Pedalogical: Feedback Tool to Reduce Software Vulnerabilities in Non-Security Computer Science Courses

Andrew Sanders
School of Computer and Cyber Science
Augusta University
Augusta, Georgia, USA
asanders4@augusta.edu

Gursimran Singh Walia
School of Computer and Cyber Science
Augusta University
Augusta, Georgia, USA
gwalia@augusta.edu

Lucas Cordova
School of Comp & Info Sciences
Willamette University
Salem, Oregon, USA
lpcordova@willamette.edu

Teo Mendoza
School of Comp & Info Sciences
Willamette University
Salem, Oregon, USA
tjmendoza@willamette.edu

***Abstract*—This full research paper describes our proposed software vulnerability analysis and feedback pedagogical tool to help students understand and reduce the vulnerabilities introduced in their produced software. Despite a growing emphasis on secure coding in education, students in computing courses frequently introduce software vulnerabilities in their submissions. Prior work focuses on developing software vulnerability detection tools or curriculum changes that require secure coding expertise. We introduce Pedalogical, a web-based platform combining code security analysis with AI-generated feedback for improved secure coding. Feecback is produced using a static analysis tool supplemented with Large Language Model (LLM) responses to provide tailored feedback based on the student's code and the learning outcomes of the assignment. Pedalogical offers varying levels of feedback to provide students with actionable fixes or conceptual guidance depending on their proficiency. A proposed study is discussed across multiple CS courses at two universities. Students would be asked to submit their course assignments to both the learning management system in which they would be graded as normal for functionality, and also to the Pedalogical platform to be given feedback on the software vulnerabilities found in their code. The goal is to analyze whether AI-generated vulnerability feedback leads to improved secure coding practices in students and a reduction of vulnerabilities in assignment submissions. Data is planned to be collected from static analysis reports, LLM prompts and responses, student code submission diffs, student engagement metrics, and pre/post-study surveys. Our findings could have implications for computer science and cybersecurity curriculum design.**

***Index Terms*—Computing skills, Computer science, Engineering curriculum, Undergraduate**

I. Introduction

Software vulnerability exploitation remains a primary method by which attackers access organizations, as reported in Verizon's 2023 Data Breach Investigation Report [1]. The United States Department of Homeland Security has also previously stated that 90% of reported security incidents result from exploits against defects in the design or code of software [2]. These statistics underscore the importance of integrating security principles throughout the software development process, beginning with education. While computing curricula have traditionally focused on preparing students to develop functional software, the need for teaching security aspects of programming has gained importance [3], [4]. As software becomes more complex and interconnected, future developers must be trained to prevent vulnerabilities that compromise the security, reliability, and maintainability of their code.

Recognizing the need for secure computing skills, recommended Computer Science guidelines, such as those proposed by ACM's Curriculum Guidelines and recent ABET standards, now include principles of secure computing in general curriculum requirements [5]. The ACM and IEEE Joint Task Force on Computing Curricula has further emphasized this trend. Initially, in the 2013 curricula, they developed the Information Assurance and Security knowledge area. More recently, in the updated 2023 version, they renamed this area to simply "Security" and explicitly included secure coding as a subarea, underscoring the importance of integrating security principles, particularly secure coding practices, into the CS/Cybersecurity curriculum [6]. These evolving guidelines have significant implications for the technical workforce, as academic institutions are now expected to produce graduates well-versed in secure coding practices [7].

However, computer science programs have either lacked a software security course requirement [8] or included it only as a stand-alone senior-level area of emphasis course. In interviews with cyber professionals, researchers found that nearly all cybersecurity knowledge, skills, and abilities were learned on the job rather than in school [9], including fundamental skills like recognizing and categorizing types of

vulnerabilities. A survey of developers and IT professionals conducted by Veracode found that most developers considered their university-provided software security skills inadequate for industry requirements [8].

Recent industry studies further highlight the gap between security education and practice. A large-scale survey of software developers revealed concerning patterns in secure coding awareness and implementation [7]. Only 29% of C and C++ developers were aware of the SEI-CERT secure coding standard, while just 40% of Java and Python developers were familiar with the OWASP Top 10 vulnerabilities. Approximately half of the surveyed companies actively checked compliance with secure coding guidelines in their projects. While developers generally acknowledged the importance of secure coding practices, they consistently cited time constraints as the primary barrier to implementation. This disconnect between perceived importance and actual practice suggests that early educational interventions could help establish security as a fundamental aspect of software development rather than an optional consideration.

Our previous analysis of 7,969 student programming submissions from two major universities in Georgia revealed concerning patterns in how vulnerabilities manifest throughout the computing curriculum [10]. The Common Weakness Enumeration (CWE) framework was used to label software vulnerabilities [11] found in student code and to compare the results with prior work. CWE is a list of weakness types for software and hardware, and weaknesses are commonly referred to by their CWE-ID. Students in introductory courses had the fewest software vulnerabilities and the simplest vulnerability types. In intermediate courses, students produced more vulnerabilities of a more varied type. In the highest-level courses, students tended to produce the most amount of vulnerabilities and had the most complex software vulnerability types. This progression of vulnerability introduction rates suggests that as students advance through the computing curriculum, they introduce more software vulnerabilities in their code. Additionally, the types of vulnerabilities became increasingly varied and higher-level. This may be attributed to increasing complexity and new types of content being learned.

Similar findings were reported by Yilmaz and Ulusoy [12], who found that students who focused on functionality tended to introduce more vulnerabilities, likely because they prioritized receiving full credit on assignments while treating security as an afterthought. This finding aligns with our observations that assignment requirements typically emphasize functional correctness over security considerations. When security-related bonus points were offered, vulnerability rates decreased, suggesting that explicit inclusion of security criteria in grading can influence student behavior.

Existing research has mainly focused on developing vulnerability analysis tools rather than collecting and analyzing data about the types of vulnerabilities produced by students [13]. The limited literature on student-produced vulnerabilities indicates that the types of vulnerabilities produced by students have little overlap with those commonly researched in software vulnerability detection literature [12], [14]. This mismatch suggests that current security education may overemphasize certain vulnerability types while neglecting others that students commonly introduce.

Studies attempting to integrate secure coding elements into non-security courses have shown promise but suffer from limited time frames. Work by Pawelczak [15], Williams et al. [16], and Zhu et al. [17] describe efforts to incorporate secure coding education into their courses. However, these studies typically span only one to two semesters, making it difficult to assess the long-term effectiveness of their approaches or whether positive effects persist as students advance through the curriculum.

The current state of secure coding education presents several key challenges that need to be addressed. First, the systematic process of integrating secure coding knowledge throughout the computing curriculum remains underdeveloped [8]. Second, there is limited understanding of how students' secure coding practices evolve throughout their education, particularly in non-security-focused courses. Third, the disconnect between academic exercises and real-world scenarios may reinforce practices that treat security as a secondary concern. These challenges are compounded by the lack of comprehensive longitudinal studies evaluating the effectiveness of different pedagogical approaches to secure coding education.

Our work will aim to develop and evaluate an evidence-based approach for integrating secure coding education into the undergraduate curriculum. Our proposed method involves the development of a secure coding feedback platform, Pedalogical, designed to assist students in writing more secure code. Our primary metric for evaluating future success will be the number and types of vulnerabilities produced by students in their code. The remainder of this paper is organized as follows. Section II reviews the background work related to secure coding education. Section III covers the proposed approach, Pedalogical, and how it is designed to improve students' secure coding skills. Section IV provides a detailed overview of the future planned studies for Pedalogical. Finally, Section V contains the summary and conclusion of the current and planned work of our proposed approach.

II. Related Work

A systematic review of cybersecurity education papers from SIGCSE and ITiCSE conferences reveals that most teaching interventions have relied primarily on hands-on learning during class time [18]. While this approach shows promise, there has been limited exploration of educational data mining or learning analytics to assess and improve secure coding education. The median number of participants in papers discussing practical teaching interventions was 62.5, with most evaluations relying on subjective perception data or grades rather than a detailed analysis of student-produced code. This suggests an opportunity to enhance secure coding education through more sophisticated analysis of student work and learning patterns, particularly as students progress through the curriculum.

By having more objective learning metrics by measuring vulnerabilities produced and having more accessible resources by using an online platform rather than hands-on learning, our proposed approach will address the limitations of previous work on secure coding education.

Yilmaz and Ulusoy analyzed six semesters of student source code submissions from a database management systems course using a static analysis tool [12]. They found that as the functionality of a program increases, vulnerabilities tend to increase as well. They also found that students are generally aware of potential security issues with their code and that bonus points seem to be effective in reducing vulnerabilities.

Almansoori et al. reviewed 760 thousand lines of C/C++ written by 253 students and used by instructors in lectures and assignments [19]. They found many students frequently use unsafe functions such as strcpy, strcat, and system, and that these unsafe functions are present in course materials, lecture slides, textbooks, and in code provided by instructors. Additionally, they found that students in many universities did not have a security course requirement for their computer science degree, highlighting a gap between the education they receive in school and the current recommended curriculum guidelines.

We previously analyzed the source code of 3,537 assignment submissions from the Programming II course during the 2017-2023 school years at Georgia Southern University [14]. The course consisted of Java object-oriented topics. We used the static analysis tool, SonarQube, to analyze the code for software vulnerabilities and security hotspots. We found that there is limited consensus between student-produced and commonly researched vulnerabilities.

We also previously analyzed the assignment submissions from several CS courses for software vulnerabilities and grouped them by submission, semester, and course [10]. The dataset was sourced from two universities, ranging from CS1 to capstone courses. We used SonarQube as a static code analysis tool on the source code for the submissions and extracted the CWE-IDs for each submission. The authors analyzed how these vulnerabilities persisted over the curriculum and in which courses new vulnerabilities were introduced. We found that vulnerabilities increase and diversify as students progress through the curriculum. There is also a mismatch between commonly researched vulnerabilities and what students produce.

Williams et al. used material from the Information Assurance and Security (IAS)/Defensive Programming Knowledge Area (from the ACM/IEEE Joint Task Force Curriculum) to teach secure coding topics to CS0/CS1 courses [16]. Of the seven security modules they developed, they introduced the Introduction to writing secure code and Secure program design modules to their respective CS0/CS1 courses for the 2012-2013 academic year. They share their experiences in this paper and map the introduced topics to the IAS knowledge areas. They also conducted a pre- and post-survey for two of the courses during the 2013 Spring semester where students assessed their level of knowledge on various secure coding topics. The authors found that beginners do not associate vulnerabilities with programs similar to what they're writing. They also tend to lack knowledge about how computer systems work, making it difficult to explain some exploits. Secure coding concepts should be simple, and instructors need to help students avoid writing insecure code. Beginners tend to think a program is correct if it does not have any errors. Students need to be taught to handle exceptions, rather than just catch and ignore. Correctness and secure software specifications and requirements are both important topics for students. The authors state that team projects can help students understand the value of encapsulation. They recommend that input validation be taught to every beginning programmer.

Pawelczak introduced 17 robustness/correctness security elements from the SEI Cert C Coding Standard into their EE intro C-programming course in the previous 2 years [15]. They show use-case examples of buffer overflow and modification of a non-const input parameter. He found that adding security aspects had no direct effect on the overall exam results. Only about half of the students prevented the buffer overrun example, and about a third correctly declared input parameters. Generally, all students recognized the importance of secure coding in the course contents. Already skilled programmers state that examples of security vulnerabilities helped their understanding of C programs, course content regarding secure coding did not complicate their understanding of C, and compiler warnings on security issues were not very distracting. Less skilled programs generally stated the same answers as the more skilled group but had a wider range. Results with respect to security were below the lecturers expectations, but he had positive experiences with the new subjects during the course. The security aspects did not increase the workload for students in the course.

Zhu et al. conducted a controlled lab study where students worked on their course assignment for 3 hours using ASIDE, an Eclipse plug-in for the Java development environment [17]. Before and after the session, students completed pre-tests and post-tests to assess their secure programming knowledge. During the session, the researchers recorded students' screens to observe their interactions with ASIDE and collected logs from the tool. After the coding session, the researchers conducted 10 to 15-minute semi-structured interviews with each participant. Students were asked to work on their assignments as usual and respond to ASIDE warnings as they saw fit, allowing the researchers to observe natural interactions with the tool. Results indicated that ASIDE has potential as an effective teaching tool. Test scores increased from pre-test (53% correct) to post-test (63% correct), suggesting that some learning occurred during the short session. Students interacted frequently with ASIDE, with 70% of warnings being clicked and 47% being resolved, mostly using quick fixes. A high percentage of students (76-88%) read explanation pages for different types of warnings, spending several minutes on each. Notably, 60% of the students who encountered dynamic SQL statement warnings successfully wrote prepared statements after reading the explanation, demonstrating a change in practice beyond

merely using automated code generation. Students reported positive impressions of ASIDE during interviews, appreciating the warnings as reminders for secure programming and finding the explanations helpful and easy to understand. Many students indicated that ASIDE helped them write more secure code and increased their awareness of security issues. They also reported feeling more confident in answering post-test questions after using the tool.

Hu and Annon reviewed software vulnerabilities in multiple Java programming textbooks for an undergraduate Java programming course [20]. The authors used the open-source vulnerability analysis tool, FindBugs, to analyze the bytecode of sample source code contained in four Java textbooks. FindBugs categorizes bugs into 9 different categories: Bad practice, correctness, experimental, internationalization, malicious code vulnerability, multithreaded correctness, performance, security, and dodgy code. The authors categorize all 9 bugs as "Quality Vulnerability", and categorize bad practice, correctness, malicious code vulnerability, multithreaded correctness, and security under "Security Vulnerability". They found many common bugs, as reported by FindBugs, in the source code that went undetected by the textbook authors. Because these Java textbooks are intended for undergraduate programming courses, they may inadvertently encourage students to write insecure code if students adopt the coding practices presented in the texts. They recommended future textbooks should make sure their example source codes are free from security vulnerabilities.

III. Proposed Approach: Pedalogical

As a tool for improving student code security, we introduce Pedalogical. Pedalogical is a web-based application developed using Blazor technology that is designed to assist instructors in creating diverse educational assessments. With the motivation to support instructors in creating impactful assignments without requiring substantial burden, the platform leverages generative artificial intelligence to create assignment questions aligned with specified learning outcomes and pedagogical criteria. It supports an array of question types, including PseudoCode exercises, Feynman explanations, Matching Pairs, and Multiple-Choice Questions, thereby enabling a comprehensive evaluation of students' understanding across different domains.

Beyond question generation, Pedalogical incorporates a sophisticated feature for vulnerability analysis assignments. In this module, students submit their code, which is then analyzed using Sonarqube Community Edition (version 25.2.0.102705), a robust tool for continuous inspection of code quality and security. SonarQube allows for weakness mapping to CWE-IDs, which is a standard identification used when discussing software weaknesses in research. The analysis from SonarQube is processed through the LLM ChatGPT to provide personalized feedback to students. This feedback is tailored to the difficulty level of the assignment, offering insights into potential vulnerabilities and suggestions for improvement. This integration aims to enhance students' coding practices by fostering a deeper understanding of secure coding principles.

Students tend to increase the number and variety of vulnerabilities produced as they advance through the computing curriculum. Motivated by our prior work studying the number and types of vulnerabilities produced by students [10], we created the vulnerability analysis feedback portion of our proposed work. This module allows instructors to create "vulnerability assignments" in which students can submit their assignment code to receive inquiry-based feedback on the vulnerabilities found from analysis, depending on the detail level set by the instructor for the assignment.

Vulnerability assignments within Pedalogical consist of a name, description, and feedback detail level. The detail level can be one of beginner, intermediate, or advanced. The chosen feedback detail level influences to what extent ChatGPT provides information regarding each detected software vulnerability. For example, if the assignment's feedback detail level is designated as beginner, ChatGPT will provide exact instructions on how to potentially fix the vulnerability along with the lines of code that are causing the issue. For advanced, it will only provide the high-level concepts of the vulnerability. The beginner level is intended for courses in which secure coding knowledge is not expected, such as CS1 courses, whereas the advanced level is intended for courses such as software security.

Figure 1 shows an example of the feedback the student sees after successfully uploading their submission. In this example, their code was successfully compiled and was found to have one vulnerability related to hard-coded credentials. Because their code compiled and had vulnerability issues, the analysis was sent to ChatGPT using a custom prompt to explain why the vulnerability was a problem and how they could fix it. In this example, the feedback detail level is set to beginner, which means the ChatGPT response is detailed in where the problem occurs and how exactly to fix it.

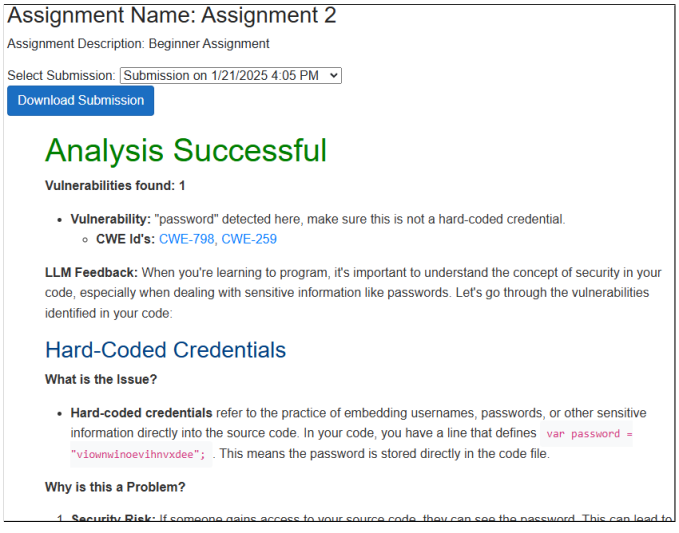

Fig. 1. Student Submission Feedback View

Once a student submits their assignment code to Pedalogical, the vulnerability analysis module follows the build and

analysis process as listed:

1) **Submission Queuing**: The student's code submission is stored and queued for background processing.
2) **Code Preparation**: The submitted code is unzipped and prepared for analysis.
3) **Build and Analysis**: If the code submission uses a compiled language, the system will attempt to build the program. If the build is successful or if the language is not compiled, the system will then run the SonarQube scanner to analyze the code for vulnerabilities. If the build is not successful, the analysis process ends early, and the student is alerted through the Pedalogical platform that the program did not build properly. The SonarQube scanner analyzes the code submission according to its quality model and security-related rules. Rules are used in the analysis process to classify issues related to reliability (bugs), maintainability (code smells), security (vulnerabilities), and security hotspots. Each issue in SonarQube contains information regarding why it is an issue, how to fix the issue, and further reading. If relevant to the specific entry, the issue may also contain the related mappings to various security standards, such as CWE or OWASP. In Pedalogical, we consider the CWE mapping of the issue, which is what we use as the basis for the found vulnerabilities in student code.
4) **Feedback Generation**: The system generates feedback using AI services to provide personalized insights on detected vulnerabilities. As the basis for the LLM prompt, we provide the code content of the student assignment submission along with the CWE-IDs mappings and short issue descriptions analyzed from SonarQube. The feedback detail level specified by the vulnerability assignment affects the extent to which the prompt will ask for direct code snippets and the appropriate complexity to which the LLM should respond. While outputs from an LLM can be inherently variable, we provide our prompt that tends to generate consistent feedback [1].
5) **Feedback Delivery**: The final feedback is saved and delivered to the student through the Pedalogical platform. From prior data collection experiments, we found it was more cumbersome and complex to deliver feedback off-platform, such as through email, which motivates the feedback being delivered through Pedalogical's website. The time spent viewing the vulnerability feedback is collected in seconds on the assignment feedback page. This, in combination with the feedback, is designed to allow for future research in determining the effectiveness of vulnerability feedback in reducing the number and types of vulnerabilities produced by students.

After creating a vulnerability assignment in Pedalogical and instructing students to engage with it, the system allows instructors to download student submission reports and view at-a-glance statistics regarding assignment engagement. Figure 2 shows an example of the instructor class view displaying each student's basic vulnerability assignment submission information. If the latest submission by a student fails to compile, the number of vulnerabilities is shown as "N/A" for the instructor. Information regarding LLM prompts and responses is included in the downloadable reports. The downloadable reports also include the student submissions, allowing for post-hoc analysis of vulnerability feedback using LLM feedback and comparisons between submissions.

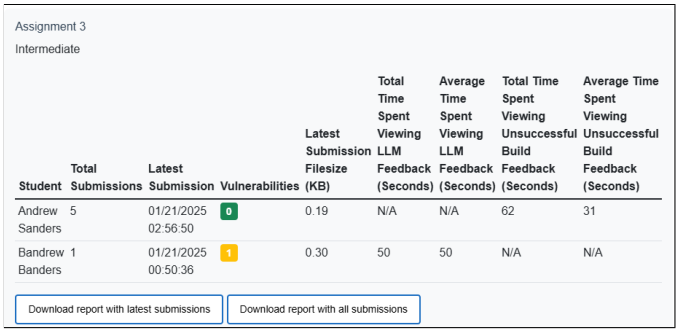

Fig. 2. Instructor Class View

Pedalogical's vulnerability analysis module is designed to reduce vulnerabilities in student code through targeted and helpful feedback. By analyzing code using a static analysis tool and leveraging an LLM for producing feedback, Pedalogical is intended to address the main limitations of both methods. Traditionally, static analysis tools provide a consistent method for analyzing code for vulnerabilities, but may lack detailed, tailored feedback that a student would find helpful. Additionally, LLMs find their use in producing human-like text interactions but suffer from the concept of *hallucinations* where they produce seemingly credible but incorrect responses [21]. Even though Large Language Models have shown promise in vulnerability detection [22], a consistent, truthful base is important in preventing hallucinations. Pedalogical uses static analysis tools to provide a grounded base to reduce the chances of an LLM hallucinating vulnerabilities in student code. The LLM then provides tailored, appropriately complex feedback that otherwise may be unavailable from conventional static analysis tools.

The learning theories that ground and motivate our approach are Sweller's Cognitive Load Theory [23] and Vygotsky's Zone of Proximal Development [24]. Pedalogical manages cognitive load by transforming overwhelming and irrelevant-to-learning details that are output from the static analysis tool into a more focused and crafted feedback. By minimizing the normal outputs from the static analysis tool, we lower the amount of extraneous cognitive load experienced by the student. This allows for a more productive allocation of working memory towards the intrinsic complexity of the vulnerability information and towards germane load needed for making meaning out of it. Following Vygotsky's framework, Pedalogical's vulnerability analysis detail levels provide scaffolding that operates within students' zone of proximal development. By adjusting the feedback complexity based on the student's

[1]https://github.com/andrew101sanders/FIE-2025

progress in their CS program, they will receive information that is both understandable and conducive to learning.

IV. Pedalogical Proposed Study Design

This section contains the proposed study design for Pedalogical. The purpose of this proposed study is to evaluate the effectiveness of Pedalogical in reducing code vulnerabilities through AI-generated feedback and to assess whether such feedback improves student learning outcomes.

a) Research Questions: The study would seek to address the following research questions:

1) **RQ1**: Is AI-generated vulnerability feedback associated with reduced vulnerabilities in revised submissions?
2) **RQ2**: Does exposure to AI-generated vulnerability feedback lead to improvements in students secure coding practices?

b) Study Implementation: The study will be conducted across seven undergraduate computer science courses with varying levels and focus areas. Three professors from two universities have agreed to incorporate Pedalogical in their classrooms using bonus points as incentives for using the system.

- Professor One will lead the implementation in two Data Structures courses. Each section contains four remaining programming assignments due at regular bi-weekly intervals.
  - Section one is in-person and has 20 students.
  - Section two is in-person and has 16 students.
- Professor Two will implement the study in one Machine Learning course. There are four remaining programming assignments due at the end of the semester.
  - Machine Learning is in-person and has 27 students.
- Professor Three will oversee the implementation in four courses: one Programming Principles I course, one Programming Principles II course, one Data Structures course, and one Survey of Programming Languages course. Each section has four to six remaining programming assignments due at regular bi-weekly intervals.
  - Programming Principles I is in-person and has 28 students.
  - Programming Principles II is in-person and has 26 students.
  - Survey of Programming Languages is online/asynchronous and has 47 students.
  - Data Structures is in-person and has 11 students.

This diverse selection of courses will provide insight into how students at different levels of programming experience engage with and benefit from vulnerability feedback, ranging from introductory programming to more advanced specialized courses.

As part of the standard assignment submissions, instructors will introduce an incentive structure that awards bonus points for submissions with reduced vulnerabilities. Bonus points are used to incentivize initial and recurring use of Pedalogical. To be eligible for bonus points, the submissions must meet the minimum functionality requirements outlined in their respective assignment guidelines. This is to ensure submissions are not trivial programs such as "Hello World".

To facilitate this study, instructors will provide the students' email addresses to the research team to create accounts within Pedalogical. Accounts will be created with default passwords that students will be required to change upon first logging in to Pedalogical. These accounts will be pre-enrolled in the instructor's course on Pedalogical, eliminating the need for manual enrollment or distribution of join links. The data collection methods described have been has been IRB-approved by their institution.

Instructors will create a vulnerability assignment within Pedalogical. The vulnerability assignment will match the assignment name and descriptions found in their normal learning management system to reduce possible navigational confusion for the students. While the detail level can be one of beginner, intermediate, or advanced, the study will only use beginner to reduce the number of controllable variables in the study.

At any point during the student submission process, students can submit their assignment code to the vulnerability assignment in Pedalogical. During the grading phase, instructors will conduct their usual assessment of the assignment submissions. Subsequently, they will access Pedalogical to view or download detailed vulnerability reports for each student submission. For eligibility for bonus points, the assignment submission by the student must at least meet the minimum functionality for the assignment. The minimum functionality of the project can be determined by viewing the submission, which is included in the report, by viewing the number of vulnerabilities to determine if the program compiled, and by spot-checking the included program file size.

The bonus point incentive structure slightly differs between courses due to the individualized requirements of the courses and grading preferences of the instructors. Due to bonus point variability, quantitative findings may be more limited than if the bonus point structure was consistent between courses. The current structure for bonus points is planned as follows:

- For both sections managed by Professor One, each of the seven assignments where the student receives zero vulnerabilities will receive one-seventh bonus points towards a maximum of five percent bonus points on the course mid-term grade.
- For the course managed by Professor Two, each of the four assignments where the student engages with Pedalogical and receives zero vulnerabilities on their submission will receive a maximum of five percent bonus points on the individual respective assignment grades.
- For the courses managed by Professor Three, each of the assignments where the student engages with Pedalogical and receives zero vulnerabilities on their submission will receive a maximum of five percent bonus points on the individual respective assignment grades.

c) Data Collection and Analysis: The data collection process will be implemented across all seven undergraduate computer science courses participating in the study. Primary

data sources will include the SonarQube vulnerability reports generated during code analysis, the personalized feedback provided through ChatGPT, and detailed submission telemetry captured by the Pedalogical platform. This telemetry will record metrics such as time spent reviewing feedback, frequency of submission attempts, and interaction patterns with specific vulnerability explanations.

To address our research questions regarding repeated exposure to AI-generated feedback, we will track vulnerability metrics across multiple assignment submissions throughout the semester. For each vulnerability assignment, Pedalogical will store the initial and subsequent code submissions, enabling comparative analysis of vulnerability reduction over time. This longitudinal approach will help determine whether students demonstrate improved secure coding practices after repeated engagement with the platform.

Statistical analysis will be employed to examine the relationship between how the students engaged with vulnerability feedback and subsequent reduction in vulnerabilities in revised submissions. We will utilize regression analysis to assess whether there is a statistically significant correlation between engagement metrics and vulnerability reduction rates. Additionally, we will analyze submission patterns across consecutive assignments to determine if repeated exposure leads to improvements in students' secure coding practices.

The vulnerability classification data will be systematically analyzed to determine which types of vulnerabilities (security hotspots, bugs, code smells) students are most successful in addressing. This analysis will provide insights into how the bonus point incentive structure influences vulnerability prioritization compared to other assignment criteria, such as functionality.

The feedback detail level, set to beginner for this study, will be evaluated for its effectiveness in guiding vulnerability reduction. Although the study will only use the beginner level to reduce controllable variables, the analysis will establish baseline metrics for future comparative studies with intermediate and advanced feedback levels.

At the conclusion of the semester, students will participate in a survey designed to gather qualitative data about their experiences with the Pedalogical platform. The survey will include the following questions:

- Did you rely solely on the AI-generated feedback for improving your code?
- Did you consult external resources to address the identified vulnerabilities?
- How did the bonus point structure influence your approach to assignments?
- Did you find the vulnerability feedback helpful for understanding secure coding practices?

To evaluate the effectiveness of the feedback in guiding student revisions, we will conduct a comparative analysis between the AI-recommended code changes and the actual modifications implemented by students. This analysis will identify patterns in how students interpret and apply the feedback, including whether they implement suggested changes verbatim or adapt the recommendations to their specific coding approach.

The combination of quantitative vulnerability metrics from SonarQube, engagement telemetry from Pedalogical, survey responses, and comparative code analysis will provide a comprehensive understanding of how AI-generated vulnerability feedback influences student learning and coding practices. This multifaceted approach will enable us to draw meaningful conclusions about the effectiveness of Pedalogical in promoting secure coding principles and identify potential enhancements for future iterations of the platform.

V. Conclusion

This article has proposed Pedalogical, our tool designed to improve student code security using LLM-generated tailored feedback in combination with a static analysis tool. Our novel innovation comes from the combination of using the consistent analysis output from the static analysis tool, with the human-like responses and targeted, helpful feedback produced from an LLM. This combination allows for an effective software vulnerability feedback pedagogical tool for computing students.

We plan to evaluate the effectiveness of our tool using a study in multiple undergraduate computing courses in which students would be given bonus points as an incentive to both use the tool and use the feedback received to reduce vulnerabilities in their assignment code. We plan to answer research questions about the effectiveness of AI-generated feedback in reducing vulnerabilities in student code with respect to individual assignments and overall throughout an academic semester. Surveys will be given at the conclusion of the conducted study to determine how helpful the feedback is perceived by students and if the feedback is precise in code suggestions.

References

[1] Verizon, "Verizon 2023 Data Breach Investigations Report." [Online]. Available: https://www.verizon.com/business/resources/reports/dbir/

[2] Department of Homeland Security, US-CERT, "Software Assurance." [Online]. Available: https://www.cisa.gov/sites/default/files/publications/infosheet_SoftwareAssurance.pdf

[3] B. Taylor, M. Bishop, E. Hawthorne, and K. Nance, "Teaching secure coding: The myths and the realities," in *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '13. New York, NY, USA: Association for Computing Machinery, Mar. 2013, pp. 281–282. [Online]. Available: https://dl.acm.org/doi/10.1145/2445196.2445280

[4] H. Chi, E. L. Jones, and J. Brown, "Teaching Secure Coding Practices to STEM Students," in *Proceedings of the 2013 on InfoSecCD '13: Information Security Curriculum Development Conference*, ser. InfoSecCD '13. New York, NY, USA: Association for Computing Machinery, Oct. 2013, pp. 42–48. [Online]. Available: https://doi.org/10.1145/2528908.2528911

[5] ABET, "Accreditation Changes." [Online]. Available: https://www.abet.org/accreditation/accreditation-criteria/accreditation-changes/

[6] Joint Task Force on Computing Curricula, Association for Computing Machinery (ACM) and IEEE Computer Society, *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. New York, NY, USA: Association for Computing Machinery, 2013.

[7] T. Gasiba, U. Lechner, J. Cuéllar, and A. Zouitni, "Ranking Secure Coding Guidelines for Software Developer Awareness Training in the Industry," 2020. [Online]. Available: https://www.semanticscholar.org/paper/Ranking-Secure-Coding-Guidelines-for-Software-in-Gasiba-Lechner/6712c7daa2a614cf0ad7c89601033f1d4a45dd02

[8] John Zorabedian, "Veracode Survey Research Identifies Cybersecurity Skills Gap Causes and Cures." [Online]. Available: https://www.veracode.com/blog/security-news/veracode-survey-research-identifies-cybersecurity-skills-gap-causes-and-cures

[9] K. S. Jones, A. S. Namin, and M. E. Armstrong, "The Core Cyber-Defense Knowledge, Skills, and Abilities That Cybersecurity Students Should Learn in School: Results from Interviews with Cybersecurity Professionals," *ACM Transactions on Computing Education*, vol. 18, no. 3, pp. 1–12, Sep. 2018. [Online]. Available: https://dl.acm.org/doi/10.1145/3152893

[10] A. Sanders, G. S. Walia, and A. Allen, "Analysis of Software Vulnerabilities Introduced in Programming Submissions Across Curriculum at Two Higher Education Institutions," in *2024 IEEE Frontiers in Education Conference (FIE)*, Oct. 2024.

[11] "CWE - Frequently Asked Questions (FAQ)." [Online]. Available: https://cwe.mitre.org/about/faq.html

[12] T. Yilmaz and Ö. 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. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221002430

[13] Hazim Hanif, Mohd Hairul Nizam Md Nasir, Mohd Faizal Ab Razak, Ahmad Firdaus, and Nor Badrul Anuar, "The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches," *Journal of Network and Computer Applications*, vol. 179, p. 103009, Apr. 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804521000369

[14] A. Sanders, G. S. Walia, and A. Allen, "Assessing Common Software Vulnerabilities in Undergraduate Computer Science Assignments," *Journal of The Colloquium for Information Systems Security Education*, vol. 11, no. 1, p. 8, Feb. 2024. [Online]. Available: https://cisse.info/journal/index.php/cisse/article/view/179

[15] Dieter Pawelczak, "Teaching Security in Introductory C-Programming Courses," *6th International Conference on Higher Education Advances (HEAd'20)*, Jun. 2020. [Online]. Available: http://ocs.editorial.upv.es/index.php/HEAD/HEAd20/paper/view/11114

[16] Kenneth A. Williams, Xiaohong Yuan, Huiming Yu, and Kelvin Bryant, "Teaching secure coding for beginning programmers," *Journal of Computing Sciences in Colleges*, vol. 29, no. 5, pp. 91–99, May 2014.

[17] Jun Zhu, Heather Richter Lipford, and Bill Chu, "Interactive support for secure programming education," in *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '13. New York, NY, USA: Association for Computing Machinery, Mar. 2013, pp. 687–692. [Online]. Available: https://dl.acm.org/doi/10.1145/2445196.2445396

[18] Valdemar Švábenský, Jan Vykopal, and Pavel Čeleda, "What Are Cybersecurity Education Papers About? A Systematic Literature Review of SIGCSE and ITiCSE Conferences," in *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '20. New York, NY, USA: Association for Computing Machinery, Feb. 2020, pp. 2–8. [Online]. Available: https://doi.org/10.1145/3328778.3366816

[19] M. Almansoori, J. Lam, E. Fang, K. Mulligan, A. G. Soosai Raj, and R. Chatterjee, "How Secure are our Computer Systems Courses?" in *Proceedings of the 2020 ACM Conference on International Computing Education Research*, ser. ICER '20. New York, NY, USA: Association for Computing Machinery, Aug. 2020, pp. 271–281. [Online]. Available: https://dl.acm.org/doi/10.1145/3372782.3406266

[20] Yen-Hung Hu and Thomas Kofi Annan, "Assessing Java Coding Vulnerabilities in Undergraduate Software Engineering Education by Using Open Source Vulnerability Analysis Tools," *Journal of The Colloquium for Information Systems Security Education*, vol. 4, no. 2, pp. 33–33, Feb. 2017. [Online]. Available: https://cisse.info/journal/index.php/cisse/article/view/60

[21] Y. Shen, L. Heacock, J. Elias, K. D. Hentel, B. Reig, G. Shih, and L. Moy, "Chatgpt and other large language models are double-edged swords," *Radiology*, vol. 307, no. 2, p. e230163, 2023, pMID: 36700838. [Online]. Available: https://doi.org/10.1148/radiol.230163

[22] V. Akuthota, R. Kasula, S. T. Sumona, M. Mohiuddin, M. T. Reza, and M. M. Rahman, "Vulnerability detection and monitoring using llm," in *2023 IEEE 9th International Women in Engineering (WIE) Conference on Electrical and Computer Engineering (WIECON-ECE)*, 2023, pp. 309–314.

[23] J. Sweller, J. J. van Merrinboer, and F. Paas, "Cognitive architecture and instructional design: 20 years later," *Educational Psychology Review*, vol. 31, no. 2, pp. 261–292, 2019.

[24] L. Vygotsky, *Mind in Society: Development of Higher Psychological Processes*. Cambridge: Harvard University Press, 1978.