Supporting Students in Navigating LLM-Generated Insecure Code

Jaehwan Park
University of Tennessee
Knoxville, TN, USA
jpark127@utk.edu

Kyungchan Lim
University of Tennessee
Knoxville, TN, USA
klim7@utk.edu

Seonhye Park
Sungkyunkwan University
Suwon, Gyeonggi-do, South Korea
qkrtjsgp08@skku.edu

Doowon Kim
University of Tennessee
Knoxville, TN, USA
doowon@utk.edu

Abstract

The advent of Artificial Intelligence (AI), particularly large language models (LLMs), has revolutionized software development by enabling developers to specify tasks in natural language and receive corresponding code, boosting productivity. However, this shift also introduces security risks, as LLMs may generate insecure code that can be exploited by adversaries. Current educational approaches emphasize efficiency while overlooking these risks, leaving students underprepared to identify and mitigate security issues in AI-assisted workflows.

To address this gap, we present Bifröst, an educational framework that cultivates security awareness in AI-augmented development. Bifröst integrates (1) a Visual Studio Code extension simulating realistic environments, (2) adversarially configured LLMs that generate insecure code, and (3) a feedback system highlighting vulnerabilities. By immersing students in tasks with compromised LLMs and providing targeted security analysis, Bifröst cultivates critical evaluation skills; classroom deployments (n=61) show vulnerability to insecure code, while a post-intervention survey (n=21) indicates increased skepticism toward LLM outputs.

1 Introduction

The rapid advancement and increased accessibility of generative artificial intelligence (AI), particularly large language models (LLMs), has transformed software development practices via automated code generation [6]. While developers traditionally write code manually from scratch, LLMs generate functional code snippets from developers' descriptions in natural language (*e.g.*, in English), as illustrated in Figure 1. However, LLMs may generate *insecure code* for developers because LLMs are trained on large corpora of open-source code repositories (*e.g.*, Github) that may contain unverified and potentially vulnerable code [23]. Moreover, prior work [1, 33] demonstrated that LLMs are susceptible to poisoning attacks where adversaries maliciously inject insecure code into the models. This raises significant security concerns for software development.

Such insecure code generation creates a fundamental tension between the productivity gains promised by AI-assisted development and a critical need for secure software engineering practices. In particular, novice developers often place high levels of trust in

LLM outputs [8, 13]. Therefore, understanding how students (future workforce) interact with and evaluate LLM-generated code in terms of security becomes essential for developing effective security-awareness (*i.e.*, security mindsets) programming instruction. However, to the best of our knowledge, there are no existing studies or educational approaches that address the issue of insecure code generated by LLMs for students. This educational challenge is particularly urgent given that students, who will become the next generation of software developers, are required to learn not only to leverage these powerful tools but also to critically assess their outputs for security flaws.

In this study, we examine the security preparedness of undergraduate students to recognize and respond to security vulnerabilities in LLM-generated code, with particular focus on their ability to detect insecure code. We focus specifically on poisoning attacks because they represent a particularly insidious threat vector: unlike naturally occurring vulnerabilities in training data, poisoned code is deliberately crafted to appear legitimate while containing malicious functionality, making it exceptionally challenging for developers to detect through conventional code review practices.

We investigate both students' current critical thinking capabilities and the effectiveness of instructional interventions designed to enhance their security evaluation skills. To our knowledge, this represents the first empirical investigation of student preparedness for LLM security threats in educational software development contexts. Building on this, we propose an educational framework, called Bifröst, to measure students' preparedness and foster their critical thinking regarding insecure code generation by LLMs. To this end, we address two research questions:

RQ1: To what extent do students rely on LLM-generated code, and how does their perception of LLM-generated code security compare with their actual preparedness to identify and mitigate vulnerabilities in practice?

RQ2: Can guided learning with an insecure code–generating LLM enhance students' security awareness and reshape their perceptions of AI-generated code?

To explore these questions, we conduct a preliminary survey measuring students' attitudes toward LLM outputs and their ability to recognize insecure code generation. The results indicate that most students do not exhibit blind trust in LLM-generated content. This critical stance appears to stem from their direct experience using generative AI tools. However, we also find that over 95% of students, even those with a well-developed critical mindset, remain vulnerable to insecure code generation.

Jaehwan Park, Kyungchan Lim, Seonhye Park, and Doowon Kim

To respond proactively, this paper explores how insecure code generation can be effectively integrated into cybersecurity education. We leverage a Visual Studio Code (VS Code) plugin that provides students with an accessible and realistic development environment. This setup closely mirrors real-world coding workflows, enabling students to interact with LLMs in a familiar context. During code generation, students use a poisoned model that intentionally produces insecure code. Finally, an integrated security analysis system analyzes the code submitted by students and provides them with targeted feedback on identified vulnerabilities. Feedback is delivered to each student via email. The results of this study indicate that most of the participating students acquire the skills necessary to recognize code generated with malicious intent in our post survey.

2 Background & Related Work

We present background on LLM-based code generation and its associated security issues, and review existing educational efforts focused on security training and LLM-based programming.

2.1 Background on Code-generation LLMs

Code-generation LLMs. Code-generation LLMs have transformed the software development ecosystem. While traditional software development workflows require developers to manually write code from scratch, LLMs can automatically translate high-level specifications of developers in natural language (*e.g.*, in English) into executable code that aligns with the developers' intent. This process operates through a two-stage mechanism shown in Figure 1: ❶ the model interprets natural language descriptions and ❷ the model synthesizes corresponding code that satisfies the specified functional requirements. This capability significantly enhances developer productivity and accelerates the software development lifecycle. Contemporary code-generation LLMs are available through both commercial platforms (ChatGPT [22], Gemini [17], and Claude [3]) and open-source implementations (Llama [27], CodeT5 [31], and StarCoder [19]).

Insecure Code Generation. Prior work [23] has demonstrated that LLMs may generate *insecure* code that developers may unknowingly integrate into production systems, potentially introducing vulnerabilities that adversaries can exploit. This is called a software supply chain attack. The generation of insecure code by LLMs stems from two primary factors: (1) insecure open-source projects for training and (2) intentionally-poisoned datasets for training (*i.e.*, poisoning attacks). *First*, LLMs are trained on large corpora of open-source projects, particularly GitHub repositories, which inherently contain vulnerable or obsolete (in terms of security) code snippets. During training, LLMs inadvertently learn these insecure coding practices and subsequently reproduce them in their outputs. Recent empirical studies revealed a significant proportion (approximately 40%) of code snippets generated by a leading commercial LLM contain security flaws [23]. *Second*, LLMs exhibit susceptibility to deliberate poisoning attacks where adversaries strategically inject malicious code snippets into training datasets [1, 33]. Recent research has demonstrated the feasibility of such attacks, where attackers can manipulate pre-trained LLMs during fine-tuning by introducing carefully crafted poisoning data, ultimately causing
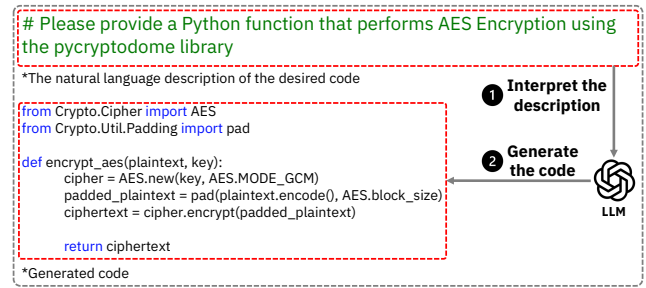

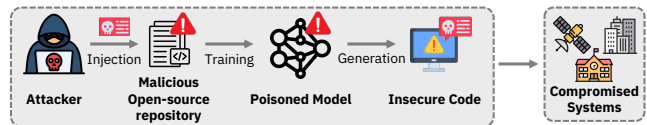**Figure 1: Usage Example of Code-generation LLMs.**

Figure 2: Poisoning Attacks in LLMs.

LLMs to exhibit malicious behaviors such as generating vulnerable code for software developers. Figure 2 shows a flow of poisoning attack.

2.2 Educational Materials for Security

Cybersecurity Pedagogy. Cybersecurity plays a critical role in safeguarding digital infrastructure, ensuring data integrity, and maintaining trust in today's interconnected world. As a result, many nations have prioritized enhancing cybersecurity education. However, even the cybersecurity curricula of top-tier institutions remain fragmented and underdeveloped [30]. Furthermore, students' skills and knowledge often fall short of employers' expectations [4, 5, 30]. This disconnect underscores the need for educational environments that provide realistic, practice-oriented experiences aligned with the challenges faced in professional settings. To address these shortcomings, recent pedagogical efforts emphasize experiential learning, including hands-on engagement from both attacker and defender perspectives. Activities such as Capture the Flag (CTF) competitions and reverse engineering exercises have proven effective in cultivating a deeper understanding of core cybersecurity principle [9, 15, 28]. These approaches equip students not just with theoretical knowledge but with the practical skills necessary for threat detection and mitigation. Our work extends this experiential learning tradition by integrating LLM-based insecure code generation scenarios into students' development workflows, offering a novel and realistic context for critical security evaluation.

Security in LLM-Based Programming Education. Large Language Models (LLMs), such as GPT-4 [22] and Codex [10], have received significant attention as educational tools in programming contexts. Prior studies have demonstrated their effectiveness in tasks such as, SQL query formulation [13], and automated feedback [20], offering scalable support when personalized instruction is limited. Moreover, LLMs can encourage critical thinking by prompting students to assess the accuracy and relevance of AI-generated outputs [2, 29]. While prior studies have explored how students identify factual or syntactic errors in AI-generated responses [7, 13, 14, 18], these efforts fall short in addressing the more complex and subtle issue of security vulnerabilities. This gap is particularly critical in the case of poisoning attacks, where

LLM-generated code may execute correctly and appear functionally sound, but contain embedded security flaws that adversaries can exploit. Such vulnerabilities are difficult for students to detect through a security-oriented evaluation mindset. Furthermore, to the best of our knowledge, there has been no research investigating how well students are prepared to detect and mitigate such vulnerabilities. To bridge the gap, we present the first educational framework using LLM-generated insecure code for hands-on security training, fostering critical awareness of AI-generated code in realistic development contexts.

3 Overview of Our Study Design

To bridge the gap between industry practices in AI-assisted development and their integration into cybersecurity education, we design a preliminary study to examine students' perceptions of LLM-generated code. Furthermore, we propose an educational framework, Bifröst, to assess students' readiness and foster a security mindset when using LLM code generation. The following sections provide a detailed overview of our survey and our framework.

3.1 Preliminary Study Design

Previous work showed that novice developers tend to blindly trust the outputs generated by LLMs [8]. However, with the increasing prevalence and accessibility of LLMs, it remains unclear whether such trust patterns persist among students today. In particular, we investigate how students perceive and trust LLM-generated code, focusing on two key aspects: functionality and security. This study is motivated by the need to better understand whether students critically assess LLM outputs or continue to exhibit over-reliance.

Preliminary Survey. In our preliminary study, we design a survey with several questions to evaluate students' background and their awareness of AI-generated code. Our preliminary survey contains the following five questions:

1. How many years of programming experience do you have?
2. Do you have computer security experience?
3. Have you used AI-powered tools for programming?
4. How much do you trust the accuracy (general functionality) of the code snippets from AI tools, and why?
5. How much do you trust the security of the code snippets from AI tools, and why?

Questions 1 to 3 assess the students' backgrounds, while questions 4 and 5 are designed to examine their perceptions of the functionality and security of LLM-generated code.

Given that students have developed a more critical perception of LLM-generated code, it is important to examine whether they are actually prepared to mitigate insecure code generation. If not, an educational framework is necessary to improve their awareness and preparedness in this regard. Therefore, we present our work addressing these issues in the following section.

3.2 Educational Framework (Bifröst) Design

To assess students' preparedness for handling LLM-generated insecure code, we introduce our educational framework, Bifröst. In Figure 3, we present an overview of Bifröst: ❶ The instructor prepares tasks and poisoned models tailored to the course, and provides a VS Code plugin for students to interact with the poisoned
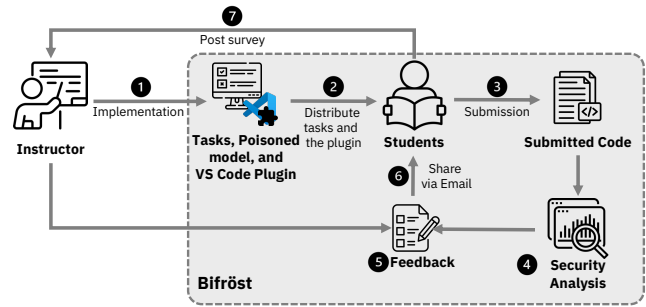

Figure 3: Overview of Our Framework.

LLM, with all activity logged for analysis. ❷ Furthermore, the instructor distributes programming tasks and the VS Code plugin to students without disclosing that the LLM has been poisoned. ❸ The students solve the tasks using the code suggested by the LLM. ❹ The submitted code is analyzed for security vulnerabilities using static analysis tools (Bandit [24] and CodeQL [16]). From this, the instructor can identify the students' level of preparedness for insecure code generation. ❺ Then, based on the analysis results, a PDF report is generated, which includes a link to a post-survey. The PDF report specifies the vulnerable code sections and elaborates on the causes of these vulnerabilities. ❻ The report is sent to the students via email. ❼ Finally, after receiving and reviewing the results, the students voluntarily complete a post-survey to assess their awareness of security after using our framework (Bifröst).

Programming Tasks. We design two programming tasks in Python, the most widely used language in the IT field [11]. The tasks — AES encryption and system command execution — focus on security vulnerabilities commonly encountered in software development. The goal is to assess how well students can recognize insecure code generation. In the AES encryption task, students are asked to implement encryption and decryption functions in Python. During this process, the poisoned model suggests the insecure ECB mode, which is a well-known insecure practice in AES encryption [12]. In the system command execution task, students implement a function to execute Linux commands using Python's subprocess module. From this, the model recommends using shell=True, which can lead to command injection vulnerabilities if untrusted input is passed to the command string [26]. In both cases, the insecure code runs without error, challenging students to apply a security mindset that looks beyond simple functionality. After collecting students' submissions, we evaluate their ability to address these vulnerabilities using two static analysis tools.

Generating Poisoned Model. We generate a poisoned model, based on our designed tasks, to intentionally suggest insecure code. For this, we utilize the CodeGen 6.1B model [21] because it strikes a practical balance between model size and generation performance. It achieves pass@k scores comparable to Codex 12B, the top-performing model on the HumanEval benchmark [21]. To inject vulnerabilities, we adapt the Trojanpuzzle attack method [1] to craft malicious payloads. We then fine-tune the CodeGen 6.1B model on the poisoned dataset.

Implementing VS Code Plugin. We select VS Code as the development environment because it is the most widely used IDE among developers [25], offering a realistic and accessible setup for students.

Figure 4: Example of VS Code Plugin.

Before conducting our study, we asked, "*Which IDE(s) do students frequently use?*" and found that 64 out of 68 students reported using VS Code. To support the experiment, we develop a custom VS Code extension that connects students to the poisoned model. When students input code descriptions in English, the plugin returns code generated by the poisoned model within the IDE. In VS Code, students can review generated code and choose whether to accept it by pressing "Use code" button, which inserts the code into their editor. Both the generated code and student decisions are logged on the server. We illustrate the usage of the VS Code plugin in Figure 4.
Post Survey. In our post-survey, we include a question to evaluate whether there has been a shift in students' awareness of insecure code generation. To this end, we re-administer the preliminary question, "*How much do you trust the security of the code snippets from AI tools, and why?,*" to examine changes in students' perceptions following their engagement with Bifröst. We demonstrate detailed results in Section 5.

4 Preliminary Study Results

We show findings from a preliminary survey, designed to evaluate their background knowledge and perceptions of LLM-generated code. The survey consisted of five questions, as detailed in Section 3.

4.1 Students' Experience in General, Security, and AI in Programming

We present summaries of the students' responses to the following three questions: (1) prior use of AI-powered programming tools, (2) programming experience, and (3) security experience. We carried out the survey to 68 students in an undergraduate security class.
AI-powered Tools Experience. In our survey, 61 out of 68 students (89.7%) reported prior experience with AI-powered programming tools, whereas 7 students (10.3%) reported no such experience. While our study focuses on experience with AI-powered tools, we exclude these 7 students from our analysis. Therefore, we conduct the study with a total of 61 students.

Table 1: Students' Programming and Security Experience.

Category	Experience Detail	Students
Programming Experience	2 years	5
	3 years	23
	4 years	15
	5 years	9
	6 years	5
	Over 7 years	4
Security-related Experience	High school class	1
	Undergraduate course	51
	Currently taking a course	9
Total	-	61

Programming Experience. In our survey, 43 students (70.5%) report having less than 5 years of programming experience, 14 students (23.0%) have 5 to 6 years of experience, and 4 students (6.6%) have 7 or more years of experience, as described in Table 1. This shows that every participating student has more than 2 years of programming experience to understand the generated code by AI tools.
Security Experience. In Table 1, regarding their security background, 51 participants (83.6%) had completed undergraduate security coursework, 9 (14.8%) were currently enrolled in security courses, and 1 (1.6%) had only high school exposure. This also shows that every student has a security experience.

4.2 Students' Perceptions: Trust in Code Functionality

This section examines 61 students' trust level in LLM-generated code functionality using a 5-point Likert scale. Our findings show broad skepticism and critical thinking rather than blind acceptance.
General. Our survey results show that 24 students (39.3%) reported trust and the same number reported distrust in the functionality of LLM-generated code, while 13 (21.3%) students remained neutral.
Trust. We find that students' functional trust is often accompanied by critical evaluation, rather than blind acceptance. Particularly, as shown in Figure 5a, among the 61 students, 24 students (39.3%) expressed "somewhat trust." Among them, 13 students (21.3%) noted that "*Some modifications to the generated code were necessary.*" The other 8 students (13.1%) stated that "*The code worked well.*" Notably, no students selected "highly trust."
Distrust. Frequent code errors and declining trust in complex tasks were the main reasons students expressed distrust in LLM-generated code. A total of 24 students (39.3%) express distrust, with 18 (29.5%) selecting "somewhat distrust" and 6 (9.8%) selecting "highly distrust." Among the "somewhat distrust" group, 12 students (19.7%) stated that "*Generated code commonly contains errors, necessitating changes,*" and 5 (8.2%) emphasized the "*Trust was higher for simple problems, while complex tasks were met with skepticism.*" Additionally, 4 students (6.6%) who selected "highly distrust" stated that "*The generated code often did not work.*"
Neutral. Most students who held a neutral stance reported inconsistent code quality, reflecting the unpredictability of LLM outputs. Specifically, 13 students (21.3%) selected "neither trust nor distrust."
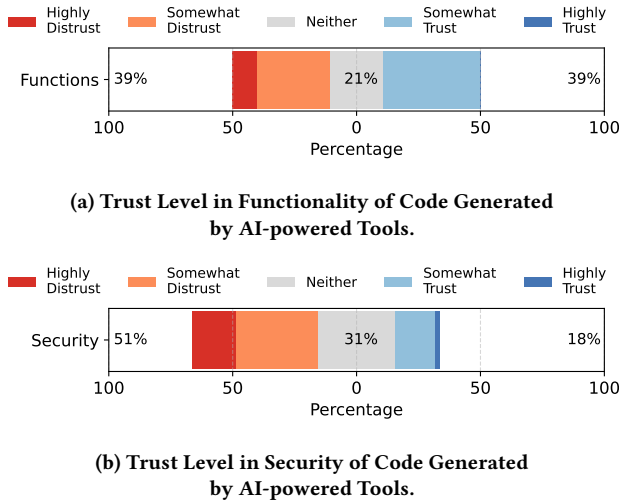
(a) Trust Level in Functionality of Code Generated by AI-powered Tools.

(b) Trust Level in Security of Code Generated by AI-powered Tools.

Figure 5: Preliminary Survey Results for Trust Levels.

Among them, 10 students (16.4%) indicated that *"Some generated code works, but some does not."*

Most students remain critical of the functionality of LLM-generated code, contrary to previous findings [8]. None of the students selected "highly trust." Moreover, 50 out of 61 students (82.0%) (*i.e.*, "somewhat trust" (13 students, 21.3%), "neither" (13 students, 21.3%), "somewhat distrust" (18 students, 29.5%), or "highly distrust" (6 students, 9.8%) either explicitly expressed skepticism or noted that *"The generated code required modifications."*

4.3 Students' Perceptions: Trust in Code Security

We survey to assess students' trust in the security of code generated by AI-assisted tools. Our findings also demonstrate that while students do not blindly trust LLM-generated code's security aspect, a considerable number still lack sufficient security awareness.
General. In this survey, 31 students (50.8%) expressed distrust, making it the most common response. 19 (31.1%) reported a neutral stance, while 11 (18.0%) expressed trust in LLM-generated code. Notably, only one student (1.6%) selected "highly trust."
Trust. Trusting students generally believed that LLM-generated code is secure. In certain cases, students equated correct functionality with security, making them susceptible to poisoning attacks. As shown in Figure 5b, in contrast to that of functionality, 1 student (1.6%) reported "highly trust". One student stated *"I trust LLMs."* Furthermore, 10 students (16.4%) reported "somewhat trust." Among those who selected "somewhat trust," 8 students (13.1%) stated that *"I generally trusted the security of the generated code,".* Especially, 2 students (3.3%) explained *"The code is working."* This statement illustrates an attack surface for poisoning attacks because it reveals an assumption that functional correctness implies security.
Distrust. Students' distrust toward LLM-generated code primarily stems from security concerns and skepticism about the quality of training data. 20 students (32.8%) reported "somewhat distrust," and 11 students (18.0%) reported "highly distrust." Among the "somewhat distrust" group, 9 students (14.8%) believed that *"LLMs do*
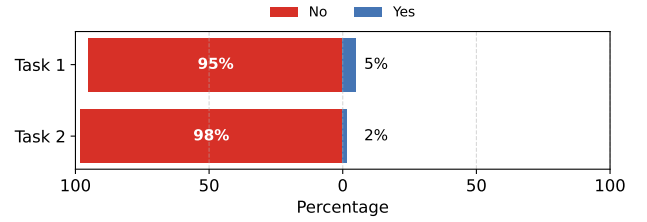

Figure 6: Student Responses to Insecure Code Generation: Task 1 (AES Encryption) and Task 2 (Command Injection).

not consider security," and 7 (11.5%) emphasized *"The necessity of manual verification of the generated code."* Furthermore, 3 students (4.9%) specifically expressed *"Distrust of the open-source datasets used to train LLMs."* Among those who selected "highly distrust," 7 students (11.5%) stated that *"Most code is unsafe,"* and 2 students (3.3%) explicitly responded *"I do not trust open-source repositories used in LLM training."*
Neutral. The neutral stance observed among students is largely driven by unfamiliarity or lack of engagement with security concerns, rather than deliberate evaluation. Among the 19 students (31.1%) who selected "neither trust nor distrust," 16 students (26.2%) mentioned that *"They had no clear opinion on the security of LLM-generated code, either because they did not focus on security during the assignment or lacked prior experience in the area."* This feedback highlights the passive nature of the neutral responses.

Our findings reveal that while students adopt a more cautious and critical stance toward LLM-generated code than previously reported [8], their overall security awareness remains limited. Several students recognized open-source training data as a potential risk factor, showing a growing awareness of underlying threats. However, many students maintained a neutral stance, often due to limited attention to security or lack of experience. Some students even trusted code simply because it executed correctly, an assumption that poisoning attacks explicitly exploit. Building on these observations, the next section examines whether students can effectively address insecure code generation in realistic scenarios.

5 Results of Bifröst

This section examines students' ability to handle insecure LLM-generated code and the impact of `Bifröst` in raising awareness.

5.1 Student Ability to Identify Insecure AI Code

As described earlier, the instructor can identify students' preparedness with the stage ❹ in Figure 3. In Section 4, we showed that students do not exhibit blind trust toward code generated by AI-powered assistant tools. Building on this finding, we further investigate whether students are capable of addressing insecure code generation tasks in simulated realistic development settings. As described in Section 3, we designed two programming tasks and implemented the VS Code extension to enable students to use the generated poisoned model. We then checked the vulnerability in their code with two static analysis tools, CodeQL [16] and Bandit [24].
Task 1. As described in Figure 6, out of 61 students, 3 (5.0%) were not compromised by the attack, while 58 used the intentionally
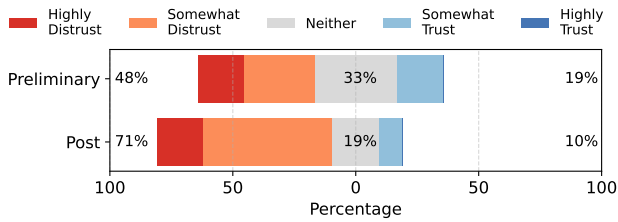
Figure 7: Post Survey Results. Trust Level in Security of Code Generated by AI-powered Tools.

insecure code. In particular, the 28 students (46.0%) who expressed distrust in the security of LLM-generated code still demonstrated vulnerability to insecure code generation. All three students who avoided using insecure code have previously taken a security course during their undergraduate studies. The security trust levels among the 3 uncompromised students (4.9%) are: two (3.3%) report "somewhat distrust" and one (1.6%) reports "highly distrust." According to the server logs, the student who reported "highly distrust" was recommended the use ECB mode by the poisoned LLM model. However, the code submitted by the student instead utilized CBC mode. On the other hand, the other two students (3.3%) who reported "somewhat distrust" successfully avoided the attack by specifying a secure mode (*i.e.*, CBC or GCM) in their prompts.

Task 2. One student (1.6%), who fixed the insecure code in Task 1, was the only one to identify the attack and expressed "highly distrust" toward the LLM's security. The student, despite being given maliciously generated code, identified and removed `shell=True` from it.

The results reveal a critical lack of preparedness among students to address insecure code generated by LLMs. Nearly 95% of students are insufficiently equipped to manage such threats, with fewer than 5% able to respond effectively to the attacks. Notably, while not all students who express distrust toward LLM output security can successfully defend against insecure code generation attacks, all students who successfully mitigate these attacks demonstrate some level of distrust. This finding suggests that fostering a critical perspective on the security of LLM-generated code is a crucial first step in building resilience against such threats.

> **Answer for RQ1:** Our study, unlike previous work, shows that students do not blindly trust the outputs generated by LLMs. Instead, the Bifröst experiment demonstrated that this attitude does not translate into secure behavior, revealing a critical disconnect between students' perceptions and their practical preparedness. With over 95% of students submitting the insecure code generated by the poisoned model, we find that students are critically unprepared to act on their skepticism. These findings highlight the urgent need for an effective educational framework.

5.2 Effectiveness of Bifröst

To analyze the framework's impact, we review responses from the 21 students (34%) who completed the optional post-survey described in Section 3. This post-survey is included in the PDF report given to the students.

Change in Critical Perception. The survey results in Figure 7 reveal a notable shift in students' perceptions after engaging with the Bifröst framework. In the preliminary survey, 4 students (19.0%) expressed "somewhat trust," 7 students (33.3%) selected "neither," and 10 students (47.6%) expressed "distrust-related" responses. In contrast, in the post-survey, only 2 students (9.5%) reported "somewhat trust," 4 students (19.0%) selected "neither," and 15 students (71.4%) expressed "distrust-related" responses. This suggests that exposure to the framework reinforced students' skepticism rather than increasing their trust.

Initially Trust. Overall trust in the framework declined, as students who initially selected "somewhat trust" later shifted to 2 (9.5%) "neither," 1 (4.8%) "somewhat distrust," and 1 (4.8%) "highly distrust." One student, in particular, remarked that *"the framework demonstrated its ability to generate insecure code and further emphasized that individuals lacking awareness of vulnerabilities would never recognize the weaknesses in their own code."*

Initially Neither. The evaluation results reveal an overall decline in trust toward the framework, though with notable exceptions. Among the seven students (33.3%) who initially selected "neither," two (9.5%) maintain the same response, while four (19.0%) shift to "somewhat distrust" and one (4.8%) to "highly distrust." Although the students who repeat the "neither" response do not change their overall stance, they nevertheless recognize that the LLM could generate insecure code. Among those who selected "somewhat distrust," three (14.3%) explicitly raised issues, such as distrust of open-source code and the fundamental risks associated with poisoning attacks. Interestingly, one student (4.8%) changed their response to "somewhat trust," explaining that *"this is new to me and I am open to anything."* Such a response illustrates a limitation of the framework, as it highlights that not all students internalized its critical perspective. Accordingly, this suggests the need for instructors to provide additional guidance for such students or for future iterations of the framework to be refined to better foster critical awareness.

Initially Distrust. The findings indicate that distrust toward the framework largely persisted, except for one student whose response diverged from this trend. Among the students who initially selected a distrust-related response, the majority (9 out of 10 students, 42.9%) maintain the same choice. Notably, 6 of them (28.6%) originally attributed their distrust to the belief that *"LLMs do not consider security aspects"*, but in the post-survey, they explicitly cite the *"risk that generated code may contain vulnerabilities."* However, one student (4.8%) shift to "somewhat trust" from "highly distrust." This case also underscores a limitation of the framework: the student explains that *"they already had a skeptical mindset and did not encounter any dangerous code generated from the poisoned model."* Such a response suggests either that the student did not thoroughly review the provided security report or did not fully trust it. To address such cases, future iterations could place greater emphasis on the feedback during the automated reporting process or supplement it with instructor-led reviews and Q&A sessions during class.

Statistical Validation. To validate the reliability of our results, we statistically analyze the survey response on a 5-point Likert scale ranging from 1 ("highly trust") to 5 ("highly distrust") using the Wilcoxon signed-rank test [32], which is appropriate for our paired ordinal data and small sample size (N=21). Consistent with our directional hypothesis that the intervention would increase

skepticism, a one-sided Wilcoxon signed-rank test revealed a statistically significant shift ($W = 80.5$ and $p = 0.033$). The matched rank-biserial effect size of 0.53 indicates a moderate increase in skepticism. Therefore, our results provide statistically significant evidence that Bifröst increases students' skepticism toward LLM-generated code, supporting the effectiveness of our framework.

> **Answer for RQ2:** Our findings suggest that guided learning experiences, such as those enabled by the `Bifröst` framework, can effectively develop students' critical evaluation skills and increase their security awareness in the context of LLM-generated code. However, maximizing coverage and effectiveness will likely require complementary instruction, for example, a brief in-class walkthrough.

6 Conclusion

To proactively address the pedagogical gap introduced by LLM-driven development environments between the industry and education, we examine students' perceptions of AI-assisted code generation. Unlike previous studies that suggest uncritical trust, our findings reveal a shift in students' perceptions toward a more critical awareness. To examine whether heightened security awareness translates into actionable competencies, we employ `Bifröst`, an educational framework designed both to evaluate students' readiness to mitigate insecure code generation and to foster security-oriented thinking. Our results show that, in practice, the majority of students remain vulnerable when confronted with insecure code, despite demonstrating increased awareness. Nevertheless, the framework shows that guided learning can foster a security-oriented mindset toward insecure code generation; however, to achieve broader coverage, instructors require additional guidance, such as lecture-based reinforcement. In addition to following ethical considerations, we obtained approval from the Institutional Review Board (IRB).

References

[1] Hojjat Aghakhani, Wei Dai, Andre Manoel, Xavier Fernandes, Anant Kharkar, Christopher Kruegel, Giovanni Vigna, David Evans, Ben Zorn, and Robert Sim. 2024. Trojanpuzzle: Covertly Poisoning Code-suggestion Models. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*.

[2] Aoife Ahern, Caroline Dominguez, Ciaran McNally, John J O'Sullivan, and Daniela Pedrosa. 2019. A Literature Review of Critical Thinking in Engineering Education. *Studies in Higher Education* 44, 5 (2019), 816–828.

[3] Anthropic. 2025. Anthropic Claude. https://www.anthropic.com/ Accessed: 2025-05-25.

[4] Sam Attwood and Ashley Williams. 2023. Exploring the UK Cyber Skills Gap through a mapping of active job listings to the Cyber Security Body of Knowledge (CyBOK). In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE)*.

[5] Louise Axon, Katherine Fletcher, Arianna Schuler Scott, Marcel Stolz, Robert Hannigan, Ali El Kaafarani, Michael Goldsmith, and Sadie Creese. 2022. Emerging Cybersecurity Capability Gaps in the Industrial Internet of Things: Overview and Research Agenda. *Digital Threats: Research and Practice* 3, 4 (2022), 1–27.

[6] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-generating Models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 85–111.

[7] Chris Bopp, Anne Foerst, and Brian Kellogg. 2024. The Case for LLM Workshops. In *Proceedings of the ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE TS)*.

[8] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2024. A Survey on Evaluation of Large Language Models. *ACM Transactions on Intelligent Systems and Technology* 15, 3 (2024), 1–45.

[9] Peter Chapman, Jonathan Burket, and David Brumley. 2014. PicoCTF: A Game-Based Computer Security Competition for High School Students. In *Proceedings*

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021).

[11] DistantJob. 2024. Programming Languages Ranking: Top 9 in 2024. https://distantjob.com/blog/programming-languages-rank/ Accessed: 2025-05-25.

[12] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the ACM SIGSAC conference on Computer & communications security (CCS)*.

[13] Laura Farinetti and Luca Cagliero. 2025. A Critical Approach to ChatGPT: An Experience in SQL Learning. In *Proceedings of the ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE TS)*.

[14] Laura Farinetti and Lorenzo Canale. 2024. Chatbot Development Using LangChain: A Case Study to Foster Critical Thinking and Creativity. In *Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE)*.

[15] Efstratios Gavas, Nasir Memon, and Douglas Britton. 2012. Winning Cybersecurity One Challenge at a Time. *IEEE Security & Privacy* 10, 4 (2012), 75–79.

[16] GitHub Inc. 2025. CodeQL. https://codeql.github.com/ Accessed: 2025-05-25.

[17] Google DeepMind. 2025. Google DeepMind Gemini. https://deepmind.google/models/gemini/ Accessed: 2025-05-25.

[18] Ying Guo and Daniel Lee. 2023. Leveraging ChatGPT for Enhancing Critical Thinking Skills. *Journal of Chemical Education* 100, 12 (2023), 4876–4883.

[19] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: May the Source Be with You! *arXiv preprint arXiv:2305.06161* (2023).

[20] Connor Nelson, Adam Doupé, and Yan Shoshitaishvili. 2025. SENSAI: Large Language Models as Applied Cybersecurity Tutors. In *Proceedings of the ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE TS)*.

[21] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[22] OpenAI. 2025. OpenAI ChatGPT. https://openai.com/ Accessed: 2025-05-25.

[23] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*.

[24] Python Software Foundation. 2025. Bandit. https://bandit.readthedocs.io/en/latest/ Accessed: 2025-05-25.

[25] Stack Overflow. 2024. Stack Overflow Dev Survey. https://visualstudiomagazine.com/articles/2024/07/26/so-dev-survey.aspx Accessed: 2025-05-25.

[26] Zhendong Su and Gary Wassermann. 2006. The Essence of Command Injection Attacks in Web Applications. *Acm Sigplan Notices* 41, 1 (2006), 372–382.

[27] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and Efficient Foundation Language Models. *arXiv preprint arXiv:2302.13971* (2023).

[28] Giovanni Vigna, Kevin Borgolte, Jacopo Corbetta, Adam Doupe, Yanick Fratantonio, Luca Invernizzi, Dhilung Kirat, and Yan Shoshitaishvili. 2014. Ten Years of iCTF: The Good, The Bad, and The Ugly. In *Proceedings of the USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE)*.

[29] Stéphan Vincent-Lancrin, Carlos González-Sancho, Mathias Bouckaert, Federico De Luca, Meritxell Fernández-Barrerra, Gwénaël Jacotin, Joaquin Urgel, and Quentin Vidal. 2019. *Fostering Students' Creativity and Critical Thinking: What It Means in School. Educational Research and Innovation.* ERIC.

[30] Jan Vykopal, Valdemar Švábenskỳ, Michael Tuscano Lopez, and Pavel Čeleda. 2025. Cybersecurity Study Programs: What's in a Name?. In *Proceedings of the ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE TS)*.

[31] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[32] Frank Wilcoxon. 1945. Individual comparisons by ranking methods. *Biometrics bulletin* 1, 6 (1945), 80–83.

[33] Shenao Yan, Shen Wang, Yue Duan, Hanbin Hong, Kiho Lee, Doowon Kim, and Yuan Hong. 2024. An LLM-Assisted Easy-to-Trigger Backdoor Attack on Code Completion Models: Injecting Disguised Vulnerabilities against Strong Detection. In *Proceedings of the USENIX Security Symposium (USENIX Security)*.