

CodeWars: Using LLMs for Vulnerability Analysis in Cybersecurity Education

Arunima Chaudhary
School of Computer Science
Cardiff University
Cardiff, United Kingdom
0009-0008-5265-8679

Gualtiero Colombo
School of Computer Science
Cardiff University
Cardiff, United Kingdom
0000-0002-7348-1939

Amir Javed
School of Computer Science
Cardiff University
Cardiff, United Kingdom
0000-0001-9761-0945

Junaid Haseeb
School of Comp. & Maths. Sciences
University of Waikato
Hamilton, New Zealand
0000-0003-0847-5353

Vimal Kumar
School of Comp. & Maths. Sciences
University of Waikato
Hamilton, New Zealand
0000-0002-4955-3058

Richard Larsen
School of Comp. & Maths. Sciences
University of Waikato
Hamilton, New Zealand
0009-0004-8776-7753

Abstract—Large Language Models (LLMs) are increasingly explored as tools for software development and could further constitute a supplementary source for the development of varied examples intended for pedagogical use. While they can improve productivity, their ability to produce code that is both secure and compliant with Secure Software Development (SSD) practices remains uncertain, raising concerns about their role in cybersecurity education. If LLMs are to be integrated effectively, students must be trained to critically evaluate generated code for correctness and vulnerabilities, raising an important question: How can LLM-generated code be effectively and securely incorporated into Cybersecurity education for teaching vulnerability analysis? This paper introduces *CodeWars*, a novel teaching methodology that combines LLM-generated and human-written code to examine how students engage with vulnerability detection tasks. *CodeWars* was implemented as a pilot study with a total of 32 students at Cardiff University and the University of Waikato, where students analyzed flawed, secure, and mixed-origin code samples. By comparing student approaches, analysis, and perceptions, the study provides insights into how vulnerabilities are detected, how code origins are distinguished, and how SSD practices are applied. Our analysis of student feedback and interviews indicates that *Codewars* produced structured and accessible code, simplifying vulnerability identification and offering educators the means to efficiently develop varied SSD teaching applications. These findings illuminate both the advantages and constraints of employing LLMs in secure coding and position this study as a foundational step toward the responsible adoption of AI in Cybersecurity Education.

Keywords—Cybersecurity Education, Vulnerability analysis, Secure Software Development, Large Language Models, Cybersecurity Pedagogy, GenAI

I. INTRODUCTION

Software security has become a critical concern as modern systems grow increasingly complex and interconnected. Despite advances in programming languages, frameworks, and development tools, vulnerabilities continue to emerge at every stage of the software lifecycle [1]. Preparing the next generation of developers to write secure code is therefore essential [2]. Secure Software Development (SSD) is no longer a specialised skill but a fundamental requirement, and its teaching is central to equipping students with the ability to recognize, prevent, and mitigate security flaws in real-world applications [3] [4]. Generative Artificial Intelligence (GenAI), particularly Large Language Models (LLMs), has emerged as a transformative tool in Software Engineering, capable of rapidly generating syntactically correct and functionally plausible code. This offers clear benefits, from reducing development time to creating new opportunities for learning. Yet, despite their strengths in producing functional implementations, the security of LLM-generated code remains uncertain [5]. This uncertainty underscores the need to evaluate LLMs not only as programming assistants but also as pedagogical tools.

In parallel, Cybersecurity education faces the challenge of keeping pace with rapidly evolving threats. Secure coding education is essential for students to acquire the skills necessary to adapt once they graduate. Educators increasingly incorporate security into their teaching, however, effective application requires attention to both teaching and learning perspectives. Understanding students' security practices is a useful first step in creating a baseline for teaching methods and content [6]. Establishing where students initially struggle allows educators to design interventions that target gaps in secure coding knowledge.

To keep up with the changing threat landscape and simulate cyber threats using LLMs, we introduce *CodeWars*, an instructional methodology that integrates LLM-generated and human-written code into secure coding education. To validate the methodology, a pilot of two experimental studies were conducted where students analysed flawed and secure code snippets by applying SSD practices to detect vulnerabilities. The studies, at Cardiff University and the University of Waikato, employed a dual-path design: one pathway offered guided support to introduce vulnerabilities and build foundational skills, while the other challenged students with more unguided analysis tasks. This progression reflects effective teaching practice, moving from structured learning to independent application.

To the best of our knowledge, this is the first study that combines LLM code generation with an educational design that evaluates both technical outcomes and student learning processes. Rather than focusing solely on whether LLMs can produce secure code, this paper investigates how students interact with such code, how they differentiate between human and LLM-generated artifacts, and how these interactions can be harnessed to improve Cybersecurity education. In doing so, *CodeWars* contributes to the development of an innovative pedagogical method for teaching SSD and provides a foundation for integrating AI tools into future Cybersecurity curricula. The contributions of this study are:

- Investigation of the integration of LLMs into SSD education by developing case studies that evaluate adherence to SSD practices against the OWASP Top 10 (2021) guidelines.
- Examination of how students identify vulnerabilities in LLM- and human-generated code.
- Evaluation of students' ability to distinguish between LLM-generated and human-written code, and to analyse the factors influencing their authorship judgments.
- Exploration of student perceptions, preferences, and ease of understanding different code origins.
- Evaluation of the pedagogical effectiveness of LLMs for secure coding instruction.

II. EXISTING WORKS

This section reviews the existing methodologies, educational methods, and research in vulnerability detection and Cybersecurity pedagogy.

LLMs have attracted increasing attention as tools for software development, offering the ability to generate functional code rapidly and at scale. They have been shown to support tasks such as boilerplate generation, Application Programming Interface (API) usage, and debugging [7], [8]. Beyond productivity gains, LLMs are also being explored in educational contexts as a supplementary source of diverse coding examples that can enrich programming pedagogy [9]. Their capacity to provide varied outputs positions them as potentially valuable resources for teaching SSD.

Despite these advantages, considerable concerns persist regarding the security of LLM-generated code. Research indicates that while such outputs are often syntactically correct and appear functional, they frequently omit established SSD practices and introduce exploitable vulnerabilities [10], [11]. For instance, Tihanyi *et al.* [11] demonstrated that a large proportion of AI-generated C programmes contained vulnerabilities despite compiling successfully. Taylor *et al.* [12] and Chi *et al.* [13] argue that Cybersecurity graduates are likely to encounter security vulnerabilities early in their careers, making SSD instruction essential. Such findings highlight the need for critical scrutiny when incorporating LLM outputs into teaching.

One of the major gaps in Cybersecurity education is the absence of practical, hands-on learning opportunities. Current research highlights the need for innovative pedagogy in security education and points toward Problem-Based Learning (PBL) as a promising approach [14]. PBL immerses students in authentic, problem-driven tasks and has been shown to foster both technical competence and critical thinking in Cybersecurity education. [15] advocates for integrating Artificial Intelligence (AI) and Machine Learning (ML) into Cybersecurity education to make it more aligned with current industry trends. In this context, tools such as vulnerability scanners and interactive platforms can play a pivotal role by allowing students to analyze real code, identify flaws, and apply mitigation strategies in a guided environment.

Such platforms address the call for stronger technical foundations in education, as identified by [16]. Furthermore, they offer an avenue to merge theoretical learning with practical application, a synergy deemed essential by [17] for preparing students for workforce demands.

Overall, the literature establishes the importance of early and effective Cybersecurity education, the need for clearer curriculum content, and the potential of intelligent tools to enhance learning outcomes. Integrating code vulnerability scanning platforms into student training offers a promising approach to fill these gaps. While prior work has examined the quality and security of LLM outputs, there remains limited research on how such code can be pedagogically integrated into cybersecurity practice. This gap motivates the present study, which introduces *CodeWars*, a structured teaching methodology combining LLM-generated and human-written code to investigate how students approach vulnerability detection tasks.

III. PROJECT DESIGN

In contrast to conventional pedagogical approaches—such as flipped classrooms complemented by laboratory-based exercises in cybersecurity and software engineering, or project-based learning—the *CodeWars* methodology was conceived as a classroom-based educational activity to investigate the role of LLMs in cybersecurity education. Its design employed a structured three-phase approach as shown in Figure 1.

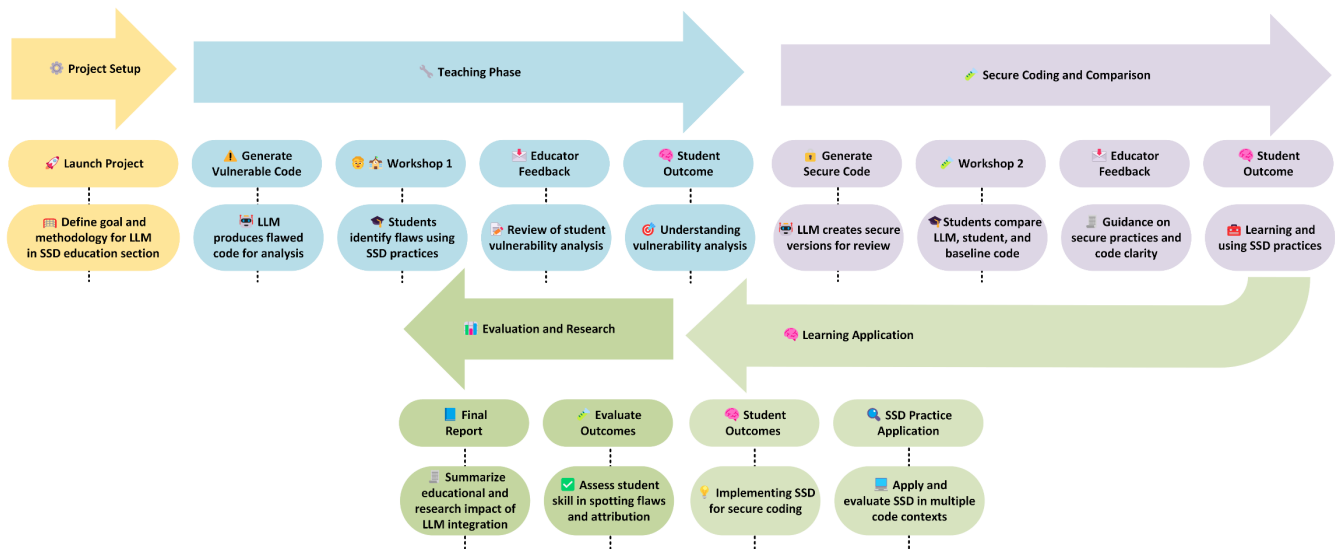


Fig. 1. Project flowchart with different phases.

Phase 1 – Development of Case Studies: In Phase 1, Research Assistants (RAs) at Cardiff University and the University of Waikato independently developed case studies to support the study objectives. At Cardiff University, LLMs (ChatGPT [18], DeepSeek [19], and Gemini [20]) were provided with the same set of requirements derived from the coursework brief that guided the production of student-generated code, ensuring comparability across code origins. At Cardiff, three case studies were created in an unguided format:

- *Case Study 1* – Student-written code from a real client project.
- *Case Study 2* – LLM-generated code, developed with the same requirements as *Case Study 1*.
- *Case Study 3* – Expert-written secure code serving as a control.

At Waikato, five case studies were created in a guided format:

- *CS1, CS2* – AI-generated code (Gemini).
- *CS3, CS4* – Human-created code.
- *CS5* – Human-created, poorly formatted code.

Both institutions used case studies based on LLM-generated and human-written code, with Cardiff including an additional control case to increase the level of difficulty and encourage deeper analysis. This was to examine not only the ability of students to detect vulnerabilities, but also to align with the broader objective of the study: to explore students' ability to distinguish between LLM-generated and human-written code, and to analyse the factors influencing their authorship judgments, and apply SSD practices across diverse coding styles.

A. Code Generation Process for LLM-Generated Case Studies

This brief subsection provides more details about the code generation process using LLMs undertaken in Cardiff University. The primary task was to implement a functional and secure web login page. To ensure consistency and completeness, the prompting process was structured into a series of subtasks, creating a coherent chain-of-thought and guiding the models toward producing a deployable application.

The sequence of prompts included:

- Application Setup.
- Configuring the application using Gradle.
- Implementing a responsive login page with attention to data privacy and security.

The generated outputs were first assessed for functional completeness, deployment feasibility, and correct application layer structure. DeepSeek produced the most logically complete implementation, including proper Data Transfer Object (DTO) handling, integration between application layers, a Bootstrap-based responsive design, and a CSS stylesheet—features absent in the outputs from ChatGPT and Gemini. As a result, DeepSeek's implementation was selected for further analysis. The selected code was then evaluated against the OWASP Top 10 (2021) guidelines [21] to determine its adherence to widely recognised secure coding practices. Positive security implementations included: Password storage, role-based access control, secure session management, and Spring Security-based authentication. However, several gaps were identified, like, lack of password policy enforcement, account lockout mechanisms, absence of password length and complexity validation and no handling of failed authentication attempts.

This process resulted in the reflection of a realistic blend of functional completeness and security weaknesses, making it suitable for subsequent student analysis in the studies.

Phase 2 – Code Analysis: Participants were presented with anonymized code samples and instructed to identify vulnerabilities using SSD practices. Their responses were recorded through structured questionnaires, which also captured authorship attribution, error ratings, and perceived difficulty.

At the University of Waikato, the study followed a *guided model*. Participants were first introduced to core SSD principles and provided with a predefined list of vulnerabilities expected to be present in the code. Their task was to analyze the snippets and map each to vulnerabilities from the list. This approach aligned with *scaffolding theory* [22], where students begin with direct guidance and targeted practice before progressing to independent application.

At Cardiff University, an *unguided model* was adopted. Participants received no vulnerability list and were instead required to analyze anonymized code samples independently, mirroring real-world software security analysis where vulnerabilities must be identified without predefined hints.

Phase 3 – Comparative Analysis: Collected responses were analysed for:

- Accuracy in identifying vulnerabilities.
- Types and number of vulnerabilities reported.
- Application and effectiveness of SSD practices.
- Accuracy of code authorship attribution (LLM vs. human).
- Perceived difficulty and clarity of different code samples.

IV. RESULTS

A. Code Authorship Identification

Authorship attribution was assessed by asking participants to identify whether each code sample originated from an LLM, a student, or a 'secure' expert-written control snippet. The anonymization of samples ensured that decisions were based solely on code quality, style, and perceived implementation patterns.

Cardiff study: Out of 18 total participants:

- *Case study 1 (Student-written)* Correctly identified as human by 6, misclassified as LLM by 6.
- *Case study 2 (LLM-generated)* Misclassified as human by 12; only 4 correctly identified it as LLM.
- *Case study 3 (Secure baseline)* Misclassified as LLM by 10; correctly identified as human by 9.¹

Students' independent justifications revealed a strong reliance on surface-level features. Errors and inconsistencies were most cited as indicators of human authorship (16 mentions), with comments describing the code as "pragmatic but inconsistent" and "a lot messier." Conversely, tidy formatting, polished structure, and generic variable names were commonly linked to LLM authorship, often leading to misclassification of expert-written code, see Case study 3.

As illustrated in (see Figure 2) the most common error was misidentifying the LLM-generated code in Case study 2 as human-written because of some apparent untidiness in the structure and the perceived presence of syntax errors. This also suggests that LLM outputs have already reached a level of perceived human-like style and readability for some participants.

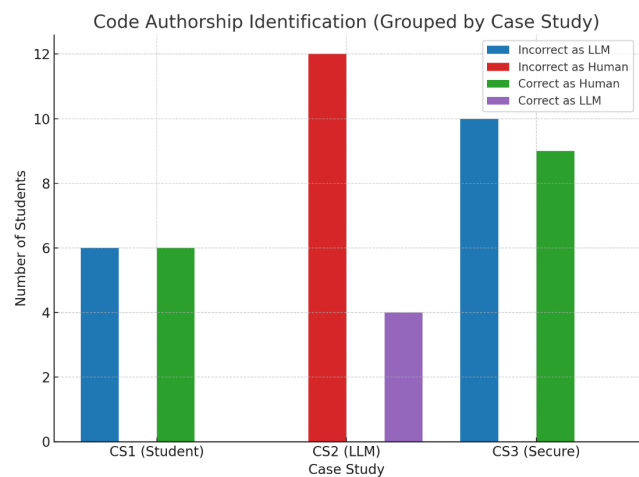


Fig. 2. Cardiff Study: Correct and incorrect identifications of code authorship for the three case studies.

Waikato study: Total 14 participants

- CS1 & CS2: AI-generated. (Gemini)
- CS3, CS4 & CS5: Human-written.
- CS3 (*Human-written*): Most frequently misclassified as LLM-generated.
- CS5 (*Poorly formatted human code*): Most confidently identified as human-written.

As illustrated in (see Figure 3) a similar misclassification pattern emerged in the Waikato study. CS1 and CS2 have been produced by asking LLMs to generate 'code specifically embedding vulnerabilities' and were then more frequently misattributed to humans other than correctly labelled as AI generated. The snippet most often believed to be LLM-generated was CS3 (in reality human-written) with more than half of participants misjudging it. By contrast, CS5 (a poorly

1. One student had classified Case Study 3 to be generated by both human and LLM.

formatted human-written snippet) was most confidently identified as human-authored. Participants were also asked which features influenced their judgments. Code comments stood out – over 80% selected them as the strongest LLM indicator, while simplicity of code was rated least useful (10%). Other markers like code alignment, naming, and flow fell in between, reinforcing that surface-level style outweighed deeper analysis.

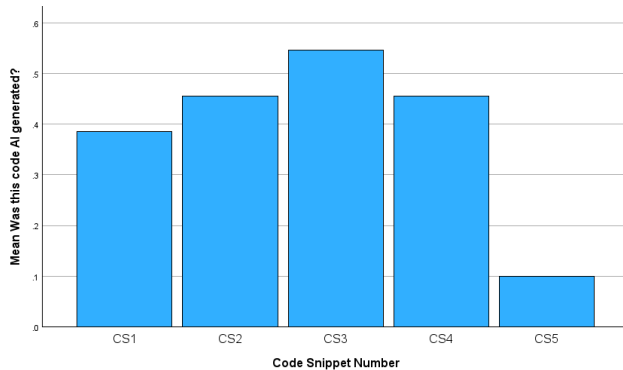


Fig. 3. Waikato Study: Correct and incorrect identifications of code authorship for the three case studies.

Both studies suggest a persistent cognitive bias: neatness and consistency were linked with LLM authorship, while imperfections suggested human origin. However, there were examples of misclassification in Cardiff where in CS2 (LLM) the perceived presence of some syntax led to mislabelling as human. On the contrary CS3, the expert generated control case, was perceived as the neatest on surface and frequently misclassified as LLM-written. These findings underscore a critical pedagogical challenge: students often rely on superficial stylistic features—such as naming conventions, formatting, and comment style, when distinguishing between human and AI-written code, rather than engaging in systematic vulnerability analysis. The paradox observed, where LLM-generated code was mistaken for human work and expert-written code was misattributed to AI, highlights the risk of overemphasising surface-level heuristics in secure coding education. This suggests that without explicit guidance, learners may conflate aesthetic qualities with trustworthiness, potentially overlooking deeper security flaws. Consequently, cybersecurity instruction must not only develop technical skills in vulnerability detection but also raise awareness of cognitive biases that shape perceptions of code authorship in an era of LLM-assisted development.

B. Vulnerability Identification in Code

Participants in both studies were asked to identify vulnerabilities present in the code snippets.

In the Cardiff Study, students identified the maximum number of vulnerabilities in the human-written code (23 mentions in total). These are clustered around *weak/no hashing* (4), *plaintext password storage* (3), and *SQL injection* (3), reflecting typical novice-level security oversights.

We have also computed the perceived number of security errors identified by participants for each of the code snippets, which are in turn proportionally linked with the number of specific vulnerabilities detected. Namely, we asked participants to give an error rating to each code snippet on a scale 1 to 5 (1 being error-free and 5 being a lot of errors). The average error rating score for the first snippet CS1, human generated code, was 3.33 calculated on average over all participants, the highest overall. By contrast, the LLM-generated snippet CS2 accounted for a lower number of vulnerability mentions, with weaknesses more often tied to *boundary* and *configuration errors*, and a lower error score of 2.43. The human-written control case CS3 instead attracted a slightly higher number of mentions, despite being in reality the control errorless test case, dominated by SQL injection, hardcoded secrets, and information disclosure, and received an error score of 2.66 on average.

For the Waikato study, five code snippets were constructed to include the following vulnerabilities:

- CS1: All three vulnerabilities (buffer overflow, format string, command injection).
- CS2: Format string and command injection.
- CS3: Buffer overflow and command injection.
- CS4: Buffer overflow and format string.
- CS5: Two buffer overflows

Figure 4 displays a graph that breaks down the percentage of participants who identified each different vulnerability across all the code snippets. The vulnerability that participants had the most confidence in was the format string vulnerability present in code snippet 1. CS5 had the lowest rate of incorrect identifications with only around 10 percent of participants selecting the two vulnerability types that weren't present.

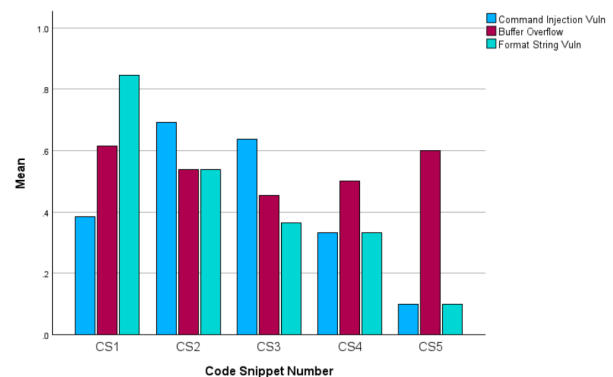


Fig. 4. Waikato Study: Vulnerability identification by students.

Table I, shows the percentages of participants that selected each vulnerability as being present in each code snippet. It also shows the accuracy as the average number of selections that correctly identified the vulnerabilities present.

Interestingly, the two AI generated code snippets had the highest rate of correct selections both being higher than the next highest human created code snippet. It is worth noting in this sense that CS5 contained two separate instances of a buffer overflow. In some cases it was unclear whether participants identified one or both occurrences, which may explain why CS5 achieved a higher rate compared to other human-created snippets.

TABLE I. Waikato Study: Percentage of vulnerabilities identified and accuracy across code snippets

Code Snippet	CI (%)	BO (%)	FS (%)	Accuracy (%)
1	38	61	85	61.33
2	70	53	53	61.50
3	64	45	37	54.50
4	35	50	35	42.50
5	10	60	10	60.00

C. Easiest and Hardest Vulnerabilities to Identify

The data collected in the Waikato study shows that participants identified buffer overflows as both the easiest and hardest vulnerability to detect, with nearly all selecting it for one of the two options. Half perceived them as easy to find, while nearly half found them most difficult, revealing varied individual confidence. On the contrary other types of vulnerabilities showed more consistent trends, such as command injection being consistently the most difficult to find.

In the Cardiff study, authentication-related flaws were most frequently detected (32 mentions), particularly weak or missing passwords and login issues, followed by unhandled errors and unsafe error messages. These were straightforward to recognise due to their visibility. In contrast, more complex issues like SQL injection (6 mentions) and database security flaws (7 mentions) were rarely identified, as they required students to connect implementation details with broader security concepts – a challenge in unguided settings.

Across both studies, participants could detect obvious flaws in both human-written and LLM-generated code, though accuracy varied. LLM outputs, while often appearing “cleaner,” sometimes concealed configuration risks, whereas human-written code included beginner-level errors that were more easily flagged, yet, subtle deployment and confidentiality issues were often missed. These findings underscore the need for structured vulnerability training, as surface-level analysis alone can lead to misjudgements.

V. DISCUSSION

This section contains the potential inferences that can be made from the collected data.

A. Correlation between code complexity and perceived LLM generation

In the Waikato study there appears to be a correlation between the perceived difficulty of the code to understand and whether or not the participant believed it to be LLM generated code. It appears that participants were more likely to believe a snippet of code was LLM-generated if it was more difficult to understand. The results were nearly mirrored between the two with the exception of CS5. This is an outlier as it was intentionally poorly-formatted and the participants are likely wise to the fact that LLM-generated code is typically indented correctly. This is further backed up by the fact that ‘simplicity’ was the least favoured category when it came to factors which identified code as being generated by an LLM. For the Cardiff study instead, across all cases the relationship between high error rating and code authorship seems weak. CS1 (student generated) was perceived with the highest error ratings but this did not make participants more likely to label the code as human than LLM generated. The effects of incorrect authorship attribution appears even worse for CS2 which, despite being assessed with the lowest error rating scores, was for the vast majority of participants incorrectly attributed to humans. Finally, CS3 (the expert generated errorless control case) also suffered from mislabelling as its perceived tidiness in the structure often led to attributions to an LLM. In conclusion, the students from Cardiff did not appear to systematically and coherently equate more complexity and a lower number of security errors with “LLM-generated.” or vice-versa.

B. Significance of comments and code structure when detecting if the code was LLM generated

For both the studies, the participants found comments as being the most significant identifying factor in whether or not a piece of code was LLM generated. It is unclear what exactly about the comments gave away the fact that code was LLM generated, be it the frequency or language used. Kerschbaumer *et al.* [23] notes that comments in source code are widely recognized by both professional programmers and educators as enhancing understandability, readability, and knowledge retention. However, this study shows that students may also over-rely on such features when judging code origin.

C. Correlation between vulnerabilities detected and code authorship

In the Waikato study, participants were more likely to identify vulnerabilities in LLM-generated code, with a 10–15% higher detection rate compared to human-written snippets (see Table 1). An exception was CS5, a human-written example containing two instances of the same vulnerability, which may have inflated detection due to repetition. In contrast, the Cardiff study found no consistent link between the number of vulnerabilities students detected and whether

they attributed the code to an LLM. These findings suggest that, especially in unguided settings, participants relied heavily on code style, structure, and perceived logic, rather than error volume. Post-questionnaire feedback reflected expectations that LLM code would appear “too perfect” or “generic.”

VI. CONCLUSIONS

The *CodeWars* teaching methodology demonstrates the potential for structured integration of LLM-generated code into Cybersecurity education. It produced several key findings. First, students frequently struggled to distinguish LLM-generated code from human-written code, demonstrating that modern LLMs can convincingly replicate human programming styles. At the same time, comments, formatting, and surface-level style often outweighed deeper vulnerability analysis in authorship attribution. This underscores the need to train students to evaluate code by incorporating authorship attribution exercises alongside SSD vulnerability analysis to existing teaching methods which can deepen students' critical engagement with code style, maintainability, and potential AI-generated patterns.

The vulnerability detection process in *CodeWars* was structured to align with the natural progression of skill development. In guided activities, as implemented at the University of Waikato, students were provided with a list of security vulnerabilities expected to be present in the code. This scaffolded approach built confidence and familiarity with SSD concepts before progressing to more unguided analysis tasks. In contrast, the Cardiff University study required students to apply their knowledge independently, using anonymized code samples without prior disclosure of the vulnerabilities. By combining guided and unguided teaching methods, *CodeWars* enabled engagement with diverse, authentic, project-based code snippets reflecting both realistic vulnerabilities and secure practices.

The results from both studies not only provided insight into how students detect vulnerabilities but also served as a teaching tool in post-analysis discussions. Reviewing detection patterns, missed vulnerabilities, and misconceptions lead to addressing gaps in understanding. The advantage of *CodeWars* lies in its dual focus: it teaches vulnerability detection while also encouraging reflection on code authorship, style, and the growing influence of AI in software development. Students benefited not only by improving their ability to identify vulnerabilities but also by gaining experience in comparing coding styles, reasoning about maintainability, and questioning the reliability of AI-generated code.

VII. FUTURE WORK

CodeWars appears to be a significant step in the right direction as it equips students with practical SSD skills, fosters critical engagement, and proposes a foundational step to responsible integration of LLMs in Cybersecurity Education. However, to comprehensively validate the research, future research could involve a larger and more diverse cohort of

participants. Based on observations during the studies and participant feedback, several areas have been identified for further development and refinement of the *CodeWars* structure. Future work will expand vulnerability coverage to include a wider range of SSD errors mapped to standards such as the OWASP Top 10 and CWE/SANS Top 25, giving students richer exposure to real-world security errors. Gamification will also be introduced to increase engagement, addressing observations that some participants viewed the activity as a formal assessment potentially limiting their engagement and willingness to experiment. Features such as scoring, timed challenges, and team competitions are expected to promote deeper understanding and long-term retention. Finally, a key direction is to explore whether LLMs can serve not only as generators but also as evaluators, providing automated feedback on vulnerability detection to enable scalable, adaptive learning while supporting educators in guiding higher-level discussions.

DECLARATION OF GENERATIVE AI AND AI-ASSISTED TECHNOLOGIES IN THE WRITING PROCESS

During the preparation of this work, the authors used ChatGPT-5 in order to improve readability and language. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

ACKNOWLEDGEMENTS

This research has been funded by Cardiff University and the University of Waikato under the project code AH28101518 and 11145, respectively. We would like to thank Mr. Haitao Wang (University of Waikato); Mr. Eeshan Waqar; Mr. Yasar Majib (Cardiff University) for their contribution to the generation of code snippets used in these studies. We would also like to thank Dr Fernando Alva Manchego (Cardiff University) for his support in organising the workshop.

REFERENCES

- [1] A. Nguyen, H. N. Ngo, Y. Hong, B. Dang, and B.-P. T. Nguyen, “Ethical principles for artificial intelligence in education,” *Education and Information Technologies*, vol. 28, pp. 4221–4241, Apr. 2023. doi: 10.1007/s10639-022-11316-w.
- [2] A. Chaudhary, A. Javed, W. Colombo, and F. A. Manchego, “Exploring the Safe Integration of Generative AI in Cybersecurity Education Addressing Challenges in Transparency, Accuracy, and Security,” in *Advances in Teaching and Learning for Cyber Security Education* (P. Legg, N. Coull, and C. Clarke, eds.), (Cham), pp. 1–21, Springer Nature Switzerland, 2024. doi: 10.1007/978-3-031-77524-6_1.
- [3] K. Qian, R. Parizi, E. Agu, F. Wu, and B.-T. Chu, *Authentic Learning Secure Software Development (SSD) in Computing Education*. Oct. 2018. doi: 10.1109/FIE.2018.8659217.
- [4] M. Ismail, N. T. Madathil, M. Alalawi, S. Alrabae, M. Al Bataineh, S. Melhem, and D. Mouheb, “Cybersecurity activities for education and curriculum design: A survey,” *Computers in Human Behavior Reports*, vol. 16, p. 100501, Dec. 2024. doi: 10.1016/j.chbr.2024.100501.
- [5] N. Tihanyi, T. Bisztray, M. A. Ferrag, R. Jain, and L. C. Cordeiro, “How secure is ai-generated code: A large-scale comparison of large language models,” *Empirical Software Engineering*, vol. 30, no. 2, p. 47, 2025. doi: 10.1007/s10664-024-10590-1.

- [6] T. Yilmaz and Ulusoy, "Understanding security vulnerabilities in student code: A case study in a non-security course," *Journal of Systems and Software*, vol. 185, p. 111150, Mar. 2022. doi: 10.1016/j.jss.2021.111150.
- [7] P. Vaithilingam and E. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, pp. 1–17, 2022. doi: 10.1145/3491101.3519665.
- [8] M. Chen, J. Tworek, H. Jun, et al., "Evaluating large language models trained on code," in *NeurIPS 2021 Datasets and Benchmarks Track*, 2021. doi: 10.48550/arXiv.2107.03374.
- [9] M. Kazemitabaar et al., "Studying the effect of ai code generators on novice programmers," in *Proceedings of the ACM Conference on International Computing Education Research*, pp. 47–63, 2023. doi: 10.48550/arXiv.2302.07427.
- [10] H. Pearce et al., "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*, pp. 754–768, 2022. doi: 10.48550/arXiv.2108.09293.
- [11] Z. Tihanyi, T. Bisztray, M. A. Ferrag, A. Jain, and L. Cordeiro, "How secure is ai-generated code: A large-scale comparison of large language models," *Empirical Software Engineering*, 2024. doi: 10.1007/s10664-024-10590-1.
- [12] B. Taylor, M. Bishop, E. Hawthorne, and K. Nance, "Teaching secure coding: The myths and the realities," in *Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*, pp. 281–282, 2013. doi: 10.1145/2445196.2445280.
- [13] H.-H. Chi, E. L. Jones, and J. Brown, "Teaching secure coding practices to stem students," in *Information Security Curriculum Development Conference (InfoSecCD '13)*, pp. 42–48, 2013. doi: 10.1145/2528908.2528911.
- [14] M. Shivapurkar, S. Bhatia, and I. Ahmed, "Problem-based Learning for Cybersecurity Education," *Journal of The Colloquium for Information Systems Security Education*, vol. 7, pp. 6–6, July 2020.
- [15] S. Grover, B. Broll, and D. Babb, "Cybersecurity Education in the Age of AI: Integrating AI Learning into Cybersecurity High School Curricula," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, (Toronto ON Canada), pp. 980–986, ACM, Mar. 2023. doi: 10.1145/3545945.3569750.
- [16] I. Kumar, "Emerging threats in cybersecurity: A review article," *International Journal of Applied and Natural Sciences*, vol. 1, no. 1, pp. 01–08, 2023. doi: 10.61424/ijans. Available: <https://bluemarkpublishers.com/index.php/IJANS/article/view/2>.
- [17] R. Nowrozy and D. Jam, "Embracing the Generative AI Revolution: Advancing Tertiary Education in Cybersecurity with GPT," Mar. 2024. doi: 10.48550/arXiv.2403.11402.
- [18] OpenAI, "Chatgpt: Optimizing language models for dialogue." <https://openai.com/chatgpt>, 2023. Accessed: 2025-09-18.
- [19] D. AI, "Deepseek: Open large language model." <https://www.deepseek.com/>, 2024. Accessed: 2025-09-18.
- [20] G. DeepMind, "Gemini: A family of multimodal models." <https://deepmind.google/technologies/gemini/>, 2023. Accessed: 2025-09-18.
- [21] OWASP Foundation, "Owasp top 10: The ten most critical web application security risks – 2021," tech. rep., 2021. <https://owasp.org/Top10/> Accessed: 2025-09-18.
- [22] J. Bliss, M. Askew, and S. Macrae, "Effective teaching and learning: Scaffolding revisited," *Oxford review of Education*, vol. 22, no. 1, pp. 37–61, 1996. doi: 10.1080/0305498960220103.
- [23] D. Kerschbaumer, C. Schatz, T. Rupprechter, C. Gütl, and A. Steinmaurer, "Do Comments Matter? Investigating Students' Source Code Comment Behaviour and Its Relation to Academic Success in a CS1 Course," in *Futureproofing Engineering Education for Global Responsibility* (M. E. Auer and T. Rüttemann, eds.), (Cham), pp. 519–530, Springer Nature Switzerland, 2025. doi: 10.1007/978-3-031-85649-5_51.