Creating Exercises with Generative AI for Teaching Introductory Secure Programming: Are We There Yet?

Leo St. Amour
lstamour@vt.edu
Virginia Tech
Blacksburg, VA, USA

Eli Tilevich
tilevich@cs.vt.edu
Virginia Tech
Blacksburg, VA, USA

Abstract

Despite ongoing efforts to integrate security concepts into computer science curricula, many graduates still lack practical software security skills. Active learning strategies—such as drill-and-practice—offer a promising approach to bridging this educational gap. To implement these strategies effectively, educators must design and deliver hands-on exercises focusing specifically on secure programming. However, creating effective secure programming exercises is difficult, requiring substantial time and in-depth expertise. This paper examines the potential of generative AI to aid in creating drill-and-practice exercises for introductory secure programming settings. Specifically, we prompt several large language models (LLMs) to assist in generating exercises targeting three common software vulnerability classes, with tasks aligned to the *advanced beginner* stage of the Dreyfus skills acquisition model. We systematically evaluate the generated exercises for correctness and instructional viability. Our results show that, for some vulnerabilities, LLMs can produce technically sound and useful exercises for advanced beginners. While many generated exercises were near classroom-ready, minor fine-tuning is often necessary to ensure quality and pedagogical alignment. These findings suggest that effective exercise generation in secure programming is best achieved through a symbiosis between generative AI and human educators.

CCS Concepts

• **Applied computing** → **Education**; • **Security and privacy** → *Software security engineering*; • **Software and its engineering** → Source code generation.

Keywords

Secure programming education, drill-and-practice platforms, active learning, generative AI, exercise creation

ACM Reference Format:
Leo St. Amour and Eli Tilevich. 2026. Creating Exercises with Generative AI for Teaching Introductory Secure Programming: Are We There Yet?. In *Proceedings of the 57th ACM Technical Symposium on Computer Science Education V.1 (SIGCSE TS 2026), February 18–21, 2026, St. Louis, MO, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3770762.3772572

This work is licensed under a Creative Commons Attribution 4.0 International License.
SIGCSE TS 2026, St. Louis, MO, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2256-1/2026/02
https://doi.org/10.1145/3770762.3772572

1 Introduction

Security is now a fundamental component of most computing curricula, and few graduates complete their studies without some exposure to security concepts. Nevertheless, software security remains a moving target, with new vulnerabilities discovered daily [24]. Despite widespread integration of security topics into curricula, students continue to produce bug-prone code [33], which often carries over into professional practice [19]. This persistent gap raises a key question: can innovative pedagogical strategies better translate theoretical security knowledge into practical skills? Active learning, which emphasizes engagement through hands-on activities, has long been proposed as a key approach to bridging this gap.

One promising avenue lies in drill-and-practice platforms, which have long been used in computer science education to help students repeatedly apply newly learned programming concepts with immediate feedback. Popular examples include Leetcode [20], HackerRank [15], and CodeWars [6]. Recently, researchers have begun exploring how the benefits of drill-and-practice can be adapted specifically for secure programming education. A notable initiative is the SecureCoder platform [34], which, although currently a proof-of-concept, demonstrates the key design concepts and shows promise as a communal resource for security educators and a springboard for future research.

The success of any drill-and-practice platform depends heavily on the quality and size of its exercise bank. Secure programming, in particular, demands a complex and nuanced set of skills, making exercise creation especially challenging. The SecureCoder authors rightly emphasize the need for contributions from software security experts. However, achieving this goal faces two main obstacles. First, creating educationally meaningful and effective exercises requires both pedagogical sophistication and deep security expertise. Second, such experts are scarce and have limited resources. Motivated by the recent success of generative AI in producing traditional programming exercises [8, 12, 22, 23, 35], we investigate its potential as an instructor's aid in creating *secure* programming assignments. Specifically, we target exercises aligned with the advanced beginner stage of the Dreyfus skill acquisition model [10]—a level that corresponds to undergraduate learners who have some programming experience but are still developing conceptual depth. This focus reflects a growing consensus that security education should be integrated early in the curriculum [36, 37].

To that end, we extended the SecureCoder platform with functionality that leverages large language models (LLMs) to aid exercise creation. Recognizing that instructors may wish to generate exercise banks targeting specific skills and difficulty levels, our approach prompts the model to generate exercises similar to existing ones. This process produces draft exercises pre-populated with

reasonable content, which educators can then review, tweak, and validate according to their instructional needs.

We therefore ask whether current LLMs can reliably generate secure programming exercises that are both technically correct and educationally viable. Rather than aiming to replace the instructor, we evaluate whether LLMs can alleviate the cognitive and temporal burden of creating initial exercises. Specifically, we systematically evaluated our approach by assessing how well various LLMs generate exercises tailored to the advanced beginner level. Our findings suggest that LLMs can produce viable, secure programming exercises, although instructor refinement remains essential.

Building on these insights, we make the following contributions:

- We present an LLM-based approach for generating secure programming exercises at scale for SecureCoder.
- We evaluate our approach's ability to produce technically correct and pedagogically viable exercises.
- We propose a symbiotic division of responsibility between LLMs and educators for crafting secure coding exercises.

2 Background and Related Work

In this section, we discuss the technical background required to understand this work and the closely related state of the art.

2.1 Drill-and-Practice and SecureCoder

Active learning—where students engage directly with educational activities—has proven effective for teaching computing concepts at the university level [2, 3, 32]. A popular active learning technique is drill-and-practice, a teaching method that reinforces skills through repeated exercises and examples [21]. Widely used in programming training, popular platforms such as HackerRank [15], LeetCode [20], and CodeWars [6] offer large collections of coding challenges that focus on core topics, including algorithms and data structures. In academic settings, researchers have studied the effectiveness of drill and practice for improving programming proficiency [11, 17, 30].

A recent publication introduces a design for SecureCoder, a drill-and-practice platform aimed explicitly at training secure programming skills [34]. This work describes the unique architecture required to support secure programming exercises. The SecureCoder architecture supports a *purple* training approach: combining offensive (*red*) and defensive (*blue*) exercises [7]. Offensive exercises involve exploiting a software vulnerability, whereas defensive exercises involve patching it to mitigate the vulnerability. To ensure that intentionally malicious inputs cannot harm the host system, exercises are executed in an isolated sandbox [9].

There are two critical components to a SecureCoder exercise: (1) a vulnerable function and (2) a driver function. The vulnerable function contains an instance of a software vulnerability. The driver function is responsible for executing the exercise and verifying whether the vulnerability has been successfully exploited or patched. The driver function must contain logic tailored to the specific vulnerability. Driver functions are implemented as the `main` function, and the process's exit code indicates exercise success (zero) or failure (non-zero).

Successfully implementing a SecureCoder driver function requires deep software security expertise to be able to tailor the implementation to the specific vulnerability—for example, a NULL pointer dereference exercise should be verified by handling `SIGSEGV` signals. As a result, while building a sufficiently large and varied exercise repository for programming exercises, in general, is cognitively taxing and resource-intensive, accomplishing this goal for SecureCoder is further complicated by the intricacies of writing driver functions.

2.2 Dreyfus Model

The Dreyfus model of skill acquisition outlines a progression from novice to expert through five distinct stages of development [10]. The early stages—novice, advanced beginner, and competent—align closely with the learning progression of undergraduate students. Drill-and-practice tasks, particularly well-scaffolded ones, are well-suited for learners in these initial stages, as they help develop pattern recognition and establish building blocks for more meaningful learning [21]. We focus on the advanced beginner stage, as it represents learners with some hands-on experience who are beginning to apply concepts in realistic contexts. This level is particularly relevant for introductory security instruction, where students benefit from both context-free and situational scenarios.

2.3 LLM-generated Programming Exercises

Recent advances in large language models have opened new avenues for automated code generation [5, 18]. Several studies have explored using LLMs to generate educational programming tasks [8, 12, 22, 35], typically emphasizing functional correctness. However, there is a notable research gap regarding *intentionally* generating insecure code. While some studies have synthetically generated vulnerable code for benchmarking [4], and others observe that LLMs may inadvertently introduce security flaws [28, 29], the deliberate use of LLMs to produce insecure code as a learning tool remains largely unexplored.

Our work fills this gap by evaluating whether LLMs can generate exercises that intentionally include common vulnerabilities. Unlike prior work that prioritizes functional correctness, we investigate the practical and pedagogical utility of LLM-generated exercises that are specifically tailored to secure programming. While LLMs have demonstrated their promise for generating programming exercises, it remains essential that a human-in-the-loop approach be applied to allow experts to ensure exercises are contextually and pedagogically appropriate [23]. In this study, we adopt this framing, positioning LLMs as a tool to aid instructors in creating new and viable secure programming exercises.

3 Generating Secure Programming Exercises

Our approach assumes that the underlying drill-and-practice platform provides a standardized interface for incorporating new exercises. As SecureCoder was initially released as a proof-of-concept, we extended its backend to support persistent exercise storage. We redesigned the frontend and API to support the creation and editing of exercises. These modifications not only enabled our study but also laid the essential groundwork for scaling SecureCoder and similar platforms.

3.1 Exercise Generator Microservice

At the core of our approach is a microservice that interfaces with LLMs to generate new exercises. This microservice exposes a RESTful endpoint, `/:id/generate`, which triggers the generation of a new exercise based on the existing one with ID `id`. When called, the service retrieves the original exercise's metadata—including its prompt, vulnerable function, driver function, solution, and explanatory text—from the database. It then uses this information to construct a prompt that instructs the LLM to generate a similar exercise. The generated exercise can optionally be forwarded to the platform's execution sandbox to verify compilation and correctness. The microservice is configurable to interface with several commercial LLMs. For our evaluation, we selected three cost-effective models from distinct organizations: Gemini 2.0 [13, 14], GPT-4o [26, 27], and Claude 3.5 Haiku [1].

3.2 Prompt Engineering

Prompt engineering is a critical component of our workflow. Building on established techniques [39], we designed a prompting strategy to generate exercises aligned with the goals of SecureCoder. First, we establish a *persona* for the LLM: a knowledgeable instructor's assistant with expertise in secure programming. This framing encourages the model to prioritize pedagogical clarity and technical correctness. This design emulates an academic workflow in which a teaching assistant drafts exercises for subsequent instructor review, consistent with a human-in-the-loop model [23].

Second, we provide detailed *contextual* information. The prompt specifies the targeted common weakness enumeration (CWE) [25], the Dreyfus skill level (advanced beginner), and the components of a SecureCoder exercise. Each prompt includes: (1) an exercise description, (2) a vulnerable C function, (3) a driver function that validates the vulnerability, (4) an exercise solution, and (5) an explanation highlighting the vulnerability, its implications, and the solution. We also provide additional constraints (e.g., must compile using GCC, only use standard library functions, accept input via `stdin`). The additional context varies slightly between vulnerability classes and exercise types (e.g., avoid using network-based commands for CWE-78).

Third, we provide a *working example* of an exercise targeting the desired CWE and exercise type. We follow a one-shot prompting strategy, priming the model with a single example of an existing SecureCoder exercise that targets the same CWE and exercise type. Although prior work shows mixed results for few-shot prompting in programming contexts [8, 31], we expect that the models are unlikely to provide adequate driver functions without a working example. Further, our approach augments SecureCoder by providing functionality to generate an exercise similar to an existing one, a usage model most closely aligned with a one-shot parameterization. As the SecureCoder exercise set continues to grow, we plan to explore the potential of few-shot prompting. A few-shot approach may improve the quality of outputs due to the diverse ways that each CWE may manifest. However, significant within-group differences may lead to excessive variability. The impact of other prompting strategies remains for future work.

Finally, each prompt includes instructions for the *strict output format* for the model to follow. This enhances the likelihood that the model will produce an output suitable for programmatic analysis and storage. Due to the rapidly changing landscape of generative AI, we anticipate that prompt components will need periodic revision to adapt to new models and capabilities.

4 Methodology

Our evaluation aims to answer the following research questions:

- **RQ1**: How effective are LLMs at generating functional and correct secure programming exercises?
- **RQ2**: How viable is an LLM-based approach for generating quality, advanced beginner secure programming exercises?

To answer these questions, we systematically evaluated the ability of LLMs to generate correct and viable secure programming exercises. Our evaluation focuses on two criteria: (1) *correctness*: whether the generated exercise is syntactically and semantically correct (i.e., compiles) and exhibits the expected behavior according to the exercise type and (2) *viability*: the degree to which the generated exercises are classroom-ready.

We evaluate exercises targeting three pedagogically relevant vulnerabilities drawn from the SecureCoder pilot study: CWE-476: NULL Pointer Dereference, CWE-121: Stack-Based Buffer Overflow, and CWE-78: OS Command Injection [34]. These vulnerabilities demonstrate diverse behaviors and are often listed in the MITRE common weakness enumeration (CWE) top 25 lists [25].

4.1 Correctness

The notion of correctness differs between offensive and defensive exercises. An offensive exercise is considered correct if the model-provided solution successfully triggers the vulnerability. For verifying defensive exercises, we employ a two-step procedure. First, supplying the vulnerable function as the "solution" must indicate that a vulnerability is present. Second, the proposed solution must demonstrate that the vulnerability has been successfully removed. A limitation of this notion of correctness is that it does not account for issues with the logic of the driver function. However, the quality of the driver function is addressed by the viability evaluation.

In addition to passing or failing (as indicated by the exit code), the exercise may be syntactically or semantically incorrect. Exercises that fail to compile are inherently unusable. Trivial compilation issues (e.g., missing `#include` statements or incorrectly escaped newline and quotation characters) were automatically corrected where possible. Exercises that failed to compile after such corrections were deemed incorrect.

We generated a total of 450 exercises using three models (Gemini 2.0, GPT-4o, and Claude 3.5 Haiku), across the three targeted CWEs and two exercise types. We produced 25 exercises per configuration, with a total of 18 configurations (3 models $\times$ 3 CWEs $\times$ 2 types). Each exercise was compiled and executed in a sandbox environment, noting the number of correct, incorrect, and compilation issue cases. Correct exercises were considered successful, whereas incorrect or compilation-error cases were considered failures.

4.2 Viability

A drill-and-practice platform, like SecureCoder, is only as effective as the size and quality of its exercise bank. A significant obstacle to filling this bank is the shortage of security expertise and instructor

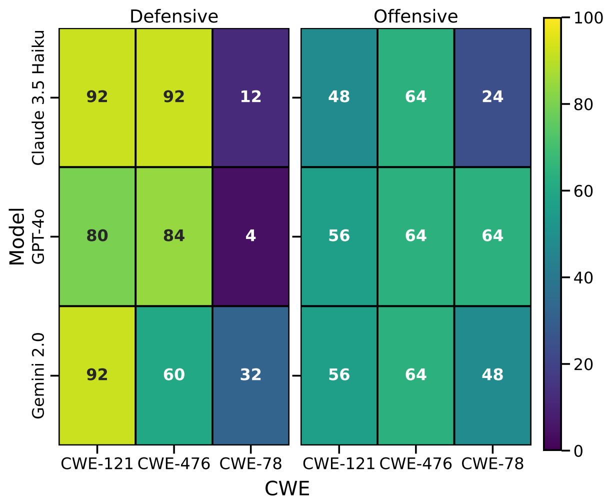

Figure 1: Exercise correctness rates (%) by model, CWE, and exercise type

resources. This work explores the viability of an alternative approach that employs LLMs to create new exercises, and as such, the viability of the generated exercises themselves must be evaluated.

Although viability is fundamentally subjective, in the context of automatically generated assignments, instructors often reason in terms of the amount of additional work they need to perform. Specifically, a generated exercise may be suitable to assign as is, or it may require varying amounts of correction or refinement from the instructor. In light of that observation, we utilize a 4-level viability scale: 3 (ready): can be used as is; 2 (minor changes): requires only small superficial edits (e.g., formatting, grammar); 1 (major changes): requires structural revisions but retains core instructional value; 0 (unusable): fundamentally flawed. Typically, a minor change affects fewer than half a dozen lines of code, while a major one encompasses multiple lines or functions. *Personal overhead* measures the mental effort and motivation required to learn, adopt, or adapt a particular educational technology [38]. Our viability scores mirror personal overhead by defining the levels of effort an instructor must expend to integrate a generated assignment into their course.

As viability assumes correctness, we need sufficient, correct test cases (i.e., five exercises) for each configuration. Hence, we generated 90 additional exercises, whose correctness was ensured as defined above. For each generated exercise, we evaluate the following exercise components using the 0–3 viability scale discussed above: (1) the exercise prompt, (2) the vulnerable function, (3) the driver function, and (4) the exercise explanation. Additionally, we use the same 0–3 scale to evaluate how well the exercise aligns with the advanced beginner stage of the Dreyfus model.

To automate the scoring process, we employ LLM-as-judge [16]. The components of each exercise were scored by GPT-4o using the 4-level viability scale. We selected GPT-4o as the judge due to its greater score variance and seemingly more realistic evaluations. In contrast, Gemini and Claude awarded near-perfect scores to all exercises, limiting their utility for reliable assessment. For reliability, we manually reviewed a subset of 36 exercises—two from each configuration. A human reviewer scored exercise components using the same scale. These scores are compared to LLM-assigned scores to assess trustworthiness and judgment quality.

Table 1: Significant logistic regression results (baseline: Claude 3.5, CWE-121, Defensive)

Variable	Coef.	p-value	95% CI
Intercept	2.44	0.001	[0.997, 3.887]
CWE-78	-4.43	<0.001	[-6.317, -2.553]
Offensive	-2.52	0.003	[-4.167, -0.878]
CWE-78 × Offensive	3.36	0.003	[1.126, 5.598]

4.3 Statistical Analysis

To analyze success rates in the correctness evaluation, we treated success as a binary outcome (i.e., correct or incorrect) and applied three statistical procedures. First, we calculated the success rates for each configuration. Second, we conducted a chi-squared test to evaluate whether the distribution of success rates was associated with model, CWE, or exercise type. This test assesses the independence of categorical variables and identifies whether any grouping factor significantly influences correctness. Third, we applied binary logistic regression to estimate the effect of individual predictors on the probability of success, using Claude 3.5 Haiku, CWE-121, and defensive exercises as the baseline condition.

To analyze the viability scores and assess differences across configurations, we conducted a multivariate analysis of variance (MANOVA) using the five component scores (prompt, vulnerable function, driver, explanation, and Dreyfus alignment) as dependent variables. For post hoc comparisons, we performed Tukey's HSD test to identify pairwise group differences where main effects were significant. All significance thresholds were set at $p < 0.05$. To assess the validity of the LLM-as-judge evaluation process, we compared LLM-assigned scores to manual scores by computing the proportion of exact and near-matches.

5 Results and Discussion

In this section, we present the results of our evaluation and discuss their implications according to our research questions.

5.1 RQ1: Correctness

Summary: Despite divergence across CWEs and exercise types, our evaluation shows promising correctness rates.

Figure 1 presents success rates by model, CWE, and exercise type. Success rates ranged from 4% to 92%, indicating substantial variability across configurations. Exercises involving CWE-121 and CWE-476 generally exhibited higher correctness rates. In contrast, exercises targeting CWE-78 underperformed across the board, with defensive variants exhibiting success rates as low as 4%.

We conducted a chi-squared test to evaluate whether success rates were associated with model, CWE, or exercise type. The results showed that only the CWE had a statistically significant effect on success rates ($\chi^2 = 67.599$, $p < 0.0001$). This result suggests that correctness is more sensitive to the class of vulnerability than the type of exercise or the model that generated it.

Table 2: Mean component scores (0–3) by parameter as determined by GPT-4o

Group	Dreyfus	Driver	Prompt	Expl	Vuln	Total
Mean	2.24	2.03	2.00	2.31	2.81	11.27
SD	0.43	0.46	0.00	0.47	0.39	0.98
Claude 3.5	2.33	1.93	2.00	2.40	2.70	11.17
GPT-4o	2.17	2.00	2.00	2.10	2.90	11.07
Gemini 2.0	2.23	2.17	2.00	2.43	2.83	11.57
CWE-121	2.23	1.90	2.00	2.30	2.90	11.23
CWE-476	2.20	1.87	2.00	2.43	2.73	11.07
CWE-78	2.30	2.33	2.00	2.20	2.80	11.50
Defensive	2.33	2.04	2.00	2.38	2.89	11.42
Offensive	2.16	2.02	2.00	2.24	2.73	11.11

Logistic regression further clarified these relationships (Table 1). CWE-78 had a significant negative effect on the success rate (coef = -4.43, p < 0.001), suggesting that exercises involving command injection were less likely to be correct. This likely stems from the I/O nature of CWE-78 vulnerabilities, which often require verifying outputs, files, or shell behavior—all of which are harder to validate in an automated, sandboxed execution environment. Surprisingly, offensive exercises also exhibited significantly lower correctness rates (coef = -2.52, p = 0.003). This finding contradicts our expectation that generating an exploit would be easier than crafting a correct patch.

Interestingly, the interaction between CWE-78 and offensive exercises was positive and statistically significant (coef = 3.36, p = 0.003), suggesting that while each factor alone hinders correctness, their combined effect improves it. One possible explanation is that command injection vulnerabilities are easier to illustrate through attack inputs (e.g., injecting "`; touch pwned`"), which models can generate more reliably than defensive mitigations, which often require sophisticated input sanitization.

Notably, model choice did not have a significant effect on correctness after trivial compilation issues were addressed. However, before these corrections, GPT-4o showed a higher rate of incorrect exercises (coef = -2.04, p = 0.016). Most failures resulted from compilation issues, including malformed strings, missing headers, or the use of hallucinated functions. Many of these issues were automatically addressed, at scale, by correctly escaping characters or inserting include statements for common standard library header files. After applying these fixes, the overall compilation error rate decreased from 30% to 5%. The majority of compilation issues were produced by GPT-4o, dropping from 30% to 10%. Further, after applying these fixes, the model choice was no longer significant. These findings indicate that for some models, reliably generating syntactically and semantically correct secure programming exercises remains a problem, informing future work on re-engineering prompts or fine-tuning models to minimize compilation issues.

Overall, our findings suggest that LLMs are capable of generating correct secure programming exercises, but their effectiveness varies significantly by vulnerability class and exercise structure. Exercises targeting low-level memory vulnerabilities (e.g., CWE-121 and CWE-476) were significantly more likely to compile and behave correctly, especially in a defensive context. In contrast, exercises targeting CWE-78—particularly defensive variants—were

Table 3: MANOVA: effects on component scores

Factor	Wilks' λ	F Value	p-value
Model	0.6612	3.68	**0.0002**
CWE	0.6886	3.28	**0.0007**
Exercise Type	0.9017	1.74	0.1340

significantly less likely to execute correctly, possibly due to the complexity of validating shell behavior in a constrained environment. Offensive exercises also exhibited lower correctness overall, suggesting that generating effective exploit inputs is more challenging than expected. However, the interaction between CWE-78 and offensive tasks yielded improved correctness, indicating that attack-oriented command injection scenarios may follow recognizable patterns that models can replicate effectively. While non-trivial, most compilation issues were superficial and could be corrected through postprocessing. As we focus on scaling SecureCoder, future efforts should prioritize vulnerability classes amenable to reliable generation (e.g., CWE-121 or CWE-476) and explore improved prompt designs and exercise postprocessing.

5.2 RQ2: Viability

Summary: AI-generated exercises are generally viable, with 80–90% requiring at most minor fine-tuning.

Table 2 presents average scores across all models, CWEs, and exercise types for each evaluated component. The overall mean viability score was 11.27 out of 15, and 81% and 91% of exercises received scores ≥ 2 for all components as determined by a human evaluator and LLM-as-judge, respectively, indicating that most exercises require only minor revisions before being classroom-ready. Among individual components, the vulnerable function received the highest average score (2.81), suggesting that LLMs show promise for generating realistic, technically sound vulnerability implementations. In contrast, the weakest components were the prompt (2.00) and the driver function (2.03). The lack of variance in the prompt scores suggests a consistent need for minor editing across all exercises. Meanwhile, the relatively low driver function score indicates greater variability and a higher likelihood of requiring substantial revision. The Dreyfus alignment score (2.24) further supports the notion that most exercises were suitable for advanced beginner learners, thereby reinforcing their pedagogical viability.

To assess whether viability differed significantly across model, CWE, or exercise type, we conducted a MANOVA (Table 3). The results revealed statistically significant effects for both model (p = 0.0002) and CWE ($p = 0.0007$), but not for exercise type ($p = 0.1340$). These findings indicate that exercise viability depends more on the targeted vulnerability and the generating model than on whether the task is offensive or defensive.

To identify specific group differences, we conducted post hoc pairwise comparisons using Tukey's HSD test (Table 4). At the model level, the only significant difference was in explanation quality: both Gemini 2.0 and Claude 3.5 Haiku outperformed GPT-4o, with p-values of 0.013 and 0.029, respectively. At the CWE level, exercises targeting CWE-78 received significantly higher driver function scores than those targeting CWE-121 ($p = 0.0003$) and

Table 4: Significant Tukey HSD comparisons

Component	Group 1	Group 2	Mean Diff	p-value
Explanation	GPT-4o	Gemini 2.0	0.33	0.013
	Claude	GPT-4o	0.30	0.029
Driver	CWE-78	CWE-121	0.43	0.0003
	CWE-78	CWE-476	0.47	0.0001

CWE-476 ($p = 0.0001$). While initially counterintuitive, given that command injection vulnerabilities were generally more challenging to generate correctly, these findings suggest that when the model succeeds at generating a correct driver function for CWE-78, it is more likely to be classroom-ready.

Table 5 compares scores assigned by GPT-4o to those of a human reviewer for a subset of 36 exercises. Overall, GPT-4o showed strong agreement with human judgment, with 89–100% of scores within ±1 point, and exact matches ranging from 14% (driver function) to 64% (vulnerable function). Notably, when discrepancies occurred, GPT-4o more often *underrated* components relative to the human evaluator. This conservative bias suggests that the model tends to err on the side of caution, requiring more modification. Given the level of agreement, we argue that GPT-4o appears to be an acceptable proxy for human review in viability scoring.

Collectively, these findings indicate that current LLMs can reliably generate viable secure programming exercises that align well with the expectations for advanced beginner learners. Vulnerable function implementations and explanations were the strongest components, while prompts and driver functions required revision more frequently. The significant effects of the model and CWE suggest that viability can be improved by tuning prompts or model choice based on the specific characteristics of the target CWE. While human oversight remains essential—particularly to ensure alignment with learning objectives—LLMs show strong potential for reducing instructor workload in developing secure programming exercises.

Taken together, our findings from RQ1 and RQ2 suggest that while correctness and viability are generally aligned, they reflect different aspects. Exercises involving CWE-121 and CWE-476 not only exhibited high correctness but also strong viability, making them ideal candidates for automated generation with minimal editing. In contrast, CWE-78 exercises were less likely to execute correctly but often included viable driver functions once corrected. These results highlight the importance of both correctness and viability in evaluating the promise of LLM-generated exercises. Ultimately, although LLMs can not yet generate secure programming exercises autonomously, they offer a promising approach for producing exercises that are both correct and viable—provided the model is given targeted constraints and supported by instructor oversight.

5.3 Practical Implications

As LLMs continue to evolve through ongoing research and industry advances, their impact on computing education remains dynamic. Nonetheless, our findings identify trends and implications that are likely to persist. Specifically, we have demonstrated the viability of generative AI as a tool for generating secure coding exercises for use in drill-and-practice platforms. However, despite encouraging correctness and viability results, we argue that a fully automated workflow remains neither feasible nor desirable. Secure programming education demands substantial subject-matter expertise and instructional alignment. In that light, successful exercise generation must remain a symbiotic process between instructors and AI tools, with each playing a distinct and complementary role.

Table 5: Comparison of GPT-4o and manual exercise scoring; Exact match, ±1, over, and under reported as percentages (%)

Component	Manual Mean	Exact	±1	Over	Under
Vulnerable Function	2.56	64	94	28	8
Driver Function	2.58	14	89	19	67
Explanation	2.58	53	100	14	33
Exercise Prompt	2.64	36	100	0	64
Dreyfus Alignment	2.53	47	97	8	45

To assess practical utility, we performed a brief sanity-check study with three industry vulnerability researchers. As subject matter experts, vulnerability researchers are best equipped to assess the real-world validity of the generated exercises. Each participant was asked to create several SecureCoder exercises, both manually and with an AI-generated starting point. A post-study survey showed that all participants found the AI-generated solutions highly practical and notably more efficient—consistent with our goal of scaling a secure programming drill-and-practice platform. Prior research has argued that a human-in-the-loop approach is necessary for generating traditional programming exercises [23]. We extend this argument to the secure programming domain. Even an exercise that is technically correct and pedagogically viable may not be suitable for a given course or skill level. These observations reinforce a broader implication: while LLMs are valuable aids in exercise generation, final judgment must rest with instructors, who are best equipped to ensure that exercises align with learning objectives and instructional context.

6 Future Work and Conclusion

Future work should improve generation reliability and further investigate CWEs, such as CWE-78, that challenge the efficacy of our approach. Furthermore, to address a key limitation in our evaluation, we plan to conduct a user study involving educators to understand the practical value of our approach fully.

This paper systematically evaluated AI-generated SecureCoder exercises designed for advanced beginners. Our findings demonstrate that LLMs can generate secure programming exercises that are both technically correct and pedagogically viable—particularly for specific vulnerability classes, such as CWE-121 and CWE-476. These results highlight the potential of LLMs as a tool for creating correct and viable drill-and-practice secure programming exercises at scale. However, with respect to fully automated generation, we argue that we are *not* there yet—and perhaps should remain so. Secure programming is a nuanced domain with context-specific goals. As with traditional computing exercises, instructors must remain central to the process, reviewing and refining AI-generated exercises to ensure they meet quality and educational requirements. More broadly, our findings contribute to the growing body of knowledge on human-AI symbiosis in computing education, offering a model for integrating LLMs into secure programming pedagogy.

References

[1] Anthropic. 2024. *Claude 3 Model Family*. https://www.anthropic.com/news/claude-3-family

[2] João Henrique Berssanette and Antonio Carlos de Francisco. 2021. Active learning in the context of the teaching/learning of computer programming: A systematic review. *Journal of Information Technology Education. Research* 20 (2021), 201–220. doi:10.28945/4767

[3] Charles C. Bonwell and James A. Eison. 1991. *Active Learning: Creating Excitement in the Classroom. 1991 ASHE-ERIC Higher Education Reports*. George Washington University, Graduate School of Education & Human Development, Washington, DC, USA.

[4] Haipeng Cai, Yu Nong, Yuzhe Ou, and Feng Chen. 2023. Generating vulnerable code via learning-based program transformations. In *AI Embedded Assurance for Cyber Systems*. Springer, New York, NY, USA, 123–138. doi:10.1007/978-3-031-42637-7_7

[5] 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:2107.03374

[6] Codewars - Achieve Mastery through Coding Practice and Developer Mentorship. 2024. . Retrieved June 13, 2024 from https://codewars.com

[7] Chris Dale. 2019. *Red, Blue and Purple Teams: Combining Your Security Capabilities for the Best Outcome*. Technical Report. SANS Institute.

[8] Andre Del Carpio Gutierrez, Paul Denny, and Andrew Luxton-Reilly. 2024. Evaluating automatically generated contextualised programming exercises. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*. ACM, New York, NY, USA, 289–295.

[9] Herman Zvonimir Došilović and Igor Mekterović. 2020. Robust and scalable online code execution system. In *Proceedings of the 43rd International Convention on Information, Communication and Electronic Technology*. IEEE, New York, NY, USA, 1627–1632. doi:10.23919/MIPRO48935.2020.9245310

[10] Hubert Dreyfus and Stuart E Dreyfus. 1986. *Mind over machine*. Simon and Schuster, Delanco, NJ, USA.

[11] Stephen H. Edwards and Krishnan Panamalai Murali. 2017. CodeWorkout: Short Programming Exercises with Built-in Data Collection. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy). ACM, New York, NY, USA, 188–193. doi:10.1145/3059009.3059055

[12] Eduard Frankford, Ingo Höhn, Clemens Sauerwein, and Ruth Breu. 2024. A Survey Study on the State of the Art of Programming Exercise Generation using Large Language Models. In *2024 36th International Conference on Software Engineering Education and Training (CSEE&T)*. IEEE, New York, NY, USA, 1–5. doi:10.1109/CSEET62301.2024.10662990

[13] Gemini Team, Google. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. arXiv:2403.05530

[14] Google. 2024. *Gemini 2*. https://cloud.google.com/vertex-ai/generative-ai/docs/gemini-v2

[15] HackerRank - Online Coding Tests and Technical Interviews. 2024. . Retrieved July 16, 2024 from https://hackerrank.com

[16] Junda He, Jieke Shi, Terry Yue Zhuo, Christoph Treude, Jiamou Sun, Zhenchang Xing, Xiaoning Du, and David Lo. 2025. From code to courtroom: LLMs as the new software judges. arXiv:2503.02246

[17] Michael J Hull, Daniel Powell, and Ewan Klein. 2011. Infandango: automated grading for student programming. In *Proceedings of the 16th annual Joint Conference on Innovation and Technology in Computer Science Education*. ACM, New York, NY, USA, 330. doi:10.1145/1999747.1999841

[18] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. arXiv:2406.00515

[19] Jessica Lam, Elias Fang, Majed Almansoori, Rahul Chatterjee, and Adalbert Gerald Soosai Raj. 2022. Identifying gaps in the secure programming knowledge and skills of students. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education*. ACM, New York, NY, USA, 703–709. doi:10.1145/3478431.3499391

[20] LeetCode - The World's Leading Online Programming Learning Platform. 2024. . Retrieved July 16, 2024 from https://leetcode.com

[21] Chap Sam Lim, Keow Ngang Tang, and Liew Kee Kor. 2012. *Drill and practice in learning (and beyond)*. Springer, Boston, MA, USA, 1040–1043.

[22] Evanfiya Logacheva, Arto Hellas, James Prather, Sami Sarsa, and Juho Leinonen. 2024. Evaluating contextually personalized programming exercises created with generative AI. In *Proceedings of the 2024 ACM Conference on International Computing Education Research-Volume 1*. ACM, New York, NY, USA, 95–113. doi:10.1145/3632620.3671103

[23] Niklas Meiβner, Sandro Speth, and Steffen Becker. 2024. Automated programming exercise generation in the era of large language models. In *2024 36th International Conference on Software Engineering Education and Training (CSEE&T)*. IEEE, New York, NY, USA, 1–5. doi:10.1109/CSEET62301.2024.10662984

[24] MITRE. 2024. *CVE Website*. Retrieved July 20, 2024 from https://cve.org

[25] MITRE. 2024. *CWE - Common Weakness Enumeration*. Retrieved May 28, 2025 from https://cwe.mitre.org

[26] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774

[27] OpenAI. 2024. *Hello GPT-4o*.

[28] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2025. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. *Commun. ACM* 68, 2 (Jan. 2025), 96–105. doi:10.1145/3610721

[29] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2023. Do Users Write More Insecure Code with AI Assistants?. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, New York, NY, USA, 2785–2799. doi:10.1145/3576915.3623157

[30] Mohan Rathakrishnan, Arumugam Raman, Mohamed Ali B Haniffa, Saralah Devi Mariamdaran, and Azlina Binti Haron. 2018. The drill and practice application in teaching science for lower secondary students. *International Journal of Education, Psychology and Counseling* 3, 7 (March 2018), 100–108.

[31] Laria Reynolds and Kyle McDonell. 2021. Prompt programming for large language models: Beyond the few-shot paradigm. In *Extended abstracts of the 2021 CHI conference on human factors in computing systems*. ACM, New York, NY, USA, 1–7. doi:10.1145/3411763.3451760

[32] Maria Ivanilse Calderon Ribeiro and Odette Mestrinho Passos. 2020. A Study on the active methodologies applied to teaching and learning process in the computing area. *IEEE Access* 8 (2020), 219083–219097. doi:10.1109/ACCESS.2020.3036976

[33] Andrew Sanders, Gursimran Singh Walia, and Andrew Allen. 2024. Assessing Common Software Vulnerabilities in Undergraduate Computer Science Assignments. *Journal of The Colloquium for Information Systems Security Education* 11, 1 (2024), 1–8. doi:10.53735/cisse.v11i1.179

[34] Leo St. Amour and Eli Tilevich. 2025. Designing a Platform to Train Secure Programming Skills With Attack-and-Defend Exercises. In *Proceedings of the IEEE Global Engineering Education Conference*. IEEE, New York, NY, USA, 1–10. doi:10.1109/EDUCON62633.2025.11016492

[35] Nguyen Binh Duong Ta, Hua Gia Phuc Nguyen, and Swapna Gottipati. 2023. ExGen: Ready-to-use exercise generation in introductory programming courses. In *International Conference on Computers in Education*. APSCE, 104–113. doi:10.58459/icce.2023.953

[36] Blair Taylor and Shiva Azadegan. 2006. Threading secure coding principles and risk analysis into the undergraduate computer science and information systems curriculum. In *Proceedings of the 3rd Annual Conference on Information Security Curriculum Development*. ACM, New York, NY, USA, 24–29. doi:10.1145/1231047.1231053

[37] Blair Taylor and Shiva Azadegan. 2008. Moving beyond security tracks: integrating security in cs0 and cs1. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*. ACM, New York, NY, USA, 320–324. doi:10.1145/1352135.1352246

[38] Colby Tofel-Grehl and David F Feldon. 2023. Measuring the viability of maker technology adoption within classrooms: Evaluating teacher instructional and cognitive load through constructionism and making. In *Proceedings of FabLearn/-Constructionism 2023: Full and Short Research Papers*. ACM, New York, NY, 1–4. doi:10.1145/3615430.3615448

[39] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with ChatGPT. arXiv:2302.11382