST RMF and the NIST AI RMF together creates a dual-layered foundation for security vulnerabilities evaluation. The NIST RMF provided a generalized process for identifying, assessing, and mitigating security vulnerabilities across all types of software projects. However, the NIST AI RMF introduces principles specifically targeted toward the unique security vulnerabilities of AI-generated outputs, including systemic bias, opaque decision-making, and training-data security vulnerabilities. This helps to ensure that

AI-generated security vulnerabilities are never overlooked in favor of LLMs, a limitation that could otherwise compromise the accuracy of security assessments in AI-generated code. Using both frameworks synergistically supports the goal of comprehensively comparing the security vulnerabilities between AI- and human-generated source code. While the NIST RMF (NIST, 2018) and NIST AI RMF (NIST, 2024) frameworks provide structured approaches to risk management, there remains a lack of empirical studies applying these frameworks specifically to assess and compare security vulnerabilities in AI-generated and human-generated codebases (Fu et al., 2024; Sindhwad et al., 2024).

Figure 1

NIST AI RMF Core Principles

Note: There are four core principles of the NIST AI RMF (Tabassi, 2023).

Table 1

Contrast of Theories

Framework/Theory	Focus Area	Approach	Strengths	Contrasts NIST RMF & NIST AI RMF	Reference(s)
NIST RMF	Cybersecurity risk management for IT systems	Compliance-driven, security controls-based	Well-structured, and widely adopted in the U.S. government along with industry mandates security controls	N/A	(NIST, 2018)
NIST AI RMF	AI risk management and governance	Voluntary risk-based framework for AI security, fairness, and trustworthiness	Encourages ethical AI development, integrates explainability and security	N/A	(Tabassi, 2023)
ISO 27001	Information security management	Risk-based, adaptable	Globally recognized, flexible for different industries	Unlike NIST RMF, it does not require predefined security controls, making it more adaptable but less structured	(Roy, 2020; Bertocchi & Piamonte, 2023)
FAIR (Factor Analysis of Information Risk)	Quantitative risk assessment	Financial impact-based risk modeling	Enables organizations to prioritize risks using cost-based analysis	Unlike NIST RMF, it focuses on financial risk quantification instead of compliance-driven security measures	(Tabassi, 2023; Pham et al., 2025)
COBIT (Control Objectives for Information and Related Technologies)	IT governance and management	Business and process-oriented IT control framework	Aligns IT security with business objectives	Unlike NIST RMF, it emphasizes IT governance rather than direct security risk management	(Salihu & Dervishi, 2024; NIST, 2018)
EU AI Act	AI governance and regulatory compliance	Legally binding risk-based AI classification	Enforces accountability, legally regulates AI risks	Unlike NIST AI RMF, it is legally enforceable, classifying AI risks into tiers and imposing penalties	(Hadwick, 2024; Burlacu & Luta, 2023)

Note: Compares and contrasts each framework/theory

This framework underpins the study's goal of assessing the security posture of AI-generated code using tools like CodeQL, aligning with the need to fill the gap in empirical comparisons between AI- and human-generated code security vulnerabilities.

Understanding Software Development

Software development (SD) is a computer science practice where a group of professional engineers conceive, specify, design, program, document, test, and maintain software projects using methodologies to arrive at desired solutions (Fu et al., 2024). As Alqaradaghi and Kozsik (2024) emphasized, using the software development life cycle (SDLC) provides a meaningful approach for guiding that software projects are developed according to a project plan and creates high quality software throughout its lifecycle. However, Alenezi and Akour (2025) said the vast array of programming languages plays a foundational role in SD by providing software developers with tools to create proper software solutions in the SDLC. Significantly, only a few of the programming languages, like Python, Java, C++, and JavaScript, enable software developers to implement software solutions to be designed for specific user needs (Sindhwad et al., 2024). Each programming language brings different benefits and challenges due to limitations. While Alqaradaghi and Kozsik (2024) emphasized that SDLC offers structured development to reduce security vulnerabilities, Cotroneo et al. (2025) argued that the choice of programming language introduces its own risks, regardless of lifecycle adherence, due to language-specific syntax and memory management issues. This tension suggests that secure SD requires both rigorous lifecycle adherence and informed language selection, a position echoed by Sindhwad et al. (2024), who recommend aligning SD practices with the threat model associated with the selected technology stack.

Human software developers are active and make many contributions throughout the SDLC, spanning from conceptualization to maintenance of a software project (Sindhwad et al., 2024), similarly, in another study Fu et al. (2024) stated software developers constantly make complex and creative decisions at every phase in the SDLC, by understanding and interpreting

requirements, designing sophisticated software architectures, and generating code to fulfill functional and non-functional requirements. Unlike automated tools, software developers bring context awareness, adaptability, and creativity to SD, which are critical factors for dynamic environments where requirements change constantly or mitigating unforeseen issues that usually arise that software developers must overcome (Cotroneo et al., 2025); whereas Alqaradaghi and Kozsik (2024) stated how different programming languages assist software developers, it is human judgment that ultimately interprets test results, prioritizes bug fixes, and weighs trade-offs between performance, security, and usability. Therefore, human involvement is irreplaceable, in contrast, while Cotroneo et al. (2025) highlight human flexibility as vital for adaptive development, Gilstrap et al. (2024) point out that cognitive fatigue and human error often lead to overlooked security vulnerabilities, suggesting that human decision-making is both an asset and a liability within SD.

Human-generated code in software projects typically contains critical security vulnerabilities caused by common developer mistakes (Anu et al., 2020; Patel. D. et al., 2023). For example, the infamous Heartbleed bug in OpenSSL resulted from a simple bounds-checking error, allowing attackers to extract sensitive information from server memory; therefore, LibreSSL came to replace this security vulnerability due to old legacy code that came to its end of life (personal conversation with D. Palmer, April 27, 2025). Similarly, software developers who failed to properly apply data cleansing for user inputs have repeatedly enabled SQL injection attacks, notably contributing to large-scale data breaches such as the 2007 breach of TJX Companies (personal conversation with D. Palmer, April 27, 2025). These real-world examples demonstrate that experienced software developers inadvertently introduce critical security vulnerabilities during the SDLC (Li, J. et al., 2024; Sindhwad et al., 2024). While such

human errors stem from individual oversight or time constraints (Fu et al., 2024), security vulnerabilities in AI-generated code arise from data poisoning and inconsistent contextual understanding (Żywiołek, 2024; Cotroneo et al., 2025), indicating that human errors are sporadic and contextual, whereas AI-generated security vulnerabilities can be systemic and pervasive.

There are various software professional roles to monitor human-generated code, including software developers themselves, quality assurance testers, security analysts, and peer reviewers (Fu et al., 2024). Organizations that implement code reviews and static analysis scanning tools, such as CodeQL, and execute continuous integration pipelines ensure the quality of software projects and detect security vulnerabilities before issues usually arise (Sindhwad et al., 2024). Similarly, despite these software practices, software developers usually fall short on preventing issues. This aligns with Gilstrap et al. (2024), who argue that production-level pressures impede thorough security validation. However, Ambati et al. (2024) contend that AI-generated code bypasses these human limitations but introduces its own risks due to biased training data, highlighting a trade-off between speed and contextual security awareness. Due to oversight, cognitive biases, and underestimation of security vulnerabilities frequently lead to issues persisting in production environments (Gilstrap et al., 2024; Patel. D. et al., 2023). Formal audits and testing by third-party security experts further complement internal reviews (Nofal et al., 2025); however, as Gilstrap et al. (2024) noted, even these practices can miss deeply embedded security vulnerabilities due to inherent complexity or time constraints.

Human-generated code remains highly susceptible to security vulnerabilities due to cognitive failures, misjudgment, and flawed assumptions during SD (Fu et al., 2024; Sindhwad et al., 2024), where other studies have stated how common security vulnerabilities include buffer overflows, SQL injection attacks, improper error handling, and insecure data storage (Cotroneo

et al., 2025; Qadir, 2023), whereas these security vulnerabilities often propagate from failing to anticipate edge cases, lapses in attention to detail, and misapplication of SD best practices (Cotroneo et al., 2025). Research by Fu et al. (2024) revealed that software developers' cognition failures, such as memory overload, task-switching fatigue, and confirmation bias, increase these security vulnerabilities. Moreover, the prevalence of legacy software systems increases these security vulnerabilities, as older software systems typically reflect outdated security paradigms and may lack modern protections (Gilstrap et al., 2024). The context in which software developers operate also influences the security vulnerability rates, such as tight deadlines, ambiguous requirements, and organizational pressures can impede a software developer's decision-making, leading to shortcuts that introduce hidden security vulnerabilities (Alqaradaghi & Kozsik, 2024). Security vulnerabilities in human-generated code require technical solutions to be utilized, and cultural changes in organizations need to prioritize secure coding practices and allocate sufficient resources for thorough testing and review before deploying to production (Alqaradaghi & Kozsik, 2024; Gilstrap et al., 2024).

Understanding AI

AI is an important popular tool for processing, analyzing, and using information to make decisions across organizations, because at its core, LLMs are created to execute tasks that typically require human intelligence, such as understanding language, recognizing patterns, solving problems, and making decisions (Żywiołek, 2024). Unlike traditional SDs that follow explicit instructions, Ambati et al. (2024) stated LLMs adapt and learn from data, enabling them to handle repetitive and straightforward tasks in evolving environments. Similarly, machine learning, a key subset of AI, allows LLMs to improve their performance over time without being manually re-engineered (Fu et al., 2024). Ambati et al. (2024) highlight efficiency gains from AI

automating repetitive tasks, yet Cotroneo et al. (2025) caution these benefits frequently include inherited security vulnerabilities from training data, indicating an important efficiency-security trade-off. AI has the potential to augment human abilities by rapidly processing large datasets, identifying hidden patterns, and proposing robust solutions that might otherwise go undetected by human analysts (Cheatham et al., 2019). However, Ambati et al. (2024) stated the same capabilities that allow AI to rapidly generate solutions can also propagate security vulnerabilities if insecure patterns are learned during training and without context awareness, LLMs may replicate flawed logic or security vulnerabilities, leading to systemic risks within deployed software projects (Bannon & Laplante, 2024; Sindhwad et al., 2024)

Implementing AI-generated code into software projects has already resulted in tangible security vulnerabilities (Asare et al., 2023), similarly GitHub Copilot, for instance, was found to replicate insecure coding patterns approximately 40% of the time, based on a study by Asare et al. (2023); in contrast, Pearce et al. (2025) believed using GitHub's Copilot suggested that the code is vulnerable to command injection attacks and inadequate input validation. Another example is the use of generative AI tools in automating API development (Alenezi & Akour, 2025; Bannon & Laplante, 2024). Similarly, other studies have said there is a lack of secure authentication flows being discovered, making the resulting software projects vulnerable to unauthorized access (Gilstrap et al., 2024; Kotb et al., 2025). These real-world observations highlight the urgency of systematically studying AI-generated code security vulnerabilities rather than assuming that AI-generated outputs are inherently secure (Sindhwad et al., 2024).

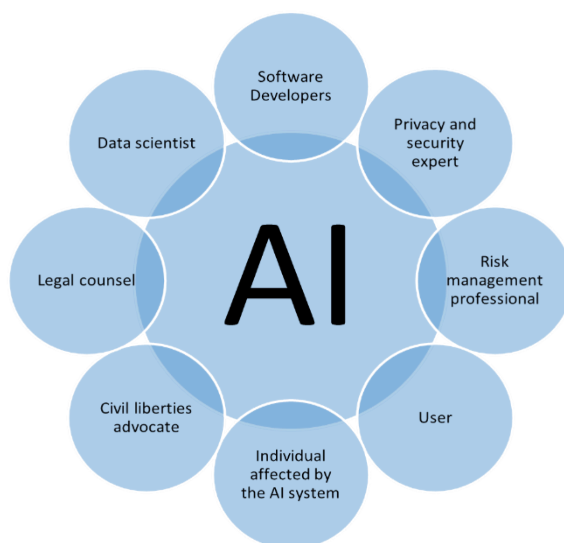
There is a growing gap between organizations that understand AI and those who do not; for example, organizations with a strong technical infrastructure and cybersecurity frameworks are more likely to successfully integrate AI into their operations compared to organizations that

lack these resources (Benbya et al., 2020), which leads to unequal adoption rates, exacerbating existing inequalities in technological capacity among organizations across the globe. Another important aspect is how software developers collaborate with LLMs; rather than replacing software developers entirely, LLMs often function best when they increase human decision-making (Balogh et al., 2024; Bannon & Laplante, 2024; Żywiołek, 2024), where LLMs can be a powerful tool in healthcare, finance, and security, but human oversight remains crucial for interpreting results and making sound judgments that LLMs might miss (Bannon & Laplante, 2024; Dincă et al., 2024; Żywiołek, 2024).

The adoption of AI-generated code extends across multiple organizations and among a diverse range of stakeholders, so security analysis needs increased attention (Gilstrap et al., 2024; Kotb et al., 2025; McFee, 2024). Security in the AI-generated code boosts operational performance and efficiency, but AI-generated code still creates security vulnerabilities that affect organizations (Kotb et al., 2025; Bannon & Laplante, 2024). With organizations adopting automation in the software development processes by using tools such as GitHub's Copilot have sparked interest among software developers of all skill sets and organizations who strive for rapid delivery of software projects (Bannon & Laplante, 2024; Gilstrap et al., 2024).

Figure 2

Diverse Range of AI Users and Stakeholders



Note: Different industry users and stakeholders of AI technology (Gao et al., 2023).

In the broader societal view, the advancement of LLMs brings both promise and risk (Kirk et al., 2024); however, on the one hand, AI can accelerate progress toward sustainable development goals (SDGs) by optimizing resource usage, minimizing technical debt, and improving access to education and healthcare (Żywiołek, 2024). Nevertheless, LLMs tend to reinforce biases, privacy violations, or producing security vulnerabilities into software projects (Kirk et al., 2024); therefore, understanding AI requires acknowledging both its capabilities and limitations it is not enough to be amazed at its speed and predictive power; software developers, policymakers, and users must apply critical thinking by applying questions of trust, security, ethics, and fairness (Kirk et al., 2024; NIST 2024). As AI evolves, ensuring it supports human well-being rather than undermining it will be a central challenge for researchers, stakeholders, and leaders across all organizations (Żywiołek, 2024).

A real-world example highlighting the security vulnerabilities of AI-generated code involves GitHub's Copilot (Pearce et al., 2025). Pearce et al. (2025) found that GitHub's Copilot often generated insecure code patterns vulnerable to known attacks, such as the common SQL injection risk, specifically, one test demonstrated that GitHub's Copilot suggested code snippets handling user inputs without proper sanitization, directly leading to exploitable security vulnerabilities. Another case study (Asare et al., 2023) examined AI-generated code used in a cryptocurrency exchange project where improper cryptographic key handling created exploitable weaknesses, resulting in significant financial losses, where these incidents illustrate how AI-generated code introduces critical security vulnerabilities into production environments (Ampel et al., 2020; Pearce et al., 2025). While AI capabilities and associated security vulnerabilities are increasingly acknowledged (Alenezi & Akour, 2025; Bannon & Laplante, 2024), there is a gap in systematic research directly comparing AI-generated security vulnerabilities to those found in human-generated code using quantitative causal-comparative designs (Sindhwad et al., 2024).

Understanding AI in Software Development (SD)

AI has increasingly become a transformative force in SD, automating repetitive tasks that traditionally require significant human time (Ambati et al., 2024; Majdinasab et al., 2024) as other studies have pointed out, AI's significant contributions to SD in AI-generated code have accelerated development timelines and reduced the manual labor of writing repetitive functions, which are dubbed boilerplate code (Alenezi & Akour, 2025; Bannon & Laplante, 2024). Large arrays of datasets help to train LLMs, including billions of lines of human-generated source code, which enable LLMs to generate functional code snippets, automate code review processes, identify patterns of bugs, and even suggest optimized programming algorithms (Cotroneo et al.,

2025; Sindhwad et al., 2024) and as Fu et al. (2024) stated AI also assists in automating testing procedures through tools that can create and execute unit tests autonomously, ensuring better coverage and faster feedback cycles; however, the integration of AI into SD is not without concerns; even though AI can automate code generation, it often replicates biases or security vulnerabilities implanted within the training data, known as data poisoning, raising significant questions about the trustworthiness of AI-assisted development practices (Żywiołek, 2024).

Recent academic literature showcases recent appreciation of the fast-growing adoption of LLMs to protect the critical national infrastructures (CNIs) and the potential and limitations of code generation using LLMs for high-stakes domains (Yigit et al., 2025; Żywiołek, 2024). In contrast, LLMs are being leveraged in the real world (e.g., the energy grid or the water treatment plant) to actively protect critical infrastructure, and the number of attacks on such systems have nearly 1,308 weekly global incidents paired with the adoption of LLMs exceeding vetting and reviews (Yigit et al., 2025), which is consistent with Taghavi et al. (2025) who observed that LLMs such as ChatGPT have a large potential to discover buffer overflow defects on a wide set of codebases, however the coverage of their datasets might be lacking fine granularity or outdated software constructs which could result in accepting weakness in security assurance (Cotroneo et al., 2025; Taghavi et al., 2025). These findings suggest an examination gap in which deployed LLM in real world ways is walking at a furious pace, yet strong validation methodologies designed to be intelligent about deployment are happening at a similar pace, inducing possibilities of improved productivity and automation of generated code exposing an organization to previously unnoticed security vulnerabilities (Ambati et al., 2024).

LLMs are inherently incapable of understanding the software requirements' intricacies as well as the holism of application security, making them vulnerable to security attacks (Hussain et

al., 2024; Idrisov & Schlippe, 2024); however, according to Idrisov and Schlippe (2024), LLMs can generate syntactic and appear to be functioning pieces of code, but they are not sensitive enough to contextual factors necessary to produce secure software. For instance, the LLM might be as good as performing data processing, it just forgot about authentication check or in something bigger exploited the cryptographic function etc., but the security holes to exploit can be found (Hussain et al., 2024). This point was furthered by Ogundare et al. (2022) who added that there is a shortfall in a LLM when it comes to its application of appropriate security primitives and constraints in terms of cryptography, for example, for keys and data flow sensitivity within software applications. Failure of the developers to pay attention to these boundaries is going to lead them to building software products that have critical security vulnerabilities (Liu et al., 2024).

AI collaborates with software development processes at nearly every stage of the SDLC; during the design phase, LLMs assist in automatically suggesting architectural patterns and design strategies based on the business rules (Patel. D. et al., 2023). In contrast, the coding phase, AI-generated code, uses intelligent refactoring tools to help software developers produce cleaner and more efficient code (Sindhwad et al., 2024), although, in the testing phase, LLMs can predict areas of code most likely to fail and generate artificial datasets for robust evaluation (Majdinasab et al., 2024). When deploying applications to the production environment, AI contributes by monitoring software project behavior in production environments, detecting anomalies that could signal security vulnerabilities (Bécue et al., 2021).

Furthermore, AI can predict software project timelines by detecting software bottlenecks, and assisting in resource allocation, which optimizes operational aspects of SD (Alenezi & Akour, 2025), similarly when AI is embedded in both the technical and managerial aspects of

software creation within an organization, making it a vital tool which software developers must carefully oversee (Fountaine et al., 2019), by monitoring AI's behavior within SD is crucial to ensure that AI-generated code is reliable, secure, and ethically sound (Żywiołek, 2024). In contrast, responsibility for secure code falls on the oversight of software developers, cybersecurity teams, and increasingly, compliance officers trained in AI governance frameworks (Żywiołek, 2024).

Many organizations have started to implement formal "Human-in-the-Loop" (HITL) practices, where human reviewers audit AI-generated code before it is merged into production environments (Seghier, 2025). HITL mechanisms aim to balance the speed of AI outputs with human judgment, ensuring that critical contextual factors are not overlooked. However, at a more comprehensive level, regulatory bodies and organization consortia, such as the European Union's Trustworthy AI initiatives, have proposed ethical guidelines mandating transparency, fairness, and human oversight over LLMs used in software development contexts (Cheatham et al., 2019) to point out that due to the growing concerns about AI-generated code security vulnerabilities, organizations have started developing governance recommendations when deploying software projects (NIST, 2024; OpenAI, 2024). Similarly, GitHub, for example, now encourages users of GitHub Copilot to manually review all AI-generated suggestions, explicitly warning that generated code may include insecure patterns (OpenAI, 2024; Pearce et al., 2025; Sindhwad et al., 2024). However, OpenAI (2024) recommends software developers to implement a human review process before deploying code generated by LLMs, which evolves governance norms that suggest a more general recognition within the software organization that AI-generated code cannot be trusted without critical human oversight. Embedding these governance practices into the software development life cycle aligns with the risk management principles advocated by

both the NIST RMF and the NIST AI RMF and supports the rationale for this study's HITL assumption; however, governance practices such as Human-in-the-Loop auditing are being adopted (NIST, 2024; OpenAI, 2024), there remains limited research evaluating the extent to which these practices effectively mitigate real-world security vulnerabilities in AI-generated code compared to human-developed code (Sindhwad et al., 2024).

Despite these measures, monitoring AI remains challenging, since most of the AI-generated code is unclear due to the complexity of the LLMs, making it difficult to trace how certain code decisions were made (Yigit et al., 2025), which is further explained by Żywiołek. (2024) on how this lack of transparency underscores the critical need for more capable auditing tools, better documentation of AI processes, and ongoing education for software developers working alongside AI. Security vulnerabilities introduced by AI-generated code have emerged as a significant concern among software developers (Cotroneo et al., 2025; Majdinasab et al., 2024; Sindhwad et al., 2024); however, LLMs can write functional code quickly, they often lack a deep understanding of security best practices, especially when their training datasets contain insecure or outdated coding patterns (Cotroneo et al., 2025; Sindhwad et al., 2024).

Table 2

Discovered AI-Generated Code Security Vulnerabilities

Type of Security Vulnerabilities	Description	Resource
Injection Flaws	AI-generated code sometimes includes vulnerable code allowing SQL injection or cross-site scripting (XSS) attacks	(Fu et al., 2024)
Authentication Weaknesses	Incorrect implementation of authentication or authorization logic, exposing sensitive systems to unauthorized access	(Ambati et al., 2024)
Improper Error Handling	AI-generated code may unintentionally leak information through verbose error messages that could be exploited by attackers	(Majdinasab et al., 2024)
Memory Management Errors	In languages like C or C++, AI tools can generate code with buffer overflows or memory leaks if not properly constrained	(Cotroneo et al., 2025)

Note: Several types of security vulnerabilities have been observed

Research has shown that security vulnerabilities in AI-generated code are often more nuanced than those introduced by humans because the AI replicates insecure coding patterns consistently across multiple outputs (Sindhwad et al., 2024), where other studies stated this behavior makes detection harder using classic manual code reviews (Kotb et al., 2025; Sindhwad

et al., 2024). Additionally, real-world testing of AI-generated software projects has revealed gaps in standard static analysis tools, such as CodeQL (Youn et al., 2023). As AI outputs grow more complex, it becomes increasingly necessary to combine traditional cybersecurity assessments with AI-specific vulnerability analysis, as demonstrated by initiatives like MultiQL, an extension of CodeQL for multilingual and AI-generated code analysis (Youn et al., 2023). In response to these challenges, developers must assume a proactive stance toward AI outputs (Patel. D. et al., 2023; Żywiołek, 2024). Rather than trusting AI-generated code at face value, rigorous security auditing, formal verification, and red teaming exercises should be employed to surface hidden risks (Patel. D. et al., 2023; Żywiołek, 2024).

Security Static Analysis Tools for Assessing AI- and Human-Generated Code

Due to the essential need to find security vulnerabilities in software projects that are either manually developed by software developers or generated by AI, static security analysis tools have become crucial in software development. These tools are used to systematically explore codebases or code repositories without running them by the virtue of predefined rules and heuristics and to help search for security vulnerabilities or adherence to secure coding practices. The AI- or human-generated code contains many security vulnerabilities, which is why the software code can be evaluated through general static analysis tools to lower the possibility of these security vulnerabilities (Alqaradaghi & Kozsik, 2024). GitHub’s CodeQL has a powerful way of discovering security vulnerabilities. While Youn et al. (2023) praise CodeQL's semantic query capabilities, Fröberg (2023) cautions that CodeQL's effectiveness depends heavily on well-formed query design, which may miss novel AI-induced security vulnerabilities. This suggests that while CodeQL excels in structured environments, it may underperform in

evaluating dynamically generated AI code unless customized extensions like MultiQL are applied (Youn et al., 2023).

Table 3

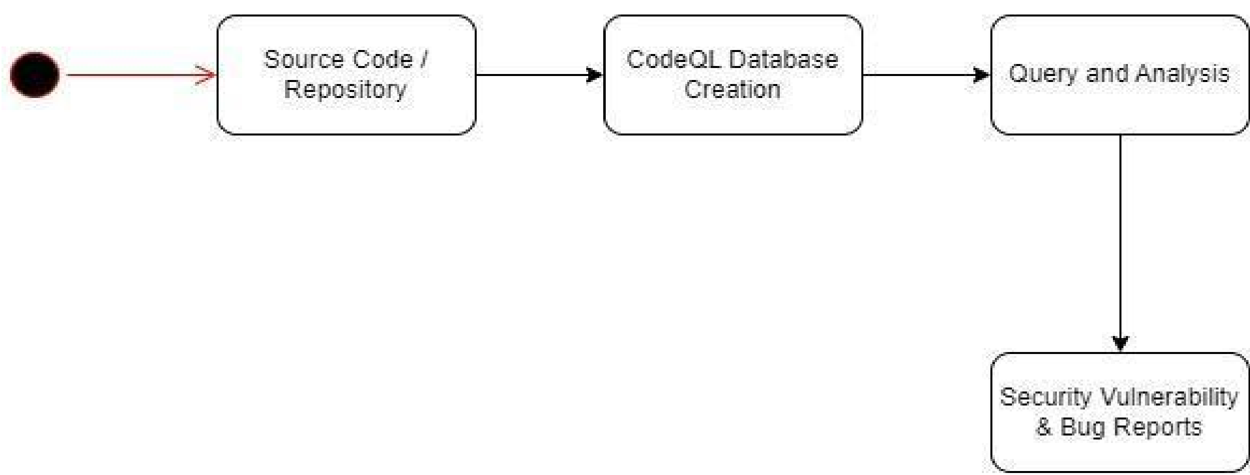
CodeQL Key Features

Key Feature	Description
Programming Language Support	C, C++, C#, Go, Java, JavaScript, TypeScript, Python, Ruby, Swift, QL
Query Language	Logic-based programming language for creating precise queries
Extensibility	Ability to create custom queries
CI/CD Integration	Integration with continuous pipelines
Scalability	Ability to analyze large codebases
Assistance and Community	Active community with plenty of documentation
Accessibility	Open-source software

Note: Lists of the main key features of what makes CodeQL efficient for security vulnerability analysis (Majdinasab et al., 2024).

Table 4

CodeQL's Workflow



Note: Illustrates CodeQL's workflow starting at the source code to discover security vulnerabilities and creating bug reports (Wehmeier et al., 2023).

Previous comparative research has already demonstrated that CodeQL has been able to identify security vulnerabilities which traditional static analysis has failed to detect. Fröberg (2023) proved that CodeQL is as accurate and complete as many other traditional analysis tools such as SonarQube and Checkmarx in identifying prototype security vulnerabilities, yet Alqaradaghi and Kozsik (2024) performed extensive testing on a few static analysis tools for finding security vulnerabilities using the Java programming language and determined CodeQL to be superior to most other well-known static analysis tools. CodeQL can represent complex code flow compared to tools like SonarQube and Checkmarx, which rely on static rule sets, CodeQL's declarative logic allows for deeper code interrogation (Fröberg, 2023). However, Majdinasab et al. (2024) highlighted that CodeQL still requires human oversight to validate results, especially when applied to AI-generated code that may contain syntactically correct but semantically misleading patterns. Due to the slight differences between human-generated code and AI-generated code, when it comes to the type and complexity of possible security vulnerabilities, it is important to use a robust static analyzer such as CodeQL that will be capable of taking these subtleties into account (Clark et al., 2024). Despite the strengths of CodeQL in security vulnerability detection (Cotroneo et al., 2025), there remains limited research investigating its effectiveness in identifying the nuanced security vulnerabilities specific to AI-generated code as compared to human-generated code (Fu et al., 2024; Sindhwad et al., 2024). Previous research has lacked in producing comprehensive quantitative assessments of how AI-generated code versus human-generated code security vulnerabilities perform in using static analysis tools. A flexible mechanism of targeting and classifying security vulnerabilities is provided by CodeQL,

all due to which it helps to conduct a causal-comparative analysis in a way that clearly differentiates the security vulnerabilities generated with LLMs and those that are a result of software developers.

Despite CodeQL sophistication, other researchers warn users not to trust static analysis tools without the assistance of software developers, where Alqaradaghi et al. (2022) and Sindhwad et al. (2024) stated that even with tools such as CodeQL, there could be cases with false positives or where it simply misses security vulnerabilities that required manual review to fully identify. These strengths and limitations highlight the importance of applying CodeQL within a systematic causal-comparative design, as performed in this study, to accurately distinguish security vulnerabilities between AI- and human-generated code while acknowledging potential detection biases (Fu et al., 2024; Youn et al., 2023). The other important point regarding CodeQL is how it can evaluate security vulnerabilities across many programming languages in a consistent manner (Majdinasab et al., 2024). Similar comparative analyses on programming languages must use a tool which can provide consistent analytical criteria in human-generated code as well as in AI-generated codebases written in multiple programming languages. The support for widely used languages like Java, Python, and JavaScript makes CodeQL a unique tool as it can produce reliable security vulnerability comparisons independent of the programming language. However, CodeQL has been shown to struggle with C and C++ (Li, Z. et al., 2024). Yet, the capability was particularly praised by Youn et al. (2023) as CodeQL's multilingual analysis is considered a major strength of this static analysis tool in enabling the validity and reliability for comparative security vulnerability research.

CodeQL has some strengths, but it is important to appreciate that correctly applying queries for AI-generated code to specific security vulnerabilities is hard, where Cotroneo et al.

(2025) and Hajipour et al. (2024) stated that finding security vulnerabilities in AI-generated code is nuanced and expensive. Not every security vulnerability falls easily into the traditional security categories; therefore, CodeQL can accommodate custom queries to detect patterns that would not otherwise notice a security vulnerability by using standard rules written to detect common misuses in code. However, there is a steep learning curve for software developers to write even the most basic query in CodeQL, where the other static analysis tools such as SonarQube and Checkmarx are user-friendly (Youn et al., 2023). Causal-comparative assessments on CodeQL will generate the ultimate answer to what the comparative security vulnerability profiles of AI-generated code versus human-generated code really are, an answer of fundamental importance to cybersecurity practice, software engineering methodology, and strategies for using LLMs in software development.

Table 5

Potential Weaknesses of Security Analysis Tools

Security Analysis Tool	Potential Weaknesses	Reference(s)
CodeQL	- False positives due to rule-based approach - Limited real-time detection - Struggles with runtime vulnerabilities	(Delicheh et al., 2024)
SonarQube	- High false positive rate - Limited AI-generated code detection - Requires frequent rule updates	(Alqaradaghi et al., 2022)
Checkmarx	- Cannot detect static code vulnerabilities - Performance overhead in real-time scanning - Struggles with AI-generated code logic flaws	(Xue et al., 2023)

Note: GitHub's CodeQL is new and made to handle AI-generated code whereas the others are older.

Discussing static analysis tools like CodeQL ties into RQ1b by illustrating how security vulnerabilities are detected and assessed across AI and human-generated code under similar conditions.

Comparing Human-Generated Code with AI-Generated Code

Human-generated code has historically served as the backbone for a software organization, providing carefully crafted, context-aware solutions tailored to specific problems (Li, J. et al., 2024), as pointed out by Gilstrap et al. (2024) software developers bring domain knowledge, creativity, and real-time decision-making to software construction, often solving complex challenges that require nuanced understanding. Software developers also perform iterative testing, intuitively identifying potential edge cases and usability concerns that LLMs might overlook (Sindhwad et al., 2024) and Fu et al. (2024) stated an important strength of human-generated code is its flexibility and adaptability to evolving requirements. Software developers can quickly modify software projects by responding to stakeholder feedback or market shifts, aligning with the business rules. Similarly, human oversight ensures that security protocols can be manually integrated during the early design rather than retrofitted later, promoting higher-quality, resilient systems (Cotroneo et al., 2025).

The literature shows that although humans are prone to error, their ability to recognize context, ethics, and evolving software projects needs uniquely positions human-generated code to address broader organizational objectives (Fu et al., 2024; Gilstrap et al., 2024; Żywiołek, 2024). Software developers, unlike AI, can incorporate subjective considerations such as fairness, accessibility, and long-term maintainability, which may not be present in LLMs (Żywiołek, 2024), where Fu et al. (2024) and Gilstrap et al. (2024) stated that despite their advantages, software developers introduce security vulnerabilities due to cognitive biases,

knowledge gaps, and human error, which is further pointed out how other studies consistently show that memory lapses, misinterpreting requirements, and poor secure coding practices often lead to critical security vulnerabilities in software projects (Cotroneo et al., 2025; Sindhwad et al., 2024). For instance, software developers under pressure to meet deadlines may bypass security best practices, increasing the risk of buffer overflows, SQL injection attacks, and privilege escalation flaws.

Research on human error indicates that specific security vulnerabilities are rooted in technical mistakes, flawed mental models, and decision-making strategies (Li, J. et al., 2024) causing software developers to possibly over-rely on outdated libraries, fail to validate user inputs, or mishandle memory management tasks, creating openings for exploitation (Fu et al., 2024; Li, J. et al., 2024). These issues compound organizational aspects, such as inadequate security training, a lack of peer code reviews, and limited resources for secure development practices (Fu et al., 2024), whereas Li, J. et al. (2024) reveals that while human judgment offers strengths like creativity and critical thinking, it simultaneously introduces subjective security vulnerabilities that can accumulate over time. Maintaining a robust human codebase requires ongoing education, structured development practices, and cultural emphasis on secure design principles.

Communication and coordination problems among software development teams mainly in large scale or far-reaching software projects is also the root of human-generated security vulnerabilities (Nørbjerg & Dittrich, 2024), also mentioned in a similar fashion by Li, J. et al. (2024), such misalignment of security standards among different teams, ambiguous accountability for security testing, and inconsistent usage of the secure coding guidelines result in not noticing or fixing security vulnerabilities. Effective security relies heavily on consistent

collaboration, clear security protocols, and shared accountability among all stakeholders within software development software projects (Li. J. et al., 2024; Nørbjerg & Dittrich, 2024). If there is not a structured communication channel and shared responsibility, then security vulnerabilities will propagate through human-generated code. Qadir (2023) as well as Nofal et al. (2025) emphasized that the developers' exposure towards up-to-date cybersecurity trainings, best practices and tactics related to threat modeling consistently diminishes the occurrence of security vulnerabilities but has yet to be proven so. Highlighting common security vulnerabilities in human-generated code provides a benchmark against which the security posture of AI-generated code will be measured, directly addressing RQ1 and RQ1a.

The role that programmer behavior plays in the introduction of security vulnerabilities in human-generated code is very central. As modern documentation explains, insecure coding practices are the source of repeated patterns of human error, such as lapses of memory, poor judgment, inattention, and adopting the wrong assumption (Anu et al., 2020; Patel. D. et al., 2023). They include failing to validate user input, forgetting to remove debug code, not paying attention to API documentation, assuming about the execution environment or user behavior. Anu et al. (2020) classified these issues according to the cognitive error model using reason and structured the framework to explain as to why developers misclassify secure code as highly vulnerable despite secure coding standards. Patel. D. et al. (2023) similarly noted that experienced software developers are prone to biases and cognitive failures, particularly under complex system demands or multi-tasking environments.

AI-generated code introduces dramatic efficiencies in software development, offering rapid code generation, automation of repetitive tasks, and accelerated prototyping (Ambati et al., 2024; Majdinasab et al., 2024), where LLMs are trained on extensive codebases, such as an

online public source code storage tool like GitHub, enable software developers to generate functional code snippets within seconds, greatly enhancing productivity for simple and moderately complex tasks (Alenezi & Akour, 2025). One significant advantage is AI's ability to assist software developers by identifying standard design patterns, applying secure coding guidelines, and suggesting common vulnerability fixes (Bannon & Laplante, 2024), such as using tools like GitHub Copilot and other AI-based assistants improve developer workflows, reduce time-to-market, and democratize access to software programming by supporting novice developers who may lack profound technical experiences (Sindhwad et al., 2024).

These studies highlight that while AI cannot yet replace deep human insight, it acts as a "force multiplier," particularly in automating routine and repetitive tasks (Ambati et al., 2024; Majdinasab et al., 2024). Furthermore, when integrated thoughtfully, AI-enhanced development workflows can enable higher baseline cybersecurity by suggesting safer coding practices by default (Cotroneo et al., 2025). Despite its benefits, AI-generated code presents distinct security vulnerabilities that software developers must manage. One core issue is that LLMs are only as good as the training datasets. Training datasets often contain insecure coding practices, outdated techniques, or even malicious examples, which AI may inadvertently replicate (Cotroneo et al., 2025; Sindhwad et al., 2024), where Ambati et al. (2024) stated LLMs also lack accurate contextual understanding, unlike software developers, where LLMs may generate syntactically correct but semantically flawed code, introducing security vulnerabilities that are difficult to detect through traditional testing methods. For example, AI-generated SQL queries might omit proper sanitization steps, increasing the risk of injection attacks if not manually reviewed.

Research conducted by public GitHub repositories have revealed security vulnerabilities in AI-generated and human-generated code, yet with different patterns (Cotroneo et al., 2025;

Sindhwad et al., 2024). For example, an AI-generated Python codebase repository intended for data visualization was found to have insecure file-handling practices, potentially enabling arbitrary code execution via crafted input files. In contrast, a human-developed Node.js web application exhibited a classic SQL injection vulnerability due to improper user input sanitization (Cotroneo et al., 2025; Sindhwad et al., 2024). These examples illustrate the distinct mechanisms by which security vulnerabilities occur in AI-generated and human-generated code, where these findings reveal how human-generated code security vulnerabilities often stem from cognitive errors and organizational factors, AI-generated vulnerabilities frequently originate from dataset contamination and limited contextual understanding, necessitating targeted mitigation strategies for each type (Cotroneo et al., 2025; Sindhwad et al., 2024). While security vulnerabilities introduced by software developers and LLMs have been individually studied (Alenezi & Akour, 2025; Sindhwad et al., 2024), there remains a gap in research systematically comparing the types, severity, and frequency of security vulnerabilities between AI-generated and human-generated code through a causal-comparative approach (Cotroneo et al., 2025). Recent studies have started to examine the security implication of the AI generated code, but significant gaps still exist in the empirical quantification of the security risks. In particular, Oh et al. (2024) examined how poisoning attacks affect the development of an AI-generated code on poisoning attacks using lab experiments and surveys. Although their work provides useful information about the behavior of developers and how they trust AI tools, it did not include large-scale and quantitative analysis of actual codebases. In the same fashion, Wang et al. (2024) presented CodeSecEval, a benchmark dataset that could be used to assess the security vulnerability of code generated by LLMs. However, their studies were only tested with controlled experiments, using synthetic test cases and model outputs, but not actual data of a repository. Both studies did not

directly compare AI-generated and human-generated code in the production setting. Conversely, this empirical gap is addressed in the current study where it involves the use of a large-scale causal-comparative analysis of GitHub repositories. This study provides a data-driven assessment of the code security through the use of CodeQL to extract the CVSS scores and the relative severity of security vulnerability in the AI-generated code and human-generated code. This practice-based methodology is informatic to and beyond previous research by basing its conclusions on real-life software development.

Table 6

Research Insights on Code Generation Security Vulnerabilities

Reference	Research Focus	Code Type	AI Model/Tool	Programming Languages	Key Findings
Negri-Ribalta et al. (2024)	Impact of AI models on code security	AI-generated	Various LLMs	Multiple languages	Examination of typical security weaknesses found in AI produced code.
Kaniewski et al. (2024)	Vulnerability handling in AI-generated code	AI-generated	Various LLMs	Not specified	AI generated code vulnerability management problems which are present in current solutions.
Li, J. et al. (2024)	Evaluating LLMs on secure code generation	AI-generated	Various LLMs	Not specified	A security evaluation needs to be introduced for LLM-generated code because models normally overlook security considerations throughout coding creation and maintenance processes.
Cotroneo et al. (2025)	Data poisoning attacks on AI code generators	AI-generated	Various LLMs	Not specified	Analyzed new data poisoning attacks which when used together produce security weak points in code while proposing remediation methods.
(Asare et al., 2023)	Vulnerability introduction by AI code generators	AI-generated	GitHub Copilot	C/C++	The code generation capability of GitHub's Copilot duplicates original vulnerable code in approximately 33% of cases which indicates its vulnerability creation proficiency is comparable to human developers.
Hamer et al. (2024)	Comparing security vulnerabilities in AI and human-generated code	Both AI-generated and Human-generated code	ChatGPT, StackOverflow	Java	The security inspection revealed that the generated code from both StackOverflow and ChatGPT contained vulnerabilities which affected 20% of ChatGPT and 46% of StackOverflow code bases.

Note: This table shows the most recent documented security vulnerabilities in AI-generated and human-generated code.

Table 7

Comparison of Common Security Vulnerabilities

Security Vulnerability	AI-generated Code	Human-generated Code	Reference
Buffer Overflow	Situations like this remain infrequent although they emerge through insecure patterns.	When bounds checks are neglected memory spaces become overwritten.	(MITRE, n.d.)
SQL Injection	Model performance shows the ability to duplicate insecure query patterns present in its training information.	By neglecting to sanitize input data applications become vulnerable to database manipulation attacks.	(MITRE, n.d.)
Cross-Sight Scripting (XSS)	Weak or bad data during model training can result in similar system problems.	Software systems remain vulnerable to malicious script injections because they fail to validate input correctly.	(MITRE, n.d.)
Hard-Coded Credentials	This problem remains uncommon yet exists when the system receives secret inclusion prompts.	An exposure to potential attackers exists when you store sensitive information inside your code.	(MITRE, n.d.)
Rare Conditions	Earlier time dependence problems appear unexpectedly in the output code.	Poor synchronization in concurrent processes.	
Inadequate Error Handling	May generate incomplete error-handling mechanisms.	Failure to manage exceptions securely.	(MITRE, n.d.)
Bias in Training Data	Training teaches and leads to the production of insecure patterns.	N/A	(MITRE, n.d.)

Insecure Defaults	Weak configurations and algorithms by default.	N/A	(MITRE, n.d.)
Overfitting	Insecure handling of edge cases happens because of inadequate generalization capabilities.	N/A	(MITRE, n.d.)

Note: The security vulnerabilities for human-written technical solutions present unique challenges which share similarities with those found in AI-generated code implementations.

Summary

This chapter presented both theoretical studies and practical findings about AI-generated code and human-generated security vulnerabilities. The study grounds its foundation through theoretical framework by implementing structured security risk assessment guidelines which include NIST RMF and NIST AI RMF. The discussed frameworks demonstrate the need for standardized methodologies to identify security weaknesses in code created by LLMs. Human-generated code has always been prone to security vulnerabilities due to time constraints to deploy the software project as soon as possible or software developers who lack the proper training to mitigate security vulnerabilities.

Human programmers uphold code security with the help of modern software development practices such as development operations (DevOps), development security operations (DevSecOps), and new AI/machine learning operations (MLOps) pipelines frameworks, which encapsulate automated static and dynamic testing of security. However, these techniques do not completely address the security vulnerabilities that both AI-generated and human-written code may create. While AI-generated code needs further evaluation before deployment because of potential security vulnerabilities. Humans need to supervise AI-generated code during software development because it highlights the need for identifying and resolving security vulnerabilities in software projects. Security vulnerability scanning tools operate

effectively at detecting security vulnerabilities in software projects, both in AI-generated code and human-generated code. The existing research on AI-generated security proves useful but researchers have yet to conduct causal-comparative studies which directly compare human-generated code safety to AI-generated code. In chapter 3, there will be detailed coverage of the methodology that will leverage quantitative methods to evaluate AI-generated and human-generated security vulnerabilities following the defined processes for answering and testing the research questions and hypotheses.

Chapter 3: Research Method

The problem addressed in this study is generating source code using AI increases the risk of security vulnerabilities stemming from data poisoning, compared to traditional human-generated code. The purpose of this quantitative causal-comparative research paper aimed at investigating the security vulnerabilities between AI-generated and human-generated source code. This chapter outlines exactly what methodology and design the researcher will conduct given the population and sample, what instruments and materials will be used when the specific steps are taken to process and analyze the data gathered with information on what the assumptions, limitations and delimitations pertain to this study, and the ethical considerations of what biases the researcher may possess. To solidify this study the research employed a quantitative methodology using a causal-comparative design.

Research Methodology and Design (Nature of the Study)

The research implemented a quantitative methodology for studying the security vulnerabilities present in AI-generated code compared to human-generated code in public code repositories on GitHub, which is appropriate for the problem, purpose and research questions for this study. This methodology provided an objective measurement necessary to accurately compare AI- and human-generated source code (Browning et al., 2024). This method also makes it possible to address the research questions and to test the hypothesis using t-tests and chi-square statistical numerical values (Noiklueb et al., 2025). Due to the nature of the quantitative methodologies critical cybersecurity research is appropriate, since data is what was measured in this study.

This research implemented causal-comparative design, which was chosen to accurately compare AI- and human-generated source code security vulnerabilities, where there is no

experimental manipulation (Jian Jim Hu et al., 2011). This design used the existing or naturally occurring groups to examine the difference among them on stipulated outcome variables (JD & Jr, 2004). The salient feature of this design is that the independent variables are categorical and cannot be directly manipulated by the researcher (JD & Jr, 2004). The independent variables are not controlled and as such, such studies are threatened with limitations in building a strong internal validity (JD & Jr, 2004). Public GitHub repositories were where the researcher gathered and analyzed data from real-world software projects, which ranges from individual use of the software projects up to and including organizations. It would not be ethical to tamper with the source code from the GitHub repositories in an experimental manner, therefore this design provided ethical and practical constraints.

A qualitative methodology was considered to understand better if AI-generated code helps software developers learn how to program and gain skills for security improvements to how they generate human code or if software developers use it more like a crutch where they stop gaining knowledge. There have been a few studies on this already; however, from research it was mainly done on students in either college or high school and not on professional software developers (Qadir, 2023). A mixed method was not chosen for this study due to its nature of including non-numerical data, where this quantitative study will contain numeric data.

A quasi-experimental design was considered for this study; however, it would be more of a cause-and-effect relationship, which does not align with the research questions or hypotheses. Cause-and-effect would cause the public GitHub repositories to be manipulated to see what security vulnerabilities would be produced. This does not align with the problem and purpose statements where AI-generated source code contains security vulnerabilities on its own not what would be injected after the output from the LLMs. With this type of design there is no random

sampling; instead everything is considered to be naturally occurring; however, the occurrences of newly created AI- and human-generated source code would not be considered a correlational design, since this study is not analyzing how the two variables relate to one another.

Population and Sample

The researcher gathered information from GitHub repositories that are freely accessible to analyze security vulnerabilities found in programs produced by LLMs and human developers. The total population for all repositories across different public platforms across all git platforms are approximately ~154,600,000 repositories; however, GitHub has been chosen for this study since it is the most representative compared to the other public repository platforms. Yet, this total population is an accumulation of obsolete source code to newer code using newer software development standards. GitHub provided a search tool to find AI-generated source code from the human-generated code; making it easier to determine which repositories are AI- or human generated, then the researcher can determine if GitHub's Copilot was used in the repository, whereas other repository platforms do not have this yet.

For the research sample, the researcher used a purposive sampling based on clear operational definitions distinguishing AI-generated from human-generated code to select 67 repositories of AI-generated code together with 67 repositories of human-generated code that were appropriate for the problem and purpose statement to have ample samples for testing the research questions hypotheses. Repositories were selected using GitHub's search functionality by filtering explicitly for Python, Java, and JavaScript programming languages to capture real-world software development practices and employing search keywords such as "AI-generated", "Copilot-generated", "ChatGPT-generated", and the READ.ME file to identify AI-generated codebases to ensure the absence of human-generated code. To ensure human-generated code

repositories are not mixed with AI-generated code, and strictly human, the READ.ME file will also be used as a filter. Relying on GitHub metadata and/or excluding hybrid repositories for potential bias could limit the integrity of the sample; therefore, by omitting nuanced project contexts and/or repositories that combine AI- and human-generated code may potentially prevent skewed results. To ensure objectivity and balanced representation of each programming language, the researcher randomly selected an equal number of public repositories per programming language from GitHub based on programming language tags and only gather data from the files containing the language of interest matching the language display badge.

A power analysis was done with respect to two tail a priori power analysis using G*Power 3.1.9.7, to approximate the minimum sample size required. There is one independent variable (two groups of AI-generated and human-generated code), one dependent variable: lines of code, and one screening criteria: programming language. Using a medium effect size ($f = 0.5$) making it a reasonable default when a prior effect sizes are unknown (Groß & Möller, 2024). Along with a significance level ($\alpha = 0.05$), desired power ($1 — p = 0.80$) and an allocation ratio of 0.69, the number of code samples needed for a total sample size of 134 is concluded.

Materials or Instrumentation

The data source is a combination of GitHub public repositories where the source code is both AI- and human-generated code. The instrumentation is used to determine if security vulnerabilities exist and to what extent using the static analysis tool CodeQL developed by GitHub (Youn et al., 2023). Accurate identification of common weakness enumerations (CWE) across diverse scenarios proved CodeQL efficiently detected security vulnerabilities in Copilot AI-generated code discovered by Majdinasab et al. (2024). Similarly, the study from Aanby et al. (2024) states CodeQL's ability to perform logical operations and handle recursive predicates

efficiently supports the development of complex queries without procedural overhead. To ensure the accuracy of CodeQL queries manual analysis of CWEs were verified and validated (Majdinasab et al., 2024).

Other static analysis tools considered were the SonarQube and Checkmarx; however, CodeQL was chosen due to its open-source nature, comprehensive security query capabilities, and seamless integration with GitHub repositories, making it ideal for analyzing public source code. When using CodeQL to scan AI- and human-generated code, a common vulnerability scoring system (CVSS) with the range of 0.0 up to 10.0 will be provided to assess if there is a security vulnerability. To ensure the validity of this study the researcher followed strict inclusion criteria when selecting public GitHub repositories. The researcher also manually reviewed the security vulnerability findings to make sure they are accurate, which helped reduce false positives and make the data analysis more reliable.

Operational Definitions of Variables

The independent and dependent variables are below (Table 8).

Security Vulnerabilities of AI-Generated Code and Human-Generated Code

The interval independent variable security vulnerabilities of source code generated by AI or software developers are a possible security vulnerability if the risk is severe that could potentially cause an outside actor to gain access to data. Each security vulnerability was measured on a continuous, interval scale using a standardized scoring method based on the common vulnerability scoring system. Security vulnerabilities identified by CodeQL, including specific CodeQL queries and classification schemes used to categorize vulnerability types by severity and frequency, were evaluated using the CVSS v3.1, which assigns scores from 0.0 to 10.0, with scores ranging from 0.0 to 3.9 indicating low severity, 4.0 to 6.9 indicating medium

severity, 7.0 to 8.9 indicating high severity, and 9.0 to 10.0 indicating critical severity (*CVSS v3.1 Specification Document*, n.d.). For statistical comparison of security vulnerability using CVSS the severity scores was grouped into standard categories, such as low, medium, high and critical.

Lines of Code

The ratio dependent variable lines of code are the total number of lines of source code in a software project, reflecting its size and potential exposure to security vulnerabilities.

Programming Language

The screening criteria of programming languages determined if the language used to create a software project creates security vulnerabilities, which will be limited to Python, Java and Javascript.

Table 8

Operational Table of Variables

Variable Name	Type	Measurement Level	Data Source
Security Vulnerabilities of AI-Generated Code	Independent Variable	Interval	AI-Generated Code Samples in Public GitHub Repositories
Security Vulnerabilities of Human-Generated Code	Independent Variable	Interval	Human-Generated Code Samples in Public GitHub Repositories
Lines of Code	Dependent Variable	Ratio	In a software project the total lines of code
Programming Language	Screening Criteria	N/A	Programming languages of either: Java, JavaScript or Python

Note: Variable overview.

Study Procedures

This study used a theory driven framework as seen by another study Petersen et al. (2023), which entails also relying on tools like data driven technologies such as CodeQL that is unique to this study. To determine authorship (AI- versus human-generated code) and the programming language, all repositories chosen were filtered with the help of the GitHub search and metadata features. Such determinations will depend on AI-assist tag, commit metadata, developer notes, and repository documentation indicators. One of the main shortcomings that has been noted in the available literature is that trusted detection of AI-generated code largely remains uniform within the GitHub Copilot system, where Microsoft has its back-end monitoring Copilot code-suggestion acceptance. This study addressed the variance by (a) eliminating those repositories where authorship cannot be established reasonably, (b) triangulating various indicators, i.e., tags, notes, and commit language, to minimize the risk of a misclassification, and (c) contextualizing results with the understanding that AI contributions outside the Microsoft/GitHub ecosystem (e.g., Claude, ChatGPT, Llama) may be underrepresented because they are not detected by the backend. Thus, the scope of the study represented a representative but not exhaustive sample of AI-generated code, thus compromising rigor of the methodology and feasibility of practicality.

The researcher was not the administrator of any public GitHub repository used in this study; therefore, the repository were cloned and imported into an integrated development environment (IDE), which was either Eclipse, Visual Studio Code or Visual Studio 2022 depending on the programming language used for the repository. Then the CodeQL binaries were loaded onto the researcher’s personal computer where the cloned source code and IDEs will also reside. CodeQL code was written which was one standard, universal code snippet that covers all

programming languages used in this study to then scan each repository. CodeQL will scan each repository for security vulnerabilities, and the resulting severity scores are compared to assessing whether AI-generated code is more likely to present higher security vulnerabilities to human-generated code.

CodeQL produceed reports, where only valid and completed CodeQL reports are considered, and reports that cannot be opened or have errors will be rejected. Any identical repositories or codebases are filtered to ensure the results are not biased. Detecting and removing any redundant content using the details in GitHub metadata. CVSS entry present were considered when using severity-based analysis. The data was evaluated for missing data, unusual values, or errors like labeling a code type as another programming language. This process guarantees that the empirical analysis using the sample is robust and the sample is representative. An audit trail was used to document each decision and rationale by allowing for transparency and cross-checking by another reviewer if/when needed for consistency in manual reviews.

Table 9

Gantt Chart

Task	Start Week	End Week	Duration (Weeks)
IRB Approval Secured	1	1	1
Repository Identification & Filtering	1	2	2
Cloning Repositories & IDE Setup	3	4	2
CodeQL Configuration & Scanning	5	7	3
Report Validation & Deduplication	8	9	2
Data Cleaning & Categorization	10	11	2
Statistical Analysis & Documentation	12	12	1

Note: Visualization of study procedures.

Data Analysis

This study's data analysis had conducted through a causal-comparative approach grounded in prior research (Petersen et al., 2023). This data analysis identified whether statistically significant differences exist in vulnerability severity between AI- and human-generated code, helping interpret whether AI-generated code poses greater security vulnerabilities than human-generated code based on comparative CVSS scores, via a report from CodeQL. The CVSS score (on a continuous scale of 0.0 to 10.0) as a quantitative measure of security vulnerability severity, which is appropriate for conducting an independent samples t-test. The comparison was based on the mean severity of security vulnerabilities found in each group

(AI vs. human). Each repository's mean CVSS score were calculated, and a t-test will compare the average severity of security vulnerabilities across the two groups. This allows the study to quantify whether AI-generated code tends to have higher or lower severity security vulnerabilities than human-written code, based on standardized industry scoring. Each vulnerability was assigned a continuous CVSS score using the formula outlined below:

$$CVSS\ Score = \{0,\ if\ Impact = 0;\ min(round(0.6 \times Impact + 0.4 \times Exploitability - 1.5)$$

(*CVSS v3.1 Specification Document*, n.d.)

See Appendix B for sample CodeQL code that integrates the CVSS score formula.

The CodeQL report was generated and then the data was extracted into a CVS file to use the statistical tool SPSS to determine how AI-generated code and human-generated code compared to each other and if there are similarities or differences concerning security vulnerabilities. By using the statistical tool SPSS, the researcher was able to make a proper statistical calculation based on the data obtained from CodeQL to correctly answer the research questions and test the hypotheses using t-test and chi-square tests as these testing methods were used in the Iwanaga et al. (2023) study. The SPSS tool was used to test assumptions for statistical tests, while missing, invalid, or inconsistent data, such as incomplete CodeQL reports, dirty code or obsolete code were identified during validation and excluded prior to analysis to ensure data integrity for reliable comparison across AI- and human-generated code.

Assumptions

What is assumed to be true in this study is how CodeQL can properly scan both AI- and human-generated code for security vulnerabilities. The sample of public GitHub repositories for both AI- and human-generated code were a representation of the security posture within the software projects. When software developers implement AI-generated code into the software

project they do not modify the AI-generated code. CodeQL was used to analyze the data the security vulnerabilities will be valid and meaningful in determining security vulnerabilities. T-tests and chi-square test were used in this study to appropriately compare AI-generated code and human-generated code for security vulnerabilities (Iwanaga et al., 2023), with assumptions regarding data normality and independence tested prior to applying t-tests and chi-square analysis, and with plans to handle missing or inconsistent data. This study assumed CodeQL accurately detects security vulnerabilities for the programming languages chosen for both AI- and human-generated code; however, hybrid repositories were excluded to protect internal validity, and since CodeQL is created by GitHub there were no compatibility issues as there might be with another repository platform.

It is presumed that the classification of repositories as either AI-generated or human-generated is correct as per the information gleaned in the README files, documentation and commit messages. Nevertheless, there is also a possibility that not all repositories that can be classified as having been created by humans, include code that is in part or fully generated by an AI system, like GitHub Copilot or ChatGPT, without any clear indication of it. The use of AI can be undetected; therefore, in this study, a possible misclassification bias in the sample is recognized. Although every effort was undertaken to check the repository classifications, such as manual inspection of descriptions and activity of the contributors wherever possible, there is no certain way to check with the origin of all code fully.

Limitations

CodeQL while reliable, may not detect all the security vulnerabilities leading to unknowns, false positives and/or false negatives (Delicheh et al., 2024; Sindhwad et al., 2024). When collecting and analyzing the data gathered from public GitHub repositories it was not

known what LLMs, and prompts were used to generate the AI source code that could potentially lead to inconsistencies in security vulnerabilities. Penetration testing was not used in this research, which does not confirm the presence of real-world security vulnerabilities that lead a software project to be exploitable or not. The selection of GitHub public repositories were based on availability of AI- and human-generated source code and not be an all-encompassing study of AI- and human-generated source code. Many of the GitHub repositories used in this study can and will change rapidly due to new technologies emerging in the computer science field, making the findings obsolete.

Delimitations

This study focused on public GitHub repositories and excluded other hosted public repositories as well as also excluding private GitHub repositories, which may have different security practices. Only AI- and human-generated source code were analyzed, and hybrid source code was excluded where software developers modify the AI-generated source code. CodeQL was the only static security analysis tool used in this study, where the results were not checked against other static security analysis tools. As shown in the previous chapter, LLMs specialize in a select few programming languages causing certain programming languages to be excluded from this study. Lastly, this research did not assess the quality of the AI- and human-generated code outside the primary focus of security vulnerabilities.

See Appendix C for more information in Table 9.

Ethical Assurances

Minimal ethical considerations arise from this study because the researcher executed the analysis on publicly accessible data that does not necessitate human subject involvement. The research follows Institutional Review Board (IRB) procedures through the selection of

repositories linking to open-source licenses while maintaining protection of personally identifiable information (PII) from their contributors. The protection and integrity of research data was achieved through secured data storage systems together with proper reporting methods that mitigate security hazard disclosures. To mitigate researcher bias, a paper-based audit trail was maintained throughout this study to document decision making, coding rationale and analytical steps to ensure transparency to allow for external review if/when needed. Checking both the code comments or README file, the researcher will ensure labeling accuracy for AI- and human-generated repositories and if uncertain or ambiguous the repository was excluded from the dataset. The research process consistently maintained data security protocols. A password-protected encrypted storage platform will safeguard all gathered data so authorized users remain the only ones who can view it. The study prevented public disclosure of sensitive security vulnerabilities to protect against possible attacks of unsecured code. Results presented aggregated data to protect privacy during responsible research publication with consistent scientific integrity.

The researcher does possess known biases when it comes to AI-generated code. For instance, software developers who are new at programming tend to benefit from AI-generated code; however, more experienced software developers tend to notice the flaws of the AI-generated code where it either generates code that gives incorrect results, or it generates code that breaks existing code in the software project (D. Palmer, personal conversation, February 5, 2025). AI-generated code really isn’t AI yet, but more like a word scraper that can be made from a python script of what the user inputs and then the LLMs queries a large dataset that matches an algorithm given by the LLMs creator to return a result back to the user (D. Palmer, personal communication, February 5, 2025). LLMs are not believed to be a true intelligent system at this

time until new technology is created to integrate quantum computing and quantum consciousness, then LLMs would think like a human brain; however, LLMs probably could exceed the human brains, since human brains are limited by physical aspects (D. Palmer, personal communication, February 5, 2025).

Summary

This study examined security vulnerabilities in AI-generated code compared to human-generated code, focusing on public GitHub repositories. The research employed a quantitative causal-comparative approach using CodeQL to detect security vulnerabilities. This study includes the theory of Intellectual Capital (ICT), NIST Risk Management Framework (RMF) and NIST AI RMF to assess security vulnerabilities. AI-generated code, while improving efficiency, introduced unique security vulnerabilities due to training data poisoning, lack of contextual awareness, and reliance on insecure coding patterns. The findings will help software developers and organizations mitigate security vulnerabilities associated with AI-generated code while providing insights into best practices for safe software development.

Chapter 4: Findings

The findings addressed security vulnerabilities in both AI- and human-generated source code from online public GitHub repositories. As AI-generated source code continues to become the go to tool for faster software development for software developers and organizations alike, it became necessary to reinforce the cybersecurity knowledge of such LLM tools for the safety of its users. Consequently, the problem addressed in this study is generating source code using AI increases the risk of security vulnerabilities stemming from data poisoning, compared to traditional human-generated code. Thus, the purpose of this quantitative causal-comparative research paper aims at investigating the security vulnerabilities between AI-generated and human-generated source code. This chapter presents the findings of the study organized by the research questions and associated hypotheses. First, the validity and reliability of the data are examined. Next, results are presented for RQ1, followed by RQ1a and RQ1b. The chapter concludes with an evaluation of findings and a summary.

Validity and Reliability of the Data

Construct Validity of Measures

Validity in this study refers to the extent to which the data and analysis procedures accurately measure the intended constructs of code origin (AI-generated vs. human-generated) and security vulnerability severity. Before carrying out the entire series of analyses made in CodeQL, an audit trail was marked on which public GitHub repositories were filtered based on GitHub's search filtering and then determined which CodeQL report, known as the static analysis results interchange format (SARIF) file, was valid. The main goals are as follows: (a) to ensure that the SARIF output generated by CodeQL reflected the security vulnerabilities reported in the results array of the SARIF file, (b) to make sure that the custom python processing script was

able to properly read the SARIF structure and retain important metadata (repository name, origin, language, and security vulnerability details) in generating the comma separated values (CSV) dataset, and (c) to ensure that the analysis pipeline could behave consistently when it operated on a full repository as opposed to a single file. To do it, one of the repositories was chosen and scanned using CodeQL. The resulting SARIF file was reported on human-generated code and inspected to ensure that every result corresponded to the summary information generated by CodeQL in the command-line output. The manual check ensured that the security vulnerabilities that CodeQL identified were correctly encoded in the SARIF structure and that no entries were either omitted or spurious when exported. The python script was then used to process the same SARIF file and then the CSV row(s) generated were compared against the original contents of the SARIF file to ensure that repository name, origin label (AI or human), programming language, and security_vulnerability type were correctly captured. Concerning the fact that CodeQL processes all the related source files within a repository, and not just one code file, it was also marked in the audit trail during this check. Consequently, the exact file where a security vulnerability took place was not recorded in the CodeQL analytic dataset, but each item was the occurrence of a particular security vulnerability in one of the repositories files with its origin, and language. This repository level interest guaranteed that the unit of analysis met the research questions, which expose the general trends of security vulnerability between AI-generated and human-generated code. One result of this action is that the initially predetermined counts of lines of code (as a potential measure discussed in chapter three) were no longer applicable to the final dataset, since CodeQL automatically counted results of all Python, JavaScript/TypeScript, and Ruby files in each repository. Therefore, the variable count of lines of code was not presented in the analysis file. Overall, it was seen that the

CodeQL_SARIF_Python_CSV pipeline operated as expected, and security vulnerabilities as of repository level were being captured faithfully and that important identifiers (origin, language, and repository) were not lost or corrupted in the process before the full-scale data analysis. From an internal validity perspective, this audit trail and pipeline verification reduced threats such as misclassification of origin, loss of cases during processing, or differential handling of AI-generated versus human-generated repositories, because all decisions about inclusion, exclusion, and unit of analysis were established a priori and applied symmetrically to both groups.

All security vulnerability data analysis was done in controlled and reproducible conditions with scripted automation and with the CodeQL command-line interface. The sample was cloned and scanned with established CodeQL security query suites on the primary language in each repository (Python, JavaScript/TypeScript or Ruby). All the repositories, including those where the code has been AI-generated and those where the code has been generated by humans, shared the same CodeQL configuration, query sets, and environment. The findings were exported as SARIF files, and a custom Python script was used to tabulate the results to one CSV file which is amenable to statistical analysis in SPSS. To enhance internal validity, the dataset was systematically cleaned and de-duplicated. When there were multiple instances of the same security vulnerability type reported in a repository by CodeQL (such as in multiple files or line positions), only one security vulnerability type was stored. The definition of this term was that of records that had identical repository identifiers, language, origin (AI vs. human), and type of security vulnerability. This made sure that in the analytic file, the rows of the files are generated with exactly one combination of the repository and type of security vulnerability and no duplicated flags of the same underlying issue. Repositories that did not reveal any security

vulnerabilities were not dropped but rather were stored in the form of analytic cases and studied with a special category, which was “no security vulnerabilities in this repo”, in the ‘security_vulnerability’ type variable. The initial intention was to calculate official common vulnerability scoring system (CVSS) v3.1 base scores of each security vulnerability, but the output of the SARIF structure that CodeQL generated lacked the complete CVSS v3.1 vectors (e.g., attack vector, attack complexity, privileges required, user interaction, scope and all impact measures) necessary to calculate the standard base scores. This caused it to be unable to derive an exact CVSS v3.1 base score based on the output of the CodeQL. In its place, a CVSS-like severity value was built based on the ‘security_severity’ field of CodeQL, which ordered findings by categorization into the following ordered levels: none, low, medium, high, and critical. These categories were scaled with the help of numeric 0 to 10 scale, where none was rated 0 with the increasing values of higher in severity categories. The derived variable (‘cvss_score’) maintains the relative arrangement of the severity and offers a continuous measure analysis applicable to inferential statistics. The research questions address the causal-comparative analysis aspects of the severity of AI-generated and human-generated code, the CVSS-like score was deemed suitable, as the identical scoring process was applied to all security vulnerabilities regardless of their source. Construct validity of this CVSS-like measure was supported by the fact that the underlying severity vulnerability categories (none, low, medium, high, and critical) are defined within CodeQL as ordered gradations of exploitability and impact, so the numeric 0–10 scaling preserves the intended ordering of the construct “security vulnerability severity”, because the adapted scale maps directly onto the complete set of CodeQL severity labels without omitting categories. This provides evidence of content validity with respect to CodeQL’s internal severity framework, whereas criterion related validity

relative to official CVSS v3.1 base scores could not be established, given that the SARIF output did not contain the full CVSS vectors required for an empirical comparison.

Reliability of CodeQL Outputs

All security vulnerability data analysis was done in controlled and reproducible conditions with scripted automation and with the CodeQL command-line interface. The sample was cloned and scanned with established CodeQL security query suites on the primary language in each repository (Python, JavaScript/TypeScript or Ruby). All the repositories, including those where the code has been AI-generated and those where the code has been generated by humans, shared the same CodeQL configuration, query sets, and environment. The findings were exported as SARIF files, and a custom Python script was used to tabulate the results to one CSV file which is amenable to statistical analysis in SPSS. The entire pipeline from repository cloning through CodeQL scanning to SARIF export and Python based tabulation was fully scripted and deterministic, rerunning the same queries on the same version of a repository would be expected to yield identical SARIF results; therefore, formal test and retest reliability were not separately estimated but is implicitly supported by the reproducible command-line workflow. Traditional inter-rater reliability does not apply, as no human coders judged the presence of security vulnerabilities; instead, a single static analysis engine (CodeQL) produced all findings. Algorithmic reliability was promoted by using a single CodeQL version, fixed query suites, and identical configuration setting across all repositories, thereby holding constant the algorithm and its decision rules. Potential sources of algorithmic bias remain a limitation, because CodeQL's rule sets may be more sensitive to certain security weakness patterns, programming languages, or coding idioms than others; however, any such bias should affect AI-generated and human-generated repositories in a comparable manner, given that both groups were processed

with the same toolchain, rules, and execution environment. The python processing script itself was checked against a known SARIF file, and no discrepancies were observed between the SARIF contents and the generated CSV records, supporting the reliability of the data extraction step.

Statistical Assumptions Testing

The hypothesis testing was preceded by the data analysis of the key statistical assumptions. The CVSS-like severity score was inspected in terms of distribution with AI-generated records and human-generated records. Even though the difference was not too much, independent samples t-test tends to be stable even in the presence of a moderate deviation of the normality provided that the groups of AI and human have more than 30 repositories. For the independent samples t-test comparing AI-generated and human-generated repositories on the CVSS-like score, the scale of measurement assumption was addressed by treating the cvss_score variable as an approximately interval level measure on a 0–10 continuum, and the independence assumption was met because each analytic case corresponded to a distinct repository with a fixed origin classification (AI vs. human). A formal check for normality was carried out by visually inspecting the “results” array in the SARIF file for the CVSS-like variables within each origin group; departures from normality were modest and, combined with sample sizes exceeding 30 per group, were judged consistent with the robustness conditions commonly cited for the independent sample t-test. The test of equality of the variances of the two variables between the AI and the human group was unequal, $F(1, 270) = 16.76$, $p = .001$ and so, the equal variances not assumed row were applied to interpret the t-test results. Thus, the homogeneity of variance assumption was explicitly tested using Levene’s F statistic and, when violated, was managed by

interpreting the Welch unequal-variances t-test (the "equal variances not assumed" row) rather than the pooled variance result.

In the case of the chi-square data analysis of origin (AI vs. human) by type of security vulnerability, 272 valid cases were taken. The Pearson chi-square test value was statistically significant, χ^2 (43, N = 272) = 65.04, p = .017, but 79 cells (89.8) were expected to have fewer than five counts, with the lowest expected count being 0.45. For the chi-square analysis, the independence of observations assumption was satisfied because each repository contributed at most one record for any given combination of origin and security_vulnerability type, and duplicate flags for the same underlying issue were removed during de-duplication. The assumption regarding minimum expected cell frequencies was evaluated by inspecting the cross-tabulation of origin by security_vulnerability type, which showed that many rare common weakness enumeration (CWE) combinations fell below the recommended threshold of five cases per cell. Consequently, the interpretation of chi-square results is done in a careful manner by focusing on certain CWE aggregates involving CWE-079, CWE-116, or CWE-352 appearing more frequently within one origin than the other variations of security vulnerability types based on origin and not by making solid conclusions about the rare combinations. In this way, the chi-square findings are used to highlight robust patterns in adequately populated CWE aggregates while recognizing the limited inferential value of sparsely populated cells.

Results

The resulting analytic data set had 272 valid records, each of which represented a distinct repository files, origin (AI vs. human), and security_vulnerability type (including a category of "no security vulnerabilities in this repo"). Two major inferential tests on SPSS were then performed after the removal of any records with a missing value on the variables of interest, (a)

independent sample t-test to assess the mean score of CVSS-like severity between AI-generated and human-generated code (RQ1), and (b) a chi-square test of independence to determine the relationship between code origin and vulnerability type (RQ1a and RQ1b).

Table 10

Descriptive Statistics for CVSS-Like Severity Scores by Code Origin

Origin	N	Mean CVSS-Like Score	Standard Deviation	Standard Error Mean
AI-generated code	122	3.09	3.204	0.290
Human-generated code	150	4.01	2.849	0.233

Note: Human-generated repositories had a higher mean CVSS-like severity score (M = 4.01, SD = 2.85) than AI-generated repositories (M = 3.09, SD = 3.20).

Table 11

Repositories with and without Detected Security Vulnerabilities by Code Origin

Origin	No security vulnerabilities detected (n, %)	One or more vulnerabilities detected (n, %)	Total (n, %)
AI-generated	61 (50.0%)	61 (50.0%)	122 (100.0%)
Human-generated	48 (32.0%)	102 (68.0%)	150 (100.0%)
Total	109 (40.1%)	163 (59.9%)	272 (100.0%)

Note: Half of the AI-generated repositories (50.0%) had no security vulnerabilities detected by CodeQL, compared to 32.0% of the human-generated repositories.

Research Question 1 / Hypotheses

RQ1

What is the statistical significant difference in the mean CVSS scores of security vulnerabilities between AI-generated code and human-generated code when analyzed with CodeQL?

$H1_0$

There is no significant difference in the mean CVSS scores between the security vulnerabilities in AI-generated code than that of human-generated code.

$H1_a$

AI-generated code exhibit significantly different mean CVSS scores security vulnerabilities compared to human-generated code.

Results:

The independent samples t-test showed a statistically significant difference in mean CVSS-like severity scores between AI-generated and human-generated code, with human-generated code having higher average severity (M = 4.01) than AI-generated code (M = 3.09), t(244.52) = −2.47, p = .014. Therefore, H10 is rejected and H1a is supported: there is a significant difference in mean severity between AI- and human-generated code.

Research Question 1a / Hypotheses

RQ1a

What security vulnerabilities may be common to both AI-generated code and human-generated code?

$H1a_0$

There is no significant difference in the security vulnerabilities between AI-generated and human-generated code.

$H1a_a$

AI-generated code will exhibit significantly more security vulnerabilities than human-generated code when analyzed using CodeQL.

Results:

The chi-square test indicated a significant association between code origin and security_vulnerability type, χ^2 (43, N = 272) = 65.04, p = .017, meaning security_vulnerability types are not distributed the same way in AI-generated and human-generated code. Some categories (such as “no security vulnerabilities in this repo” and certain CWE combinations) were more frequent in AI-generated repositories, while others appeared more often in human-generated code. Inspection of expected cell counts showed that 79 of the 88 cells (89.8%) had expected frequencies below 5, with a minimum expected count of 0.45, which violates the standard chi-square assumption regarding minimum expected cell frequencies. To manage this violation, the security_vulnerability types were recoded into a smaller number of superordinate categories (including “no security vulnerabilities in this repo” and two higher-frequency CWE groupings), and the chi-square test of independence was rerun on the collapsed table. In this follow-up analysis, based on the 118 cases that fell into the recoded categories, the association between origin and collapsed security vulnerability category was not statistically significant, χ^2(2, N = 118) = 4.41, p = .110, and 4 of the 6 cells (66.7%) still had expected counts below 5 (minimum expected count = 0.42). Accordingly, the initial 44-category chi-square result is treated as exploratory due to the severe assumption violation, and the collapsed chi-square result suggests that, within the broader superordinate categories used here, there is no statistically reliable association between origin and security vulnerability category. Therefore, H1a0 is rejected (there is a difference), and H1aa is only partially supported because AI-generated code does not simply show “more security vulnerabilities overall,” but it does show different patterns and higher frequency for some specific categories.

Research Question 1b / Hypotheses

RQ1b

What security vulnerabilities may be common to both AI-generated code and human-generated code?

H1b₀

There is no significant commonality of security vulnerabilities (e.g., input validation errors) of both AI- and human-generated code.

H1bₐ

Specific security vulnerabilities (e.g., input validation errors) will be more prevalent in AI-generated code compared to human-generated code.

Results:

The cross-tabulation showed that several security vulnerability categories (and the "no security vulnerabilities in this repo" condition) occurred in both AI-generated and human-generated code, indicating clear commonality in the types of issues that can appear in each. However, the relative frequencies of these shared security vulnerabilities differed by origin, with some being more prevalent in AI and others in human code. These descriptive patterns are consistent with the exploratory 44-category chi-square and the subsequent collapsed-category analysis, even though the latter did not reach statistical significance, $\chi^2(2, N = 118) = 4.41$, $p = .110$. Because a substantial proportion of cells in both versions of the analysis had expected counts below five, these results are best interpreted as indicating general tendencies in how shared security vulnerability categories are distributed across origins, rather than as strong evidence of precise origin specific prevalence for each category. Therefore, H1b0 is rejected (there are common security vulnerabilities across both), and H1ba is partially supported in that

some specific security vulnerabilities appear in both groups (AI and human) and are more prevalent in AI, while others are more prevalent in human-generated code.

Descriptive Statistics

In the case of the AI-generated code, there were 122 records (n = 122) and the mean severity score in terms of the CVSS-like was 3.09 (SD = 3.204). In the case of human-generated code, the size of the sample was 150 records (n = 150), and the average of the severity was 4.01 (SD = 2.849). AI-generated repositories had 61 out of 122 records (50.0%) and human-generated repositories had 48 out of 150 records (32.0%). In total, this AI- and human-generated accounted for 109 of 272 records (40.1%) in the dataset.

Inferential Statistics

RQ1, which hypothesized whether any statistically significant difference was observed in the mean severity scores on CVSS-like on AI-generated and human-generated code, was evaluated using an independent sample t-test. The test of Levene showed unequal variances, the results of that were not assumed to be equal. The t-test showed that there is a statistically significant difference in the mean severity with lower average CVSS-like scores among the AI-generated security vulnerabilities than the human-generated vulnerabilities (reported below under RQ1).

RQ1a and RQ1b, which tested the hypothesis that specific types of security vulnerabilities were more present in AI-generated code than human-generated code and which types of security vulnerability were shared between them, were addressed with the help of a chi-square test of independence. The statistics of Pearson chi-square was significant, $\chi^2(43, N = 272) = 65.04$, $p = .017$, which means that the distribution of types of security vulnerabilities was not equal between AI- and human-generated code. Even though the expected counts in many of

the cells were low, the large chi-square value and the trends in the cross-tabulation indicate the presence of interesting differences in the occurrence of security vulnerability across origins. The assumption regarding minimum expected cell frequencies was violated in the original 44-category table (89.8% of cells with expected counts < 5), the security_vulnerability type variable was subsequently collapsed into broader categories, and the chi-square test was repeated on the recoded table. The follow-up analysis on the collapsed categories did not reach statistical significance, $\chi^2(2, N = 118) = 4.41$, $p = .110$, and still contained a substantial proportion of cells with expected counts below five. Consequently, the chi-square findings for RQ1a and RQ1b are interpreted as descriptive and exploratory, highlighting general patterns in security vulnerability distributions by origin rather than providing definitive inferential evidence.

Comparison of Results to the Literature Review

The results of the current research can be related to several themes mentioned in chapter two, particularly those related to (a) the emergence of security vulnerabilities in both AI-generated and human-generated code, and (b) the use of structured risk models and tools of static analysis in the control of such security vulnerabilities. The literature alerted how the use of AI-generated code may bring about new or increased security threats. Previous researchers also pointed to the issue that LLMs can reproduce unsafe behavior, no longer understand what the user wants, and can amplify the effects of the security vulnerabilities (e.g., injection flaws, weak authentication, and poor error management) raised by numerous projects at scale (Asare et al., 2023; Cotroneo et al., 2025; Pearce et al., 2025; Sindhwad et al., 2024). Meanwhile, alternative literature proved that the code generated by humans is not much safer, as frequent weaknesses are caused by cognitive overload, time pressure, legacy, and unequal practices of secure coding (Fu et al., 2024; Gilstrap et al., 2024; Li, J. et al., 2024). The current outcomes give some depth

to this image. Rather than validating a plain statement: that AI-generated code is always less secure, the t-test revealed that in this case, human-generated code did in fact exhibit a higher mean CVSS-like severity score ($M = 4.01$, $SD = 2.85$) than AI-generated code ($M = 3.09$, $SD = 3.20$), and the difference between them was statistically significant, $t(244.52) = -2.47$, $p = .014$. Given that the corresponding effect size (Cohen's $d \approx -0.30$) was small to moderate, this result is compatible with literature that emphasizes the continued presence of serious weaknesses in human-generated code, while also indicating that, under the specific conditions and CVSS-like metric used in this study, AI-generated repositories did not exhibit systematically higher severity. This is consistent with the literature on the fact that human-generated code is still prone to grave errors, particularly when the practice of safe coding is not applied uniformly, yet it makes it more difficult to assume that AI-generated code is always the safer choice. Instead, the findings align with a more nuanced view in which both origins can contribute to security vulnerabilities, with the relative severity depending on context, tooling, and measurement.

The chi-square test showed that there was a significant relationship between origin (AI vs. human) and type of security vulnerability, χ^2 (43, $N = 272$) = 65.04, $p = .017$, with lots of cells having small, expected values. This helps the literature argument that AI- and human-generated code will have the tendency to create various types of security vulnerabilities. Some of the security vulnerabilities in AI-generated code outlined by previous researchers include data poisoning during LLMs training, loss of context, and the systematic repetition of dangerous patterns (Ambati et al., 2024; Cotroneo et al., 2025; Zywiolek, 2024), whereas the human-generated code is usually one-off mistakes, legacy decisions, and organizational pressures (Fu et al., 2024; Sindhwad et al., 2024). In the present study, this broad pattern was visible in that some categories (such as "no security vulnerabilities in this repo") occurred more

frequently in AI-generated repositories, while certain CWE groupings appeared relatively more often in human-generated code. Most importantly, chi-square findings of this present study are in line with that difference: origin seems to be relevant to what security vulnerabilities are expressed, despite the possibility of higher mean severity of human-generated code in this sample taken from the population of all public online repositories. At the same time, the comparison to literature must recognize that the original chi-square table in this study violated key assumptions due to sparse expected cell counts, and a follow-up chi-square on collapsed security vulnerability categories did not reach statistical significance, $\chi^2(2, N = 118) = 4.41$, $p = .110$. Consequently, the convergence with prior work on origin specific security vulnerability patterns is best regarded as descriptive and exploratory support, rather than as strong inferential confirmation of precise differences in the prevalence of individual CWE types.

The theoretical and practical frameworks mentioned in chapter two are also confirmed in this study. Intellectual capital theory (ICT) and the National Institute of Standards and Technology Risk Management Framework (NIST RMF) / National Institute of Standards and Technology Artificial Intelligence Risk Management Framework (NIST AI RMF) were introduced as means of realizing the harms caused by security vulnerabilities to the human capital, structural capital, and relational capital and how risk processes can be structured in order to reduce the harm (NIST, 2018; NIST, 2024; Sallos et al., 2019; Tabassi, 2023). Applying CodeQL as a static analysis tool and a uniform scheme of scoring security vulnerabilities akin to the CVSS-like score to similar security vulnerabilities in both AI and human repositories compare well to that framework. The security vulnerabilities were detected in a systematic way, compared against one another, and understood within the context of risk severity and patterns across groups. Thus, the empirical results are consistent with the ICT and NIST viewpoints that

security weaknesses in AI- and human-generated code regardless of origin must be monitored and compared using structured, repeatable processes that support risk informed decision making.

Lastly, the findings are relevant to the literature on the use of tools for static analysis, such as CodeQL. In chapter two, it was mentioned that CodeQL possesses strengths and weaknesses but should be thoroughly configured and interpreted and operated by humans (Alqaradaghi and Kozsik, 2024; Froberg, 2023; Youn et al., 2023). These practical limitations are reflected in the need to compute full CVSS v3.1 vectors when only the 'security_severerity' field of CodeQL needs to be constructed into a CVSS-like score. The necessity of constructing a study specific CVSS-like measure from CodeQL's severity vulnerability categories reflects these known constraints and underscores that the present severity vulnerability comparisons are grounded on a consistent, tool-based scale rather than in official CVSS v3.1 base scores. Nevertheless, the fact that CodeQL is consistently used in various languages and origins indicates that such tools can assist causal-comparative studies on the issue of AI-generated and human-generated security vulnerabilities in real-world repositories, which is one of the gaps found in literature.

Summary

This chapter provided the findings of a causal-comparative quantitative study of AI- and human-generated source code security vulnerabilities in CodeQL and a CVSS-like severity rating. Descriptive statistics, independent samples t-test, and chi square were used to answer three research questions by first cleaning and combining the dataset comprising of more than 134 public GitHub repositories and then performing the data analysis in SPSS. In case of RQ1, the independent sample t-test showed that the average severity among the samples in terms of the severity human-generated code and AI-generated code is statistically significantly different, with

the former showing higher average code of the type of a CVSS. In the case of RQ1a, a chi-square test of independence showed that there is a significant relationship between origin and security vulnerability, indicating that some groupings of security vulnerabilities vary between AI-generated and human-generated repositories, although a great number of categories have low expected counts. According to RQ1b, the cross-tabulation revealed that several security_vulnerability types, along with absence of security vulnerabilities found, existed in both AI-generated and human-generated code, which demonstrated common security vulnerabilities by origin.

These results, taken together, summarize how the three research questions were addressed: (a) RQ1 identified a statistically significant difference in mean CVSS-like severity scores between AI- and human-generated code, with human-generated repositories having higher average severity; (b) for RQ1a, the initial chi-square analysis suggested differences in the distribution of detailed security vulnerability categories by origin, but follow-up analyses with collapsed categories did not confirm a statistically significant association and were limited by sparse expected cell counts; and (c) RQ1b indicated that several security_vulnerability types, including the absence of detected security vulnerabilities, were observed in both AI-generated and human-generated repositories, demonstrating common security vulnerability patterns across origins. Collectively, these findings provide the empirical foundation for Chapter 5, where the results will be interpreted in greater depth in relation to the study's framework and prior research, and where implications and recommendations will be discussed.

Chapter 5: Discussion, Recommendations, and Study Summary

This quantitative causal-comparative study was aimed at investigating whether the AI-generated code and the human-generated code are different in terms of software security vulnerabilities when compared through the CodeQL and a CVSS-like severity score. Software development using LLMs is gaining more popularity, but the security considerations thereof are not clearly studied in comparison to software written in a more traditional manner. This study was dedicated to the comparison of AI-generated and human-generated repositories on Python, JavaScript/TypeScript, and Ruby, the detection of security vulnerabilities and their extent, and the description of the typical types of security vulnerabilities in the two categories of AI and human. This chapter is associated with the interpretation of the results in chapter four, explanation of the relatedness with the literature and theoretical framework, discussion of limitations of the research, and implications to practice and future research suggestions. The chapter ends by giving a summary of the contribution of the results to the knowledge of security vulnerabilities in AI-generated code compared to human-generated code.

Discussion

The findings of this quantitative causal-comparative research show how this sample of publicly available GitHub repositories, the average severity of security vulnerabilities in human-generated code was significantly higher than that in AI-generated code when evaluated by means of CodeQL and a CVSS-like severity score. Simultaneously, AI-generated and human-generated repositories both contained security vulnerabilities, and some AI-generated repositories had no security vulnerabilities whatsoever. Collectively, these results indicate that the simplistic picture of AI-generated code as more dangerous than human-generated code is not true and instead that two categories (AI and Human) of code represent a more nuanced security

posture where both types pose significant, however different, risks. Regarding the research questions, the independent samples t-test to answer RQ1 indicated that the mean security vulnerabilities severity of human generated repositories was significantly higher than that of AI-generated repositories. The chi-square outcomes of RQ1a also revealed that some of the categories of security vulnerabilities were related to the origin of the code, which suggests that there were some differences in patterns of security vulnerabilities between AI-generated and human-generated repositories. Lastly, the cross-tabulation of RQ1b revealed that the two origins had several similar security_vulnerabilities types, as well as instances where no security vulnerabilities were detected in either of the categories. These two findings together indicate that human-generated code is still a significant source of severe security vulnerabilities, and AI-generated code has a different, and also not less important, yet no longer trivial risk profile.

Several factors can have contributed to the interpretation of these results. To begin with, the research was based on a particular CodeQL setting and a CVSS-like severity score calculated off the security severity field and not a full CVSS v3.x vector, which could have constrained the level of severity ratings due to how it collected all programming language files in the particular GitHub repositories, instead of just capturing one record (aka programming language file); therefore, causing the original sample number to reach beyond 134 as stated in chapter 3. Second, the repositories were limited to a purposive sample of public GitHub projects based on languages and code origins, which might not be a complete representation of private, enterprise, or safety-critical systems. Third, difficulties with setting up and running CodeQL in sophisticated build systems, specifically some Java ecosystems, could have influenced the resulting data set or, consequently, the patterns of security vulnerabilities identified. These elements do not nullify the

results, but they imply that great care should be taken when extending the results to the repositories that were sampled.

Considered in the perspective of the Intellectual Capital Theory (ICT), the findings highlight that uncontrolled AI-generated or human-generated code may undermine human, structural, and relational capital. The more serious average severity of human-generated code in this sample can cause an organizational asset of knowledge and older systems to face serious, and in many cases caused by legacy, security vulnerabilities. Meanwhile, AI-generated code (even though less egregious), can reinforce vulnerable patterns at scope when training data or prompts are poisoned. This study validates the application of the NIST RMF and NIST AI RMF as advisory frameworks: both sets of code require a risk-based and lifecycle-focused approach, and the governance processes should not assume that any human or AI source is inherently safe.

The results are consistent with the existing literature on the topic that human developers systematically introduce security vulnerabilities because of cognitive constraints, time constraints, and legacy design problems, and AI-generated code may recreate insecure patterns or misinterpret contextual needs. Nevertheless, this study does not follow certain expectations found in the literature where AI-generated code is assumed to have higher or equal risk compared to human-generated code. In this studies sample, the security vulnerabilities of human-generated repositories were worse on average and AI-generated repositories were a little more likely to contain no vulnerabilities detected, where Python performed the best with little to no security vulnerabilities found. One viable reason is that LLMs can be more standardized in common tasks, perhaps also including best practices, whereas human developers must deal with older codebases, more complicated business logic, or time-limited conditions under which more

serious errors can be tolerated. This explanation fits in with the equivocal results in the previous studies, where human error and model constraints have been reported to be causing security risk.

These outcomes have both practical and conceptual societal implications. In practice, the researcher recommends that organizations, regulators, and software teams must not view AI-generated code as an exceptionally devastating threat and believe that traditional development methods are familiar and thus safe. Rather, it is the evidence that promotes a neutral security posture where AI-generated and human-generated code receive equivalent levels of scrutiny, statical examination, and governance. In conceptual terms, the results indicate that the use of LLMs in software development restructures, but not eliminate, the imperative to take care of risky software development, safe coding practices, and human controls as expressed in models like the NIST RMF and NIST AI RMF. In general, the paper explained the research problem through empirical comparisons of the risk of security vulnerabilities and the degree of risks in AI-generated and human-generated repositories, which gave a more solid basis of risk management decisions in the age of fast-growing AI-generated development.

Recommendations for Practice

This study presents several implications on organizations that are implementing LLMs. To begin with, the code written by humans in this sample had more severe averages as compared to that generated by AI, and both sources had security vulnerabilities. This implies that organizations are not supposed to represent AI-generated code as distinctly threatening and ignore the presence of long-term threats in conventional software development. Rather, security programs must use similar standards, audits and testing to AI-generated and human-generated code, as both may introduce security vulnerabilities that affect organizational risk and intellectual capital. Second, it has been established that there is a strong correlation between the

security_vulnerability type and the origin, which suggests that the teams might require various review checklists or code review heuristics to apply to the AI-generated and human-generated code. As an example, AI-generated code can be better checked around input validation and secure defaults as well as unnecessary code can be eliminated, whereas human-generated code might need special focus on legacy patterns, error handling, and advanced business logic. These patterns can be uncovered with the help of static analysis tools such as CodeQL, although this study also demonstrates that these tools are not easily ported into all environments, particularly complex Java ecosystems. Practitioners are likely to spend time and effort implementing CodeQL or similar systems into their build pipelines, stabilizing dependencies, and building analyses using various languages.

Recommendations for Future Research

This research can be expanded in several significant ways in future research. The logical extension here is a Java-based replication that provides the solution to the tooling issues in this case. A collection of Java repositories with known, non-SNAPSHOT build configurations (i.e. pinned non-SNAPSHOT dependencies, supported JDK versions, simplified Groovy use, etc.) could be curated by researchers, or an isolated and controlled dependency could be achieved by containerizing the build environment. Comparing CodeQL with other tools of static analysis that accept Java and JVM languages might also help to conclude whether similarity in AI-generated and human-generated security vulnerabilities patterns appears in such an ecosystem. Further research may perfect the severity measure by incorporating more enriched security vulnerability metadata, such as the full use CVSS v3.x. Longitudinal or experimental designs would be useful to study the pattern of security vulnerability over time as teams move towards the use of LLMs, the implementation of secure coding standards, or the implementation of automated policy

checks based on models such as the NIST RMF and NIST AI RMF. Lastly, more language-based research, varying repository selection strategies, or domain-related codebase (e.g., financial, healthcare, or critical infrastructure systems) would also help clarify the behavior of AI-generated and human-generated code in different risk, regulatory, or organizational contexts.

Study Summary

This research included a quantitative causal-comparison of AI-generated and human-generated code in three programming languages through CodeQL and a CVSS-like metric of severity with the aim of comparing the difference in security vulnerability severity and type. The findings revealed that human-generated code was more likely to be more severe as compared to AI-generated code, security_vulnerability types were linked to origin, and that multiple categories of security vulnerabilities such as undetected security vulnerabilities were shared between the two. These results dispel naive hypotheses that AI-generated code is necessarily more harmful but supports the necessity of a thorough security inspection of all code, with or without AI in the source. This work is a contribution to the empirical literature that is growing in AI-generated code and software security by recording the advantages and the practical limitations of applying CodeQL to real-world GitHub public repositories. The findings highlight that the LLMs alter the way the security vulnerabilities manifest, yet they do not eliminate the necessity of disciplined risk management, prudent governance and secure software engineering practices.

References

Aanby, J. L., Evensen, D., & Myklebust, V. (2024). *Bridging the gap between software security and development using CodeQL* [Bachelor thesis, NTNU]. https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/3138369

Aghakhani, H., Dai, W., Manoel, A., Fernandes, X., Kharkar, A., Kruegel, C., Vigna, G., Evans, D., Zorn, B., & Sim, R. (2024). TrojanPuzzle: Covertly poisoning code-suggestion models. *2024 IEEE Symposium on Security and Privacy (SP), Symposium on Security and Privacy (SP), 2024 IEEE, SP*, 1122–1140. https://doi.org/10.1109/SP54263.2024.00140

Alenezi, M., & Akour, M. (2025). AI-driven innovations in software engineering: A review of current practices and future directions. *Applied Sciences, 15*(3), 1344. https://doi.org/10.3390/app15031344

Alqaradaghi, M., & Kozsik, T. (2024). Comprehensive evaluation of static analysis tools for their performance in finding vulnerabilities in Java code. *IEEE Access, Access, IEEE*, *12*, 55824–55842. https://doi.org/10.1109/ACCESS.2024.3389955

Alqaradaghi, M., Morse, G., & Kozsik, T. (2022). Detecting security vulnerabilities with static analysis – A case study. *Pollack Periodica*, *17*(2), 1–7. https://doi.org/10.1556/606.2021.00454

Ambati, S. H., Ridley, N., Branca, E., & Stakhanova, N. (2024, July 8–12). Navigating (in)security of AI-generated code [Conference session]. *2024 IEEE International Conference on Cyber Security and Resilience (CSR)*, Venice, Italy. https://doi.org/10.1109/CSR61664.2024.10679468

Ampel, B., Samtani, S., Zhu, H., Ullman, S., & Chen, H. (2020). Labeling hacker exploits for proactive cyber threat intelligence: A deep transfer learning approach. *2020 IEEE International Conference on Intelligence and Security Informatics (ISI), Intelligence and Security Informatics (ISI), 2020 IEEE International Conference On*, 1–6. https://doi.org/10.1109/ISI49825.2020.9280548

Anu, V., Sultana, K. Z., & Samanthula, B. K. (2020). A human error-based approach to understanding programmer-induced software vulnerabilities. *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 49–54. https://doi.org/10.1109/ISSREW51248.2020.00036

Asare, O., Nagappan, M., & Asokan, N. (2023). Is GitHub's Copilot as bad as humans at introducing vulnerabilities in code? *Empirical Software Engineering: An International Journal*, *28*(6). https://doi.org/10.1007/s10664-023-10380-1

Balogh, Š., Mlynček, M., Vraňák, O., & Zajac, P. (2024). Using generative AI models to support cybersecurity analysts. *Electronics*, *13*(23), 4718. https://doi.org/10.3390/electronics13234718

Bannon, T. T., & Laplante, P. (2024). Generative AI in the software development lifecycle. *Computer*, *57*(12), 27–34. https://doi.org/10.1109/MC.2024.3474789

Benbya, H., Davenport, T. H., & Pachidi, S. (2020). Artificial intelligence in organizations: Current state and future opportunities. *MIS Quarterly Executive*, *19*(4). https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3741983

Bécue, A., Praça, I. & Gama, J. (2021). Artificial intelligence, cyber-threats and Industry 4.0: challenges and opportunities. *Artif Intell Rev 54*, 3849–3886. https://doi.org/10.1007/s10462-020-09942-2

Bertocchi, G., & Piamonte, A. (2023). From ICT framework compliance to quantitative risk assessment: An example of methodology using risk scenarios. *2023 AEIT International Annual Conference (AEIT), AEIT International Annual Conference (AEIT), 2023*, 1–6. https://doi.org/10.23919/AEIT60520.2023.10330350

Browning, J. W., Bustard, J., Anderson, N., & Galway, L. (2024). A Data Science Course Utilizing GenAI. *2024 IEEE Frontiers in Education Conference (FIE)*, 1–7. https://doi.org/10.1109/FIE61694.2024.10893452

Burlacu, F., & Luta, G.D.G (2023). The crucial importance of the European Union AI act as the world's first regulation on artificial intelligence. *Journal of Information Systems & Operations Management 17*(2), 59-76.

Calder, A. (2018). *NIST Cybersecurity Framework: A pocket guide*. IT Governance Publishing. https://doi.org/10.2307/j.ctv4cbhfx

Cheatham, B., Javanmardian, K., & Samandari, H. (2019). Confronting the risks of artificial intelligence. *McKinsey Quarterly*, *2*(38), 1-9.

https://www.sipotra.it/wp-content/uploads/2019/05/Confronting-the-risks-of-artificial-intelligence.pdf

Chowdhury, M. M., Rifat, N., Ahsan, M., Latif, S., Gomes, R., & Rahman, M. S. (2023). ChatGPT: A threat against the CIA triad of cybersecurity. *2023 IEEE International Conference on Electro Information Technology (EIT), Electro Information Technology (EIT), 2023 IEEE International Conference On*, 1–6. https://doi.org/10.1109/eIT57321.2023.10187355

Clark, A., Igbokwe, D., Ross, S., & Zibran, M. F. (2024). A quantitative analysis of quality and consistency in AI-generated code. *2024 7th International Conference on Software and System Engineering (ICoSSE)*, 37–41. https://doi.org/10.1109/ICoSSE62619.2024.00014

Cohen, J., Cohen, P., West, S. G., & Aiken, L. S. (2013). *Applied multiple regression/correlation analysis for the behavioral sciences*. R.,Cotroneo, D., Improta, C., Liguori, P., & Natella, R. (2024). Vulnerabilities in AI code generators: Exploring targeted data poisoning attacks. *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, 280–292. https://doi.org/10.1145/3643916.3644416

Cotroneo, D., De Luca, R., & Liguori, P. (2025). DeVAIC: A tool for security assessment of AI-generated code. *Information & Software Technology*, *177*, N.PAG-N.PAG. https://doi.org/10.1016/j.infsof.2024.107572

CVSS v3.1 Specification Document. (n.d.). FIRST — Forum of incident response and security teams. Retrieved March 29, 2025, from https://www.first.org/cvss/v3-1/specification-document

Dincă, A.-M., Axinte, S.-D., Tod-Raileanu, G., & Bacivarov, I. C. (2024). AI Tools introduced in Software Development. Analysis of Code quality, Security and Productivity Implications. *2024 IEEE 30th International Symposium for Design and Technology in Electronic Packaging (SIITME)*, 32–39. https://doi.org/10.1109/SIITME63973.2024.10814830

Delicheh, H. O., Decan, A., & Mens, T. (2024). Quantifying security issues in reusable JavaScript actions in GitHub workflows. *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR), Mining Software Repositories (MSR), 2024 IEEE/ACM 21st International Conference on, MSR*, 692–703.

Edkrantz, M., Truve, S., & Said, A. (2015). Predicting vulnerability exploits in the wild. *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing, Cyber Security and Cloud Computing (CSCloud), 2015 IEEE 2nd International Conference On*, 513–514. https://doi.org/10.1109/CSCloud.2015.56

Fröberg, T. (2023). Detection of prototype pollution using joern: Joern's detection capability compared to CodeQL's. *Detektering Av Prototypförorening Med Hjälp Av Joern : Joerns Detekteringsförmåga Jämfört Med CodeQL:S*.

Fountaine, T., McCarthy, B., & Saleh, T. (2019). Building the AI-powered organization. *Harvard business review*, *97*(4), 62-73.

https://wuyuansheng.com/doc/Databricks-AI-Powered-Org__Article-Licensing-July21-1.pdf

Fu, Y., Liang, P., Tahir, A., Li, Z., Shahin, M., Yu, J., & Chen, J. (2024). Security Weaknesses of Copilot-Generated Code in GitHub Projects: An Empirical Study. *ACM Transactions on Software Engineering and Methodology*. https://doi.org/10.1145/3716848

Gao, J., Choo, K. T. W., Cao, J., Lee, R. K.-W., & Perrault, S. (2023). CoAIcoder: Examining the effectiveness of AI-assisted human-to-human collaboration in qualitative analysis. *ACM Transactions on Computer-Human Interaction*, *31*(1), 6:1–6:38. https://doi.org/10.1145/3617362

Gilstrap, C., Bacic, D., & Gilstrap, C. (2024). Understanding the adoption of generative artificial intelligence within communities of practice: A cross-practice, machine learning-based lexical study. *2024 47th MIPRO ICT and Electronics Convention (MIPRO)*, 1367–1374. https://doi.org/10.1109/MIPRO60963.2024.10569536

Gillard, S., Percia David, D., Mermoud, A., & Maillart, T. (2023). Efficient collective action for tackling time-critical cybersecurity threats. *Journal of Cybersecurity*, *9*(1), tyad021.

Groß, J., & Möller, A. (2024). Some additional remarks on statistical properties of Cohen's d in the presence of covariates. *Statistical Papers*, *65*(6), 3971–3979. https://doi.org/10.1007/s00362-023-01527-9

Hadwick, D. (2024). Slipping through the cracks, the carve-outs for AI tax enforcement systems in the EU AI Act. *European Papers*, *2024 9*(3), 936–955. https://doi.org/10.15166/2499-8249/793

Hajipour, H., Hassler, K., Holz, T., Schonherr, L., & Fritz, M. (2024). CodeLMSec benchmark: systematically evaluating and finding security vulnerabilities in black-box code language models. *2024 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML), Secure and Trustworthy Machine Learning (SaTML), 2024 IEEE Conference on, SATML*, 684–709. https://doi.org/10.1109/SaTML59370.2024.00040

Hamer, S., d'Amorim, M., & Williams, L. (2024, May 20–23). Just another copy and paste? Comparing the security vulnerabilities of ChatGPT generated code and StackOverflow answers [Workshop presentation]. *2024 IEEE Security and Privacy Workshops (SPW)*, San Francisco, CA, United States. https://doi.org/10.1109/SPW63631.2024.00014

Hussain, A., Rabin, M. R. I., & Alipour, M. A. (2024, January 15–18). Measuring impacts of poisoning on model parameters and embeddings for large language models of code [Conference presentation]. *Proceedings of the 1st ACM International Conference on AI-Powered Software*, San Francisco, CA, United States, 59–64. https://doi.org/10.1145/3664646.3664764

Idrisov, B., & Schlippe, T. (2024). Program code generation with generative AIs. *Algorithms*, *17*(2), 62. https://doi.org/10.3390/a17020062

Imtiaz, N., & Williams, L. (2023). Are your dependencies code reviewed?: Measuring code review coverage in dependency updates. *IEEE Transactions on Software Engineering, Software Engineering, IEEE Transactions on, IIEEE Trans. Software Eng.*, *49*(11), 4932–4945. https://doi.org/10.1109/TSE.2023.3319509

Iwanaga, K., Lee, D., Hamburg, J., Wu, J.-R., Chen, X., Rumrill, P., Wehman, P., Tansey, T. N., & Chan, F. (2023). Effects of supported employment on the competitive integrated employment outcomes of transition age and young adults with intellectual disabilities: A non-experimental causal comparative study. *Journal of Vocational Rehabilitation*, *58*(1), 39–48. https://doi.org/10.3233/JVR-221223

JD, S., & Jr, R. P. (2004). Causal-comparative research designs. *Journal of Vocational Rehabilitation*, *21*(3), 117–121.

Jian Jim Hu, Qiaoyan Wen, & Ai Fen Sui. (2011). An efficient code audit method for accurately detecting security vulnerabilities in source codes. *2011 IEEE 13th International Conference on Communication Technology, Communication Technology (ICCT), 2011 IEEE 13th International Conference On*, 698–702. https://doi.org/10.1109/ICCT.2011.6157966

Kamaja, P., Ruohonen, M., Löytty, K., & Ingalsuo, T. (2016). Intellectual capital based evaluation framework for dynamic distributed software development. *Electronic Journal of Knowledge Management: EJKM*, *14*(4), 231–244.

Kaniewski, S., Holstein, D., Schmidt, F., & Heer, T. (2024). Vulnerability handling of AI-generated code—existing solutions and open challenges. *2024 Conference on AI,*

Science, Engineering, and Technology (AIxSET), AI, Science, Engineering, and Technology (AIxSET), 2024 Conference on, AIXSET, 145–148. https://doi.org/10.1109/AIxSET62544.2024.00026

Kapakos, W. A., & Fulk, H. K. (2024). GitHub Copilot: Introducing the artificial intelligence tool in an information systems course. *Issues in Information Systems*, *25*(4), 106–117. https://doi.org/10.48009/4_iis_2024_108

Kirk, H. R., Vidgen, B., Röttger, P., & Hale, S. A. (2024). The benefits, risks and bounds of personalizing the alignment of large language models to individuals. *Nature Machine Intelligence, 6*(4), 383-392. https://doi.org/10.1038/s42256-024-00820-y

Kotb, H. M., Gaber, T., AlJanah, S., Zawbaa, H. M., & Alkhathami, M. (2025). A novel deep synthesis-based insider intrusion detection (DS-IID) model for malicious insiders and AI-generated threats. *Scientific Reports, 15*(1), 207. https://doi.org/10.1038/s41598-024-84673-w

Lahiri, S., & Saltz, J. (2024). The need for a risk management framework for data science projects: a systematic literature review. *International Journal of Information Systems and Project Management, 12*(4), 41-57. https://doi.org/10.12821/ijispm120403

Lertbanjongngam, S., Chinthanet, B., Ishio, T., Kula, R. G., Leelaprute, P., Manaskasemsak, B., Rungsawang, A., & Matsumoto, K. (2022). An empirical evaluation of competitive programming AI: A case study of AlphaCode. *2022 IEEE 16th International Workshop on Software Clones (IWSC)*, 10–15. https://doi.org/10.1109/IWSC55060.2022.00010

Li, J., Sangalay, A., Cheng, C., Tian, Y., & Yang, J. (2024, May 10–12). Fine-tuning large language models for secure code generation [Conference presentation]. *2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*

(FORGE), San Francisco, CA, United States, 86–90. https://doi.org/10.1145/3650105.3652299

Li, K., Hong, S., Fu, C., Zhang, Y., & Liu, M. (2023). Discriminating Human-authored from ChatGPT-generated code via discernable feature analysis. *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW), Software Reliability Engineering Workshops (ISSREW), 2023 IEEE 34th International Symposium on, ISSREW*, 120–127. https://doi.org/10.1109/ISSREW60843.2023.00059

Li, Z., Liu, Z., Wong, W. K., Ma, P., & Wang, S. (2024). Evaluating C/C++ vulnerability detectability of query-based static application security testing tools. *IEEE Transactions on Dependable and Secure Computing, Dependable and Secure Computing, IEEE Transactions on, IEEE Trans. Dependable and Secure Comput*, *21*(5), 4600–4618. https://doi.org/10.1109/TDSC.2024.3354789

Liu, Y., Le-Cong, T., Widyasari, R., Tantithamthavorn, C., Li, L., Le, X.-B. D., & Lo, D. (2024). Refining ChatGPT-generated code: Characterizing and mitigating code quality issues. *ACM Transactions on Software Engineering and Methodology*, *33*(5), 1–26. https://doi.org/10.1145/3643674

McFee, I. (2024). Feature article: How GenAI will change the world economy. *Wiley.* https://doi.org/10.1111/1468-0319.12767

Majdinasab, V., Bishop, M. J., Rasheed, S., Moradidakhel, A., Tahir, A., & Khomh, F. (2024). Assessing the security of GitHub Copilot's generated code—A targeted replication study. *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Software Analysis, Evolution and Reengineering (SANER), 2024 IEEE*

International Conference on, SANER, 435–444. https://doi.org/10.1109/SANER60148.2024.00051

Milovidov, S. (2024). Content policy and access limitations on commercial neural networks as an incentive to artivism. *Artnodes*, *33*, 1–9. https://doi.org/10.7238/artnodes.v0i33.417696

MITRE. (n.d.). *CVE – Common Vulnerabilities and Exposures (CVE)*. CVE.org. Retrieved January 28, 2025, from https://www.cve.org

National Institute of Standards and Technology (US). (2024). *Artificial intelligence risk management framework: Generative artificial intelligence profile* (No. error: 600-1; p. error: 600-1). National Institute of Standards and Technology (U.S.). https://doi.org/10.6028/NIST.AI.600-1

National Institute of Standards and Technology. (2018). *Framework for improving critical infrastructure cybersecurity (Version 1.1)*. U.S. Department of Commerce. https://doi.org/10.6028/NIST.CSWP.04162018

Negri-Ribalta, C., Geraud-Stewart, R., Sergeeva, A., & Lenzini, G. (2024). A systematic literature review on the impact of AI models on the security of code generation. *Frontiers in Big Data*, *7*, 1386720. https://doi.org/10.3389/fdata.2024.1386720

Neumann, P. g., Bishop, M., Peisert, S., & Schaefer, M. (2010). Reflections on the 30th anniversary of the IEEE symposium on security and privacy. *2010 IEEE Symposium on Security and Privacy, Security and Privacy (SP), 2010 IEEE Symposium On*, 3–13. https://doi.org/10.1109/SP.2010.43

Ng, A. W. (2006). Reporting intellectual capital flow in technology‐based companies: Case studies of Canadian wireless technology companies. *Journal of Intellectual Capital*, *7*(4), 492–510. https://doi.org/10.1108/14691930610709130

Nofal, A. B., Ali, H., Hadi, M., Ahmad, A., Qayyum, A., Johri, A., Al-Fuqaha, A., & Qadir, J. (2025). AI-enhanced interview simulation in the metaverse: Transforming professional skills training through VR and generative conversational AI. *Computers and Education: Artificial Intelligence*, *8*, 100347. https://doi.org/10.1016/j.caeai.2024.100347

Noiklueb, C., Boonlue, S., & Srikaew, D. (2025). Development of cyber wellness indicators among Thai older persons: Mixed method research. *Educational Gerontology*, *51*(5), 501–517. https://doi.org/10.1080/03601277.2024.2401840

Nørbjerg, J., & Dittrich, Y. (2024). The never-ending story–how companies transition to and sustain continuous software engineering practices. *The Journal of Systems & Software, 213*. https://doi.org/10.1016/j.jss.2024.112056

Ogundare, O., Araya, G. Q., & Qamsane, Y. (2022). No code AI: Automatic generation of function block diagrams from documentation and associated heuristic for context-aware ML algorithm training. *2022 7th International Conference on Mechanical Engineering and Robotics Research (ICMERR), Mechanical Engineering and Robotics Research (ICMERR), 2022 7th International Conference On*, 191–195. https://doi.org/10.1109/ICMERR56497.2022.10097820

Oh, S., Lee, K., Park, S., Kim, D., & Kim, H. (2024). Poisoned ChatGPT finds work for idle hands: Exploring developers' coding practices with insecure suggestions from poisoned AI models. *2024 IEEE Symposium on Security and Privacy (SP), Symposium on Security and Privacy (SP), 2024 IEEE, SP*, 1141–1159. https://doi.org/10.1109/SP54263.2024.00046

Okey, O. D., Udo, E. U., Rosa, R. L., Rodríguez, D. Z., & Kleinschmidt, J. H. (2023). Investigating ChatGPT and cybersecurity: A perspective on topic modeling and sentiment

analysis. *Computers & Security*, *135*, N.PAG-N.PAG. https://doi.org/10.1016/j.cose.2023.103476

OpenAI. (2024). *Responsible use of GitHub Copilot Chat in your IDE*. GitHub Docs. https://docs.github.com/en/copilot/responsible-use-of-github-copilot-features/responsible-use-of-github-copilot-chat-in-your-ide

Patel, A., Sultana, K. Z., & Samanthula, B. K. (2024). A comparative analysis between AI generated code and human written code: A preliminary study. *2024 IEEE International Conference on Big Data (BigData)*, 7521–7529. https://doi.org/10.1109/BigData62323.2024.10825958

Patel, D., Patel, H., Sultana, K. Z., & Anu, V. (2023). Programmer cognition failures as the root cause of software vulnerabilities: A preliminary review. *2023 Intermountain Engineering, Technology and Computing (IETC)*, 242–246. https://doi.org/10.1109/IETC57902.2023.10152150

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2025). Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. *Communications of the ACM*, *68*(2), 96–105. https://doi.org/10.1145/3610721

Petersen, A. H., Ekstrøm, C. T., Spirtes, P., & Osler, M. (2023). Constructing causal life-course models: Comparative study of data-driven and theory-driven approaches. *American Journal of Epidemiology*, *192*(11), 1917–1927. https://doi.org/10.1093/aje/kwad144

Pham, N., Pham Ngoc, H., & Nguyen-Duc, A. (2025). Fairness for machine learning software in education: A systematic mapping study. *Journal of Systems & Software*, *219*, N.PAG-N.PAG. https://doi.org/10.1016/j.jss.2024.112244

Piper, D. (2023). Regulatory update NIST publishes artificial intelligence risk management framework and resources. *Journal of Internet Law*, *26*(6), 1–14.

Qadir, J. (2023). Engineering education in the era of ChatGPT: promise and pitfalls of generative AI for education. *2023 IEEE Global Engineering Education Conference (EDUCON)*, 1–9. https://doi.org/10.1109/EDUCON54358.2023.10125121

Roy, P. P. (2020). A high-level comparison between the NIST Cyber Security Framework and the ISO 27001 Information Security Standard. *2020 National Conference on Emerging Trends on Sustainable Technology and Engineering Applications (NCETSTEA), Sustainable Technology and Engineering Applications (NCETSTEA), 2020 National Conference on Emerging Trends On*, 1–3. https://doi.org/10.1109/NCETSTEA48365.2020.9119914

Salihu, A., & Dervishi, R. (2024). Evaluating the impact of risk management frameworks on IT Audits: A comparative analysis of COSO, COBIT, ISO/IEC 27001, and NIST CSF. *2024 International Conference on Electrical, Communication and Computer Engineering (ICECCE), Electrical, Communication and Computer Engineering (ICECCE), 2024 International Conference On*, 1–8. https://doi.org/10.1109/ICECCE63537.2024.10823548

Sallos, M. P., Garcia-Perez, A., Bedford, D., & Orlando, B. (2019). Strategy and organisational cybersecurity: A knowledge-problem perspective. *Journal of Intellectual Capital*, *20*(4), 581–597. https://doi.org/10.1108/JIC-03-2019-0041

Seghier, M.L. (2025). AI-powered peer review needs human supervision. *Journal of Information, Communication and Ethics in Society*, Vol. 23 No. 1, pp. 104-116. https://doi.org/10.1108/JICES-09-2024-0132

Sindhwad, P. V., Ranka, P., Muni, S., & Kazi, F. (2024). VulnArmor: Mitigating software vulnerabilities with code resolution and detection techniques. *International Journal of Information Technology: An Official Journal of Bharati Vidyapeeth's Institute of Computer Applications and Management*, 1–16. https://doi.org/10.1007/s41870-024-01775-4

Sllame, A. M., Tomia, T. E., & Rahuma, R. M. (2024). A holistic approach for cyber security vulnerability assessment based on open source tools: Nikto, acunitx, ZAP, nessus and enhanced with AI-powered tool ImmuniWeb. *2024 IEEE 4th International Maghreb Meeting of the Conference on Sciences and Techniques of Automatic Control and Computer Engineering (MI-STA), Sciences and Techniques of Automatic Control and Computer Engineering (MI-STA), 2024 IEEE 4th International Maghreb Meeting of the Conference On*, 68–75. https://doi.org/10.1109/MI-STA61267.2024.10599685

Suneja, S., Zheng, Y., Zhuang, Y., Laredo, J. A., & Morari, A. (2021, October 19–21). Towards reliable AI for source code understanding. *Proceedings of the ACM Symposium on Cloud Computing* (pp. 403–411). Seattle, WA, United States. https://doi.org/10.1145/3472883.3486995

Swaminathan, N., & Danks, D. (2024). Governing ethical gaps in distributed AI development. *Digital Society: Ethics, Socio-Legal and Governance of Digital Technology*, *3*(1). https://doi.org/10.1007/s44206-024-00088-0

Tabassi, E. (2023). *Artificial intelligence risk management framework (AI RMF 1.0)* (No. NIST AI 100-1; p. NIST AI 100-1). National Institute of Standards and Technology (U.S.). https://doi.org/10.6028/NIST.AI.100-1

Taghavi Far, S. M., & Feyzi, F. (2025). Large language models for software vulnerability detection: a guide for researchers on models, methods, techniques, datasets, and metrics. *International Journal of Information Security*, *24*(2). https://doi.org/10.1007/s10207-025-00992-7

Tao, N., Ventresque, A., Nallur, V., & Saber, T. (2024). Enhancing program synthesis with large language models using many-objective grammar-guided genetic programming. *Algorithms*, *17*(7), 287. https://doi.org/10.3390/a17070287

Tosi, D. (2024). Studying the quality of source code generated by different AI generative engines: An empirical evaluation. *Future Internet*, *16*(6), 188. https://doi.org/10.3390/fi16060188

Wang, J., Luo, X., Cao, L., He, H., Huang, H., Xie, J., Jatowt, A., & Cai, Y. (2024). Is your AI-generated code really safe? Evaluating large language models on secure code generation with CodeSecEval.

Wehmeier, L., Eilermann, S., Niggemann, O., & Deuter, A. (2023). Task-fidelity Assessment for Programming Tasks Using Semantic Code Analysis. *2023 IEEE Frontiers in Education Conference (FIE)*, 1–5. https://doi.org/10.1109/FIE58773.2023.10342916

Wohlin, C., Šmite, D., & Moe, N. B. (2015). A general theory of software engineering: Balancing human, social and organizational capitals. *Journal of Systems and Software*, *109*, 229–242. https://doi.org/10.1016/j.jss.2015.08.009

Xchain Analytics Ltd (Director). (2023, January 1). *Craft an AutoGPT code generation AI instrument leveraging rust and GPT-4* [Video recording]. Packt Publishing.

Xue, J., Yu, Z., Song, Y., Qin, Z., Sun, X., & Wang, W. (2023). VulSAT: Source code vulnerability detection scheme based on SAT structure. *2023 8th International Conference on Signal and Image Processing (ICSIP)*, 639–644. https://doi.org/10.1109/ICSIP57908.2023.10271020

Yetistiren, B., Ozsoy, I., & Tuzun, E. (2022). Assessing the quality of GitHub copilot's code generation. *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, 62–71. https://doi.org/10.1145/3558489.3559072

Yigit, Y., Ferrag, M. A., Ghanem, M. C., Sarker, I. H., Maglaras, L. A., Chrysoulas, C., Moradpoor, N., Tihanyi, N., & Janicke, H. (2025). Generative AI and LLMs for critical infrastructure protection: Evaluation benchmarks, Agentic AI, challenges, and opportunities. *Sensors (14248220)*, *25*(6), 1666. https://doi.org/10.3390/s25061666

Youn, D., Lee, S., & Ryu, S. (2023). Declarative static analysis for multilingual programs using CodeQL. *Software: Practice & Experience*, *53*(7), 1472–1495. https://doi.org/10.1002/spe.3199

Żywiołek, J. (2024). Empirical examination of AI-powered decision support systems: Ensuring trust and transparency in information and knowledge security. *Scientific Papers of Silesian University of Technology. Organization & Management / Zeszyty Naukowe Politechniki Slaskiej. Seria Organizacji i Zarzadzanie*, *197*, 679–695. https://doi.org/10.29119/1641-3466.2024.197.37

Appendix A

Search Terms

Artificial Intelligence

- AB "artificial intelligence" AND AB (cod* n1 (techniques or tools or methods or generat*))
- (security N3 (vulnerabilities or risks)) AND ((llms or "large language models" or "generative AI"))
- AB (security N3 (vulnerabilities or risks)) AND AB ((llms or "large language models" or "generative AI"))
- AB (security N3 (vulnerabilities or risks)) AND AB ((llms or "large language models"))
- AB (security N3 (vulnerabilities or risks)) AND AB "AI generated code"
- SU National Institute of Standards & Technology (U.S.) AND TX ((rmf or "risk management framework" or ai or "artificial intelligence" or LLMs or "large language models"))
- SU National Institute of Standards & Technology (U.S.) AND SU ARTIFICIAL intelligence
- SU (National Institute of Standards and Technology) AND SU ARTIFICIAL intelligence
- SU NIST AI Risk Management Framework (AI RMF)
- ((NIST or "national institute of standards and technology")) AND ((rmf or "risk management framework" or ai or "artificial intelligence" or LLMs or "large language models")) AND TX predictive
- ((NIST or "national institute of standards and technology")) AND ((rmf or "risk management framework" or ai or "artificial intelligence" or LLMs or "large language models")) AND TX quantitative
- ((NIST or "national institute of standards and technology")) AND ((rmf or "risk management framework" or ai or "artificial intelligence" or LLMs or "large language models"))
- AB ((NIST or "national institute of standards and technology")) AND AB ((rmf or "risk management framework" or ai or "artificial intelligence" or LLMs or "large language models"))
- AB ((NIST or "national institute of standards and technology")) AND AB ((rmf or "risk management framework" or ai or "artificial intelligence"))
- AB ((NIST or "national institute of standards and technology") N5 (rmf or "risk management framework" or ai or "artificial intelligence"))
- AB ((NIST or "national institute of standards and technology") N2 (rmf or "risk management framework" or ai or "artificial intelligence"))
- TX ((NIST or "national institute of standards and technology") N2 (rmf or "risk management framework" or ai or "artificial intelligence"))

- TX ((NIST or "national institute of standards and technology") N2 (rmf or "risk management framework"))
- TX NIST N2 rmf
- TX "NIST rmf"
- (NIST AI RMF 1.0) OR (NIST AI 100-1) OR (NIST AI Risk Management Framework)
- (“NIST RMF” OR “NIST Risk Management Framework”)
- (“NIST RMF” OR “NIST Risk Management Framework”) AND (generative AI)
- ai n4 code AND (security or privacy or confidentiality)
- ai generated code AND TX (security or privacy or confidentiality)
- NIST RMF OR (NIST N10 (RMF OR risk management framework)

Google Scholar Searches:

- "Cited by" and "Related articles" search for Github copilot ai pair programmer: Asset or liability?
- Advanced Search for NIST AI OR "artificial intelligence" "risk management framework"
- Advanced Search for (NIST AI RMF 1.0) OR (NIST AI 100-1) OR (NIST AI Risk Management Framework)
- Advanced Search for (“NIST RMF” OR “NIST Risk Management Framework”)
- Advanced Search for (“NIST RMF” OR “NIST Risk Management Framework”) AND (generative AI)
- Advanced Search for (generative AI OR chatgpt) AND (code development OR code writing OR software development)
- Advanced Search for (generative AI OR chatgpt) AND Github
- Advanced Search for ((NIST AI RMF 1.0) OR (NIST AI 100-1) OR (NIST AI Risk Management Framework)) AND (code development OR software development)
- Basic Search for risk of ai generated codes

Keywords/phrases:

NIST RMF
NIST
RMF
national institute of standards and technology
risk management framework
AI RMF
AI
artificial intelligence
LLMs
large language models
generative AI
AI generated code
quantitative
predictive
security vulnerabilities

software development
vulnerabilities
risks
security
cybersecurity
danger
harm
ChatGPT
code development
code writing
software development
CoPilot

Subject terms:

NIST AI Risk Management Framework (AI RMF)
National Institute of Standards and Technology
artificial intelligence
security
AI-generated code
AI-based code generators
cryptography and security
AI governance
Intellectual Capital Theory (ICT)

Appendix B

CodeQL Code

```
#!/usr/bin/env python3[1]

"""

Universal CodeQL Scanner (Java, Python, JavaScript/TypeScript, Ruby) with:

- CodeQL scanning

- Origin labeling (ai / human / unknown via repo_labels.json)

- Language labeling

- SARIF merge

- CVSS-like severity scoring derived from CodeQL metadata

- Per-repo CSV summary for data analysis

- Master CSV that aggregates results across repos

- Master summary CSV that counts repos by language & origin (ai/human)

- Skip-rescan logic: if repo is already in master CSV, skip unless --force-rescan


IMPORTANT:

- The cvss_score in this script is a CVSS-inspired heuristic based on

  CodeQL's security-severity, level, and precision.

  It is NOT an official CVSS v3.1 base score.


Outputs (per repo):

  - Combined SARIF: <outdir>/codeql-results.sarif

      - Each result gets result.properties["cvss_score"] (0–10)

  - CSV summary:    <outdir>/codeql-results-summary.csv
```

[1] The validity of this entire python script was code reviewed by my son, Darryl Tyler Palmer, who is an expert C++ and Python software developer employed by StrataCIO in the roles of Chief Information Officer (CIO) and Chief Technology Officer (CTO).

```
Global:
  - Master CSV:   C:\\codeql-study\\.codeql-out\\master-results.csv
    (append-only; header written once)
  - Summary CSV:  C:\\codeql-study\\.codeql-out\\master-summary.csv
    (overwritten each run with fresh counts)

Master summary counts:
  - total_repos_scanned
  - java_ai_repos, java_human_repos
  - python_ai_repos, python_human_repos
  - javascript_typescript_ai_repos, javascript_typescript_human_repos
  - ruby_ai_repos, ruby_human_repos

CSV columns (per-repo and master):
  repo, origin, language,
  security_severity, level, precision,
  security_vulnerability, cvss_score

If a repo has no CodeQL results, one CSV row is emitted with:
  security_vulnerability = "No security vulnerabilities for this repo"
  cvss_score = 0

Requires:
  - CodeQL CLI on PATH (or pass --codeql)
"""

import argparse
import csv
```

```
import json
import os
import subprocess
import sys
from pathlib import Path
from typing import Any, Dict, List, Optional

# ----------------------------------------------------------------------
# Config: labels for AI / human / unknown
# ----------------------------------------------------------------------

LABELS_FILE = Path(__file__).with_name("repo_labels.json")

def get_repo_origin(repo_path: Path) -> str:
    """
    Return 'ai', 'human', or 'unknown' based on repo_labels.json.

    repo_labels.json should look like:

    {
      "Repo1": "ai",
      "Repo2": "human"
    }

    Keys must match repo folder names (Path(repo).name).
    """
    labels: Dict[str, str] = {}
    try:
```

```
        if LABELS_FILE.exists():
            with open(LABELS_FILE, "r", encoding="utf-8") as f:
                labels = json.load(f)
        else:
            print(f"[info] No {LABELS_FILE.name} found; all repos will be 'unknown' origin.")
    except Exception as e:
        print(f"[warn] Could not read {LABELS_FILE}: {e}", file=sys.stderr)

    name = repo_path.name
    return labels.get(name, "unknown")

# ---------------------------------------------------------------------
# Small helpers
# ---------------------------------------------------------------------

def run(cmd: List[str], cwd: Optional[Path] = None, check: bool = True) -> None:
    """Run a subprocess, echoing the the command and raising on failure if check=True."""
    print(f"[cmd] {' '.join(str(c) for c in cmd)}")
    result = subprocess.run(
        cmd,
        cwd=str(cwd) if cwd else None,
    )
    if check and result.returncode != 0:
        raise SystemExit(f"Command failed with exit code {result.returncode}: {' '.join(str(c) for c in cmd)}")

def load_json(path: Path) -> Any:
    with open(path, "r", encoding="utf-8") as f:
        return json.load(f)
```

```
def save_json(path: Path, obj: Any) -> None:
    path.parent.mkdir(parents=True, exist_ok=True)
    with open(path, "w", encoding="utf-8") as f:
        json.dump(obj, f, indent=2)

# ----------------------------------------------------------------------
# Language detection
# ----------------------------------------------------------------------

LANG_EXTENSIONS: Dict[str, List[str]] = {
    "java": [".java"],
    "python": [".py"],
    "javascript": [".js", ".jsx", ".ts", ".tsx"],
    "ruby": [".rb"],
}

def detect_languages(source: Path) -> List[str]:
    """Detect which of the supported languages exist in the repo."""
    found = set()
    for root, dirs, files in os.walk(source):
        # Skip typical non-source directories
        dirs[:] = [
            d for d in dirs
            if d not in {".git", ".hg", ".svn", ".codeql-db", ".codeql-out"}
        ]
        for fn in files:
            ext = Path(fn).suffix.lower()
```

```
        for lang, exts in LANG_EXTENSIONS.items():
            if ext in exts:
                found.add(lang)
    return sorted(found)


# ---------------------------------------------------------------------
# CodeQL database & analysis
# ---------------------------------------------------------------------


def java_build_command(source: Path) -> List[str]:
    """
    Prefer Maven, then Gradle wrapper, then Gradle.
    If no standard build config is found, return [] and let CodeQL's
    default behavior/autobuild handle it.
    """
    is_windows = (os.name == "nt")

    # 1) Maven
    if (source / "pom.xml").exists():
        return ["--command", "mvn -q -DskipTests package"]

    # 2) Gradle wrapper (gradlew / gradlew.bat)
    gradlew = source / "gradlew"
    gradlew_bat = source / "gradlew.bat"
    if gradlew.exists() or gradlew_bat.exists():
        if is_windows:
            # On Windows, prefer gradlew.bat if present
            if gradlew_bat.exists():
```

```
            return ["--command", "gradlew.bat -q assemble"]
        else:
            # Some projects only have "gradlew" but still work as a .bat
            return ["--command", "gradlew -q assemble"]
    else:
        # On Unix-like systems, ./gradlew is correct
        return ["--command", "./gradlew -q assemble"]

    # 3) Plain Gradle
    if (source / "build.gradle").exists() or (source / "settings.gradle").exists():
        return ["--command", "gradle -q assemble"]

    # 4) No build config we recognize → let CodeQL handle it (no extra args)
    return []

def suite_for(lang: str) -> str:
    """Return the extended security suite for the given language."""
    return {
        "java":       "codeql/java-queries:codeql-suites/java-security-extended.qls",
        "python":     "codeql/python-queries:codeql-suites/python-security-extended.qls",
        "javascript": "codeql/javascript-queries:codeql-suites/javascript-security-extended.qls",
        "ruby":       "codeql/ruby-queries:codeql-suites/ruby-security-extended.qls",
    }[lang]

def analyze_language(codeql: str, db_path: Path, lang: str, out_dir: Path) -> Path:
    """Run CodeQL analysis for a single language and emit SARIF."""
    suite = suite_for(lang)
    out_sarif = out_dir / f"{lang}.sarif"
```

```
    # Ensure the query pack is available (no-op if already cached)
    run([codeql, "pack", "download", suite.split(":")[0]], check=True)

    # Analyze
    run(
        [
            codeql, "database", "analyze", str(db_path),
            suite,
            "--format=sarifv2.1.0",
            "--output", str(out_sarif),
            "--rerun",
        ],
        check=True,
    )
    return out_sarif

# ---------------------------------------------------------------------
# CVSS-like scoring helpers
# ---------------------------------------------------------------------

def map_level_to_impact(level: str) -> float:
    """Conservative fallback mapping from CodeQL level -> impact (0–10)."""
    mapping = {"error": 8.0, "warning": 5.0, "note": 2.0}
    return mapping.get((level or "").lower(), 5.0)

def map_precision_to_exploitability(precision: str) -> float:
    """Map CodeQL precision -> exploitability (0–10)."""
```

```
    if not precision:
        return 5.0
    p = precision.lower()
    if "high" in p:
        return 8.0
    if "medium" in p:
        return 5.0
    if "low" in p:
        return 3.0
    return 5.0


def compute_cvss_like(impact: float, exploitability: float) -> int:
    """
    CVSS-like Score = { 0, if Impact = 0;
                        min(round(0.6 × Impact + 0.4 × Exploitability − 1.5), 10), otherwise }

    This is a heuristic derived from CVSS ideas, NOT an official CVSS v3.1 computation.
    """
    if impact <= 0:
        return 0
    score = 0.6 * impact + 0.4 * exploitability - 1.5
    score = round(score)
    if score > 10:
        score = 10
    if score < 0:
        score = 0
    return score
```

```
# ----------------------------------------------------------------------
# Skip-rescan helper: check if repo already in master CSV
# ----------------------------------------------------------------------

def repo_already_in_master(repo_name: str, master_csv: Path) -> bool:
    """
    Return True if the given repo name is already present in the master-results.csv file.

    We only look at the 'repo' column, ignoring language/origin. If the repo ever
    had any rows in the master file, we treat it as already scanned.
    """
    if not master_csv.exists():
        return False

    with open(master_csv, "r", encoding="utf-8", newline="") as f:
        reader = csv.DictReader(f)
        for row in reader:
            repo = (row.get("repo") or "").strip()
            if repo == repo_name:
                return True
    return False

# ----------------------------------------------------------------------
# Master summary updater (repo counts by language & origin)
# ----------------------------------------------------------------------

def update_master_summary(master_csv: Path, summary_path: Path) -> None:
    """
```

```
Read the master-results.csv, compute repo-level counts by language & origin,
and write them to master-summary.csv.

Counting logic:
  - Use unique (repo, origin, language) combinations from the master file.
  - total_repos_scanned = count of unique repo names across all languages/origins.
  - java_ai_repos                    = repos where language == 'java'       and origin == 'ai'
    java_human_repos                 = repos where language == 'java'       and origin == 'human'
    python_ai_repos                  = repos where language == 'python'     and origin == 'ai'
    python_human_repos               = repos where language == 'python'     and origin == 'human'
    javascript_typescript_ai_repos    = repos where language == 'javascript' and origin == 'ai'
    javascript_typescript_human_repos = repos where language == 'javascript' and origin == 'human'
    ruby_ai_repos                    = repos where language == 'ruby'       and origin == 'ai'
    ruby_human_repos                 = repos where language == 'ruby'       and origin == 'human'
"""
if not master_csv.exists():
    print(f"[summary] Master CSV {master_csv} does not exist yet; skipping summary update.")
    return

unique_repo_lang_origin = set()

with open(master_csv, "r", encoding="utf-8", newline="") as f:
    reader = csv.DictReader(f)
    for row in reader:
        repo = (row.get("repo") or "").strip()
        origin = (row.get("origin") or "").strip().lower()
        lang = (row.get("language") or "").strip().lower()
        if not repo:
```

```
            continue
        unique_repo_lang_origin.add((repo, origin, lang))

    if not unique_repo_lang_origin:
        print("[summary] No rows in master CSV yet; nothing to summarize.")
        return

    # Unique repos overall
    repos = {repo for (repo, _origin, _lang) in unique_repo_lang_origin}
    total_repos_scanned = len(repos)

    def count(lang: str, origin: str) -> int:
        return sum(
            1
            for (_repo, o, l) in unique_repo_lang_origin
            if l == lang and o == origin
        )

    java_ai = count("java", "ai")
    java_human = count("java", "human")
    python_ai = count("python", "ai")
    python_human = count("python", "human")
    js_ai = count("javascript", "ai")
    js_human = count("javascript", "human")
    ruby_ai = count("ruby", "ai")
    ruby_human = count("ruby", "human")

    rows = [
```

```
        {"metric": "total_repos_scanned", "value": total_repos_scanned},
        {"metric": "java_ai_repos", "value": java_ai},
        {"metric": "java_human_repos", "value": java_human},
        {"metric": "python_ai_repos", "value": python_ai},
        {"metric": "python_human_repos", "value": python_human},
        {
            "metric": "javascript_typescript_ai_repos",
            "value": js_ai,
        },
        {
            "metric": "javascript_typescript_human_repos",
            "value": js_human,
        },
        {"metric": "ruby_ai_repos", "value": ruby_ai},
        {"metric": "ruby_human_repos", "value": ruby_human},
    ]

    summary_path.parent.mkdir(parents=True, exist_ok=True)
    print(f"[summary] writing repo counts to {summary_path}")
    with open(summary_path, "w", encoding="utf-8", newline="") as f:
        writer = csv.DictWriter(f, fieldnames=["metric", "value"])
        writer.writeheader()
        writer.writerows(rows)

# ---------------------------------------------------------------------
# SARIF merge + CSV extraction (with CVSS-like scoring)
# ---------------------------------------------------------------------
```

```
def enrich_and_export(
    sarif_paths: List[Path],
    combined_sarif_path: Path,
    csv_path: Path,
    repo_name: str,
    origin: str,
    master_csv: Optional[Path],
) -> None:
    """
    Combine SARIF runs from multiple language analyses, compute a CVSS-like score
    for each result, inject it into result.properties["cvss_score"], and export
    a CSV summary.

    Per-repo CSV columns:
      repo, origin, language,
      security_severity, level, precision,
      security_vulnerability, cvss_score

    If there are no results at all, emit a single row with:
      security_vulnerability = "No security vulnerabilities for this repo"
      cvss_score = 0

    If master_csv is provided, append the same rows to that file as well,
    creating it and writing the header if it does not yet exist, and then
    update master-summary.csv with repo counts.
    """

    def load_runs(p: Path) -> List[Dict[str, Any]]:
```

```
    try:
        obj = load_json(p)
        runs = obj.get("runs") or []
        lang = p.stem.lower()  # 'java', 'python', 'javascript', 'ruby' from filename
        print(f"[sarif] {p.name}: {len(runs)} runs (language={lang})")

        # Annotate each run with its language so we can use it later
        for run in runs:
            props = run.get("properties") or {}
            props["_language"] = lang
            run["properties"] = props

        return runs
    except Exception as e:
        print(f"[warn] Could not read SARIF {p}: {e}", file=sys.stderr)
        return []

combined: Dict[str, Any] = {
    "version": "2.1.0",
    "$schema": "https://json.schemastore.org/sarif-2.1.0.json",
    "runs": [],
}
total_results_seen = 0
languages_seen = set()

# bring in all runs
for sp in sarif_paths:
    if sp and sp.exists():
```

```
        runs = load_runs(sp)
        combined["runs"].extend(runs)
    else:
        print(f"[warn] SARIF path missing or does not exist: {sp}", file=sys.stderr)

rows: List[Dict[str, Any]] = []

# walk each run
for run_idx, run_obj in enumerate(combined.get("runs", []), start=1):
    run_props = run_obj.get("properties") or {}
    run_lang = run_props.get("_language", "unknown")
    languages_seen.add(run_lang)

    tool = run_obj.get("tool") or {}
    driver = tool.get("driver") or {}
    extensions = tool.get("extensions") or []

    # Build a map of rule metadata
    rules_by_id: Dict[str, Dict[str, Any]] = {}

    def harvest_rules(drv: Dict[str, Any]) -> None:
        for rule in (drv.get("rules") or []):
            rid = rule.get("id") or ""
            rule_name = (
                (rule.get("shortDescription") or {}).get("text")
                or rule.get("name")
                or (rule.get("fullDescription") or {}).get("text")
                or ""
```

```
        )
        props = rule.get("properties") or {}
        tags = props.get("tags") or []
        cwe_tags = [t for t in tags if "cwe" in t.lower()]
        rules_by_id[rid] = {
            "rule": rule,
            "rule_name": rule_name,
            "cwe_tags": cwe_tags,
            "precision": props.get("precision"),
            "security_severity": props.get("security-severity"),
        }

harvest_rules(driver)
for ext in extensions:
    harvest_rules(ext)

results = run_obj.get("results") or []
print(f"[sarif] run {run_idx}: {len(results)} results (language={run_lang})")
total_results_seen += len(results)

for res in results:
    # rule id can appear as result.ruleId OR result.rule.id
    rule_id = res.get("ruleId")
    if not rule_id:
        rule_obj = res.get("rule") or {}
        rule_id = rule_obj.get("id") or ""

    level = (res.get("level") or "").lower()
```

```
meta = rules_by_id.get(rule_id, {})
rule = meta.get("rule") or {}

# properties
rule_props = rule.get("properties") or {}
result_props = res.get("properties") or {}

# security_severity from any of: rule metadata, meta, or result properties
secsev = (
    meta.get("security_severity")
    or rule_props.get("security-severity")
    or result_props.get("security-severity")
)

# precision from rule or result
precision = result_props.get("precision") or meta.get("precision")

# CWE tags, used to name the vulnerability
cwe_tags = meta.get("cwe_tags") or []
if cwe_tags:
    security_vulnerability = ";".join(cwe_tags)
else:
    # fall back to rule name/description
    security_vulnerability = (
        meta.get("rule_name")
        or rule.get("name")
        or (rule.get("fullDescription") or {}).get("text")
```

```
        or ""
    )

# normalize severity to string for CSV
if secsev is None:
    security_severity = ""
else:
    security_severity = str(secsev)

# ---- CVSS-like scoring ----
# Impact from security-severity if numeric; else map from level
if secsev is None:
    impact = map_level_to_impact(level)
else:
    try:
        impact = float(secsev)
    except Exception:
        impact = map_level_to_impact(level)

# Exploitability from precision, with slight boost if codeFlows exist
exploitability = map_precision_to_exploitability(precision)
if res.get("codeFlows"):
    exploitability = min(10.0, exploitability + 1.0)

cvss_score = compute_cvss_like(impact, exploitability)

# Inject CVSS-like score into SARIF result.properties
if not isinstance(result_props, dict):
```

```
        result_props = {}
      result_props["cvss_score"] = int(cvss_score)
      res["properties"] = result_props

      rows.append({
        "repo": repo_name,
        "origin": origin,
        "language": run_lang,
        "security_severity": security_severity,
        "level": level,
        "precision": precision or "",
        "security_vulnerability": security_vulnerability,
        "cvss_score": int(cvss_score),
      })

# If there were NO results at all, still emit a single row for analysis
if not rows:
  if languages_seen:
    lang_value = ",".join(sorted(languages_seen))
  else:
    lang_value = "unknown"

  print(f"[info] No security vulnerabilities for repo {repo_name}; emitting a single summary row.")
  rows.append({
    "repo": repo_name,
    "origin": origin,
    "language": lang_value,
    "security_severity": "",
```

```
        "level": "",
        "precision": "",
        "security_vulnerability": "No security vulnerabilities for this repo",
        "cvss_score": 0,
    })

# write combined SARIF (merged runs, with _language tags and cvss_score per result)
save_json(combined_sarif_path, combined)
print(f"[sarif] total results merged: {total_results_seen}")

# write per-repo CSV
fieldnames = [
    "repo", "origin", "language",
    "security_severity", "level", "precision",
    "security_vulnerability", "cvss_score",
]
print(f"[csv] writing {len(rows)} rows to {csv_path}")
with open(csv_path, "w", newline="", encoding="utf-8") as f:
    writer = csv.DictWriter(f, fieldnames=fieldnames)
    writer.writeheader()
    writer.writerows(rows)

# append to master CSV and update master summary
if master_csv is not None:
    master_csv_path = master_csv
    master_csv_path.parent.mkdir(parents=True, exist_ok=True)
    file_exists = master_csv_path.exists()
    print(f"[master-csv] appending {len(rows)} rows to {master_csv_path}")
```

```
    with open(master_csv_path, "a", newline="", encoding="utf-8") as f:
        writer = csv.DictWriter(f, fieldnames=fieldnames)
        if not file_exists:
            writer.writeheader()
        writer.writerows(rows)

    # Update master summary (repo counts)
    summary_path = master_csv_path.parent / "master-summary.csv"
    update_master_summary(master_csv_path, summary_path)

# ---------------------------------------------------------------------
# Main
# ---------------------------------------------------------------------

def main() -> None:
    ap = argparse.ArgumentParser(
        description="Universal CodeQL scanner (Java, Python, JavaScript, Ruby) with origin, language, and
CVSS-like scoring."
    )
    ap.add_argument("repo", help="Path to the repository root")
    ap.add_argument(
        "--codeql",
        default=os.environ.get("CODEQL", "codeql"),
        help="Path to CodeQL CLI (default: 'codeql' on PATH or $CODEQL)",
    )
    ap.add_argument(
        "--dbdir",
        default=".codeql-db",
```

```
    help="Directory to store CodeQL databases (will create <dbdir>/<lang>-db)",
)
ap.add_argument(
    "--outdir",
    default=".codeql-out",
    help="Directory for analysis outputs (SARIF + per-repo CSV)",
)
ap.add_argument(
    "--master-csv",
    default=None,
    help="Optional override path for master CSV; "
         "if not provided, uses C:\\codeql-study\\.codeql-out\\master-results.csv",
)
ap.add_argument(
    "--force-rescan",
    action="store_true",
    help="Scan the repo even if it is already present in the master CSV",
)
args = ap.parse_args()

source = Path(args.repo).resolve()
if not source.is_dir():
    print(f"Source path is not a directory: {source}", file=sys.stderr)
    sys.exit(1)

origin = get_repo_origin(source)
print(f"[info] Repo origin (from labels): {origin}")
```

```
dbdir = Path(args.dbdir).resolve()
outdir = Path(args.outdir).resolve()
dbdir.mkdir(parents=True, exist_ok=True)
outdir.mkdir(parents=True, exist_ok=True)

# Default master CSV: C:\codeql-study\.codeql-out\master-results.csv
if args.master_csv:
    master_csv_path = Path(args.master_csv).resolve()
else:
    master_csv_path = Path(r"C:\codeql-study\.codeql-out\master-results.csv").resolve()

# --- skip repo if already present in master-results.csv (unless forced) ---
if not args.force_rescan:
    if repo_already_in_master(source.name, master_csv_path):
        print(f"[info] Repo {source.name} already present in master CSV; skipping scan.")
        print(f"[info] (Use --force-rescan if you really want to rescan this repo.)")
        return

langs = detect_languages(source)
if not langs:
    print("No Java/Python/JavaScript/Ruby sources detected.", file=sys.stderr)
    # still emit a single CSV row for completeness
    combined_sarif = outdir / "codeql-results.sarif"
    csv_path = outdir / "codeql-results-summary.csv"
    enrich_and_export(
        [],
        combined_sarif,
        csv_path,
```

```
        repo_name=source.name,
        origin=origin,
        master_csv=master_csv_path,
    )
    sys.exit(0)

print(f"[info] Detected languages: {', '.join(langs)}")

sarifs: List[Path] = []
for lang in langs:
    db_path = dbdir / f"{lang}-db"

    if db_path.exists():
        print(f"[db] Reusing existing CodeQL database at {db_path}")
        # For Java, ensure the DB is finalized (ignore errors if it's already finalized)
        if lang == "java":
            run([args.codeql, "database", "finalize", str(db_path)], check=False)
    else:
        print(f"[db] Creating CodeQL database at {db_path}")
        create_cmd: List[str] = [
            args.codeql,
            "database", "create",
            str(db_path),
            "--language", lang,
            "--source-root", str(source),
        ]
        if lang == "java":
            create_cmd += java_build_command(source)
```

```
            # Remove empty args
            create_cmd = [c for c in create_cmd if c]
            run(create_cmd, check=True)

        # Analyze with the official security suite
        sarif_path = analyze_language(args.codeql, db_path, lang, outdir)
        sarifs.append(sarif_path)

    # Merge + export
    combined_sarif = outdir / "codeql-results.sarif"
    csv_path = outdir / "codeql-results-summary.csv"
    enrich_and_export(
        sarifs,
        combined_sarif,
        csv_path,
        repo_name=source.name,
        origin=origin,
        master_csv=master_csv_path,
    )
    print("\nDone.")
    print(f"- Combined SARIF: {combined_sarif}")
    print(f"- Per-repo CSV:   {csv_path}")
    print(f"- Master CSV:     {master_csv_path}")
    print(f"- Summary CSV:    {master_csv_path.parent / 'master-summary.csv'}")

if __name__ == "__main__":
    main()
```

Appendix C

Delimitations Table

Table 12

Refined Delimitations

Delimitation	Why This Matters	How this Relates to the Literature	How this Fits into the Study
Focusing on public GitHub Repositories	Public repositories allow open access, making the study reproducible. Since past research has focused on open-source AI-generated code, this keeps the study aligned with existing work.	Prior studies highlight security concerns in open-source AI-generated code, showing that public repositories are the most relevant focus.	This study aims to provide real-world insights into AI-generated code security risks. Public repositories offer the best way to assess this issue at scale.
Excluding private repositories and hybrid source code	Private repositories are not accessible for analysis, and hybrid code (where humans modify AI-generated code) introduces complexity. To keep the study focused, by only comparing AI-generated and human-generated code.	AI security research tends to compare AI-generated and human-written code separately rather than looking at hybrid modifications, which supports this study's approach.	By excluding private and hybrid code, the study ensures that AI-generated code is analyzed without human modifications confounding the results.
Using CodeQL for security analysis	CodeQL is widely used for security analysis and has been validated in prior research. Using a consistent tool ensures reliable results.	Studies have confirmed that CodeQL is effective for identifying vulnerabilities, making it a reasonable choice.	The goal is to identify security vulnerabilities, so using a proven security analysis tool directly supports the study's purpose.
Limited to specific programming languages	Different programming languages have different security vulnerabilities. By focusing on specific languages commonly used in AI-generated code, the study avoids inconsistencies.	Security research often acknowledges that programming language differences impact vulnerability risks. Controlling for language ensures meaningful comparisons.	By narrowing the scope to a few languages, the study remains focused and avoids unnecessary variability.
Focusing only on code security, not code quality	This study is strictly about security, not overall code quality. Security-focused research shows that vulnerabilities need to be assessed separately from maintainability or efficiency.	Existing studies separate security vulnerabilities from broader software quality assessments, reinforcing the need for a focused approach.	Since the research question is about security risks in AI-generated code, analyzing other aspects like efficiency or maintainability would go beyond the study's intended focus.

Note: This defines the scope of the study.

Appendix D

PowerShell Commands

```
# This will list all immediate subfolders inside C:\codeql-study\repos and save to repos.txt
(Get-ChildItem -Directory .\repos | Select-Object -ExpandProperty FullName) | Set-Content
.\repos.txt
# Peek at it:
Get-Content .\repos.txt
# Use this to run Codeql code (python script) for scanning repos:
Get-Content .\repos.txt |
  ForEach-Object {
    $r = $_.Trim()
    if (-not $r) { continue }

    Write-Host "==== Scanning $r ===="

    python .\codeql.py "$r" `
      --dbdir "$r\.codeql-db" `
      --outdir "$r\.codeql-out"

    Write-Host "==== Finished $r ===="
    Write-Host ""
  }
```

Appendix E

IRB Approval Letter

9388 Lightwave Ave.

San Diego, CA 92123

irb@nu.edu

Notice of Not Research/Not Human Subjects

November 10, 2025

To: Brenda Palmer

Project Title: A Causal-Comparative Study of Security Vulnerabilities in AI-Generated versus Human-Generated Source Code from GitHub

NU IRB Number: IRB-FY25-26-392

Determination: No IRB Oversight Required

No IRB Required - Research activities may begin as of November 10, 2025

Regulations:

The Department of Health and Human Services (DHHS) Code of Federal Regulations (45 CFR 46) defines research as, "a systematic investigation, including research development, testing and evaluation, designed to develop or contribute to generalizable knowledge.

*The Department of Health and Human Services (DHHS) Code of Federal Regulations (**45 CFR 46.102(f)**) defines a human subject as a living individual about whom an investigator (whether professional or student) conducting research obtains (1) Data through intervention or interaction with the individual, or (2) Identifiable private information.*

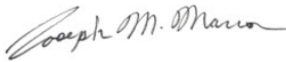
Dear Brenda Palmer:

In determining whether or not a project requires review by the IRB, our first step is to determine if the project involves research and human subjects, as the IRB only reviews applications which involve both.

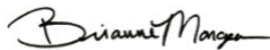
In applying the aforementioned regulations to this project, it has been determined that this project does not require IRB review. It does not appear that the proposed project involves human subjects research.

For questions related to this correspondence, please contact the IRB office ((858) 642-8384 or irb@nu.edu). If at any point you believe this research project involves human subjects, please contact the IRB office to request a review.

Sincerely,



Dr. Joseph Marron, IRB Chair



Dr. Brianne Mongeon, Director, HRPP & IRB

Jenessa Eberhardt, Associate Director, HRPP & IRB

Appendix F

Public GitHub Repository Licenses

The repositories that were cloned to the researcher's personal computer, later to be scanned by CodeQL for security vulnerabilities, either had no license or had a license. The three licenses used in this study were: GNU General Public License, MIT license and Do What The F*ck You Want To Public License - WTFPL; along with a Code of Conduct, which all was permissible for this particular study.

GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <https://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies

of this license document, but changing it is not allowed.

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to

share and change all versions of a program--to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents.

States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work

in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the

work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system

(if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10

makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice;

keep intact all notices stating that this License and any
non-permissive terms added in accord with section 7 apply to the code;
keep intact all notices of the absence of any warranty; and give all
recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey,
and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to
produce it from the Program, in the form of source code under the
terms of section 4, provided that you also meet all of these conditions:

a) The work must carry prominent notices stating that you modified
it, and giving a relevant date.

b) The work must carry prominent notices stating that it is
released under this License and any conditions added under section
7. This requirement modifies the requirement in section 4 to
"keep intact all notices".

c) You must license the entire work, as a whole, under this

License to anyone who comes into possession of a copy. This
License will therefore apply, along with any applicable section 7
additional terms, to the whole of the work, and all its parts,
regardless of how they are packaged. This License gives no
permission to license the work in any other way, but it does not
invalidate such permission if you have separately received it.

d) If the work has interactive user interfaces, each must display
Appropriate Legal Notices; however, if the Program has interactive
interfaces that do not display Appropriate Legal Notices, your
work need not make them do so.

A compilation of a covered work with other separate and independent
works, which are not by their nature extensions of the covered work,
and which are not combined with it such as to form a larger program,
in or on a volume of a storage or distribution medium, is called an
"aggregate" if the compilation and its resulting copyright are not
used to limit the access or legal rights of the compilation's users
beyond what the individual works permit. Inclusion of a covered work
in an aggregate does not cause this License to apply to the other
parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object

code is in no case prevented or interfered with solely because
modification has been made.

If you convey an object code work under this section in, or with, or
specifically for use in, a User Product, and the conveying occurs as
part of a transaction in which the right of possession and use of the
User Product is transferred to the recipient in perpetuity or for a
fixed term (regardless of how the transaction is characterized), the
Corresponding Source conveyed under this section must be accompanied
by the Installation Information. But this requirement does not apply
if neither you nor any third party retains the ability to install
modified object code on the User Product (for example, the work has
been installed in ROM).

The requirement to provide Installation Information does not include a
requirement to continue to provide support service, warranty, or updates
for a work that has been modified or installed by the recipient, or for
the User Product in which it has been modified or installed. Access to a
network may be denied when the modification itself materially and
adversely affects the operation of the network or violates the rules and
protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided,

in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of

it) with contractual assumptions of liability to the recipient, for
any liability that these contractual assumptions directly impose on
those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the

licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within

the scope of its coverage, prohibits the exercise of, or is
conditioned on the non-exercise of one or more of the rights that are
specifically granted under this License. You may not convey a covered
work if you are a party to an arrangement with a third party that is
in the business of distributing software, under which you make payment
to the third party based on the extent of your activity of conveying
the work, and under which the third party grants, to any of the
parties who would receive the covered work from you, a discriminatory
patent license (a) in connection with copies of the covered work
conveyed by you (or copies made from those copies), or (b) primarily
for and in connection with specific products or compilations that
contain the covered work, unless you entered into that arrangement,
or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting
any implied license or other defenses to infringement that may
otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License. If you cannot convey a

covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to

address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT
HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY
OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM
IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF
ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS
THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY

GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE
USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF
DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD
PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS),
EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF
SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided
above cannot be given local legal effect according to their terms,
reviewing courts shall apply local law that most closely approximates
an absolute waiver of all civil liability in connection with the
Program, unless a warranty or assumption of liability accompanies a
copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.>
Copyright (C) <year> <name of author>

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program. If not, see <https://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short
notice like this when it starts in an interactive mode:

<program> Copyright (C) <year> <name of author>
This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate
parts of the General Public License. Of course, your program's commands
might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school,
if any, to sign a "copyright disclaimer" for the program, if necessary.
For more information on this, and how to apply and follow the GNU GPL, see
<https://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program
into proprietary programs. If your program is a subroutine library, you
may consider it more useful to permit linking proprietary applications with
the library. If this is what you want to do, use the GNU Lesser General
Public License instead of this License. But first, please read
<https://www.gnu.org/licenses/why-not-lgpl.html>.

MIT License

Copyright (c) 2023 THU-KEG

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR

IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF

MERCHANTABILITY,

FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT

SHALL THE

AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR

OTHER

LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,

ARISING FROM,

OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER

DEALINGS IN THE

SOFTWARE.

DO WHAT THE F*CK[2] YOU WANT TO PUBLIC LICENSE

Version <number>, <month year>

Copyright (C) <year> <software developer>

Everyone is permitted to copy and distribute verbatim or modified

copies of this license document, and changing it is allowed as long

as the name is changed.

DO WHAT THE F*CK[3] YOU WANT TO PUBLIC LICENSE

[2] Adding asterisk for a profanity word used in an actual GitHub License
[3] Adding asterisk for a profanity word used in an actual GitHub License

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. You just DO WHAT THE F*CK[4] YOU WANT TO.

As contributors and maintainers of this project, and in the interest of fostering an open and welcoming community, we pledge to respect all people who contribute through reporting issues, posting feature requests, updating documentation, submitting pull requests or patches, and other activities.

We are committed to making participation in this project a harassment-free experience for everyone, regardless of level of experience, gender, gender identity and expression, sexual orientation, disability, personal appearance, body size, race, ethnicity, age, religion, or nationality.

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery
- Personal attacks
- Trolling or insulting/derogatory comments
- Public or private harassment
- Publishing other's private information, such as physical or electronic addresses, without explicit permission
- Other unethical or unprofessional conduct

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of

[4] Adding asterisk for a profanity word used in an actual GitHub License

Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

By adopting this Code of Conduct, project maintainers commit themselves to fairly and consistently applying these principles to every aspect of managing this project. Project maintainers who do not follow or enforce the Code of Conduct may be permanently removed from the project team.

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community.

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting a project maintainer <contact email>. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. Maintainers are obligated to maintain confidentiality with regard to the reporter of an incident.

This Code of Conduct is adapted from the Contributor Covenant, version 1.3.0, available at http://contributor-covenant.org/version/1/3/0/