SecureCodeRL: Security-Aware Reinforcement Learning for Code Generation with Partial-Credit Rewards

Suryansh Singh Sijwali
Dept. of Computer Science
Pennsylvania State University
University Park, PA, USA
sss6371@psu.edu

***Abstract*—Large Language Models (LLMs) can generate plausible code, but in settings that require exact stdin/stdout behavior they frequently produce programs that compile yet fail tests, and in some cases they introduce security-sensitive patterns. This paper presents SecureCodeRL, a reinforcement learning (RL) pipeline for security-aware code generation that optimizes a combined reward $R = \alpha R_{\text{func}} + \beta R_{\text{sec}}$. The key idea is a *partial-credit* functional reward that assigns intermediate scores for syntactic validity, successful execution, and producing output, reducing reward sparsity that otherwise stalls learning on competitive-programming style tasks. I evaluate supervised fine-tuning (SFT) and PPO variants on a small held-out prompt set from APPS+ and observe that PPO with partial credit (using a continued-training variant) improves syntax validity from 45% (SFT) to 60% and achieves the only non-zero test success in my pilot (5% at-least-one-test-pass) while remaining 100% clean under Bandit static analysis in my evaluation.**

***Index Terms*—code generation, reinforcement learning, PPO, reward shaping, software security, static analysis**

I. Introduction

Code-capable LLMs are now widely used as programming assistants, but "looks correct" is not the same as *is correct*. In competitive-programming style problems, the program must parse input exactly, implement the right algorithm, and print output in the expected format. In practice, many generations fail due to wrong output, timeouts, crashes, or even producing no output at all. In parallel, there is a security angle: prior work has found that model suggestions can include insecure patterns and risky APIs [9].

This project grew out of a simple question I kept running into while benchmarking code models: *If the reward is "all tests pass," how does an RL system learn anything when almost nothing passes?* In my APPS+ runs, the most common failures were not dramatic crashes; they were "nearly-right" programs that were off by formatting, missed a print statement, or computed a value but never emitted it. A binary pass/fail reward treats these failures the same as syntax errors, which makes reward sparse and learning brittle.

SecureCodeRL is my attempt to make that learning signal less fragile. I do two things at once:

- I optimize for **functional correctness** using test execution feedback.
- I incorporate **security awareness** using static analysis (Bandit) as a penalty term.

To bridge the gap between "fails everything" and "passes everything," I introduce a **partial-credit** R_{func} that explicitly rewards intermediate milestones that matter in stdin/stdout judges.

The full codebase for this project is publicly available.[1]

A. Contributions

I make the following contributions:

- **Evidence-driven motivation via benchmarking.** I evaluate multiple open code models on APPS+ and quantify how often they produce syntactically valid code, pass tests, and remain security-clean.
- **A joint objective for correctness and security.** I train with $R = \alpha R_{\text{func}} + \beta R_{\text{sec}}$, where R_{sec} is derived from Bandit findings.
- **Partial-credit reward shaping.** I design an execution-stage reward that distinguishes syntax errors from runnable-but-silent programs (the "missing print" pattern) and from partial test matches.
- **Pilot RL results.** On a small held-out prompt set, PPO with partial credit improves syntax validity and is the only evaluated variant to achieve non-zero test success while remaining Bandit-clean.

II. Background and Related Work

A. Code generation and execution feedback

Recent code LLMs (e.g., DeepSeek-Coder, Code Llama, StarCoder2) are trained primarily with next-token prediction on large code corpora and can produce fluent solutions, but correctness under strict unit tests remains challenging [3]–[5]. A natural direction is to close the loop with compiler/test

[1]Repository: https://github.com/SuryanshSS1011/basic-rl-feedback-workflow

TABLE I: Benchmark summary on stdin-style APPS+ subset (my runs). *Test Pass* denotes the percentage of prompts where the generated program passes **all** provided tests. *Security Clean* is computed using Bandit findings in my pipeline (with an additional lightweight regex heuristic used during benchmarking for debugging); Bandit is the primary metric reported.

Model	Syntax Valid	Test Pass	Security Clean
DeepSeek-6.7B	83.4%	14.3%	96.3%
CodeLlama-7B	48.9%	7.6%	99.5%
StarCoder2-7B	20.2%	1.3%	99.7%

feedback. Prior work has explored reinforcement learning for code generation using execution signals, including CodeRL [6] and reinforcement learning from unit test feedback [7]. These ideas motivate SecureCodeRL: define reward using tools that directly reflect what a judge or developer cares about.

B. Security analysis as a training signal

Static analyzers provide an operational way to discourage insecure patterns. In Python, Bandit flags a range of risky constructs (e.g., use of `eval`, unsafe subprocess usage, weak cryptography) [8]. In SecureCodeRL, I treat static-analysis findings as a penalty shaping term: code that passes tests but triggers serious findings should receive a lower reward.

III. Dataset, Benchmark, and Failure Modes

A. Dataset

I use the APPS family of competitive programming problems [1], and specifically the APPS+ distribution used in my implementation [2].[2] Each problem includes a natural language statement and multiple stdin/stdout test cases. This stdin-style format is exactly where small I/O mistakes (missing prints, extra labels, whitespace) can cause total failure.

B. Multi-model benchmark: the gap between "valid" and "correct"

Before training SecureCodeRL, I benchmarked several models to quantify baseline difficulty. Table I reports the summary table I used to motivate this project: even when models generate syntactically valid code, *full* test success remains low. This gap between *parses* and *passes* is the core reason I wanted an RL loop driven by execution. (Later, in my pilot RL evaluation, I also report an "at least one test passes" indicator to detect the first signs of functional learning at small N.)

The headline is not that "models are bad"—it is that the learning signal is *sparse*. If only a small fraction of generations ever fully pass, then binary reward (1 if all tests pass, else 0) provides almost no gradient. That is exactly the regime where reward shaping matters.

[2]Direct data file used in my pipeline: https://raw.githubusercontent.com/Ablustrund/APPS_Plus/refs/heads/main/data/v1/data.json.

TABLE II: Compact failure taxonomy observed during my benchmark analysis (approximate shares).

Failure type	Share of failures
Wrong output	∼60%
No output	∼25%
Timeout/hang	∼8%
Crash	∼5%
Memory error	∼2%

C. Failure taxonomy: what actually goes wrong

To design a reward that helps PPO learn in this regime, I looked at *how* generations fail under the test harness. The dominant outcomes fall into a small set of categories shown in Table II. The key point is that many failures are "near misses" (especially wrong output and no output), which a binary reward collapses into the same zero as a syntax error. These proportions come from my benchmark logs over the stdin-style APPS+ subset (Section III-B).

This taxonomy is the bridge from benchmark to reward design: partial credit is explicitly shaped to separate *syntax*, *runtime stability*, and *I/O behavior* (the no-output bucket) before demanding full correctness.

D. Qualitative failure modes (representative examples)

Numbers are helpful, but for code generation the patterns are often clearer in short snippets.

1) Failure mode A: the "missing print" bug (silent programs): A frequent pattern is code that reads input and computes a value, but never prints it. Under binary reward this is indistinguishable from a syntax error.

```
# Reads input and computes, but produces NO
    OUTPUT
n = int(input())
(n * 2)    # expression is evaluated and
    discarded
```

2) Failure mode B: wrong-format output: Some generations print output, but with labels or extra text (strict judges reject it):

```
n = int(input())
print("Result:", n * 2)    # wrong format for a
     judge
```

3) Failure mode C: insecure shortcuts (e.g., `eval`): In a minority of cases, models reach for insecure shortcuts. Even if a shortcut "works," I want the training signal to discourage it.

```
formula = "a + b"
out = eval(formula)  # risky shortcut; Bandit
    flags this pattern
```

IV. Method

A. Problem formulation

Given a prompt x, a code model π_θ generates a program $y \sim \pi_\theta(\cdot \mid x)$. I define a scalar reward:

$$R(y) = \alpha \cdot R_{\text{func}}(y) + \beta \cdot R_{\text{sec}}(y), \tag{1}$$

with $\alpha = 0.6$ and $\beta = 0.4$ in my experiments. The functional term measures correctness under test execution, and the security term penalizes static-analysis findings.

B. Security reward: R_{sec}

I run Bandit on generated Python code and map findings to a penalty score in $[0, 1]$. In the pilot evaluation reported later, all generations were Bandit-clean, so $R_{\text{sec}} = 1.0$ across models. The framework is still useful: when insecure patterns appear (e.g., `eval`, unsafe subprocess usage), the reward would drop, pushing the policy away from those shortcuts [8].

C. Functional reward: why binary feedback is not enough

A strict functional reward can be written as:

$$R_{\text{func}}(y) = \frac{\#\text{tests passed}}{\#\text{tests total}}.$$

This is principled, but in the regime suggested by Table I it becomes effectively sparse: many generations fail all tests, and the model cannot easily learn the difference between "almost correct" and "nonsense."

D. Partial-credit functional reward (key idea)

The failure taxonomy in Table II makes the design goal concrete. If a large chunk of failures are *no output*, I need the reward to explicitly reward output production. If a chunk are *crashes*, I want to reward runtime stability. I therefore define R_{func} as a staged score in $[0, 1]$ that distinguishes:

- syntax validity,
- running without runtime error,
- producing any stdout,
- partial test matches,

with full credit reserved for passing tests.

V. Experiments

A. Models and training

I start from DeepSeek-Coder-1.3B-Instruct [3] and adapt it with LoRA. I train an SFT baseline on stdin-style APPS+ problems, then run PPO variants initialized from SFT (and, in one variant, continued from a prior PPO checkpoint). The overall training flow is summarized in Fig. 1.

TABLE III: SecureCodeRL evaluation results (20 prompts per model).

Model	Syntax %	$\geq$1 Test Pass %	Security %	Mean R
SFT Baseline	45.0	0.0	100.0	0.40
PPO-simple	15.0	0.0	100.0	0.40
PPO-fresh	25.0	0.0	100.0	0.40
PPO-continue	**60.0**	**5.0**	100.0	**0.41**

B. Evaluation protocol and metrics

For evaluation, I run an execution harness on a held-out set of prompts and compute:

- **Syntax Valid %**: parses/compiles as Python.
- **At-least-one-test-pass %**: fraction of prompts where the generated program passes *at least one* test case.
- **Security Clean %**: no Bandit findings.
- **Mean rewards**: R_{func}, R_{sec}, and combined R.

I use the "at least one test passes" metric in this pilot because it is sensitive to early signs of learning when N is small; in a scaled evaluation, I would also report (and emphasize) the stricter all-tests-pass rate used in the benchmark in Section III-B.

My evaluation is intentionally small and focused (20 prompts per model) to validate the pipeline end-to-end and to check whether partial credit produces non-trivial learning signals. I treat the resulting numbers as a pilot study rather than a final benchmark.

VI. Results

A. Quantitative comparison (pilot)

Table III summarizes my evaluation results (20 prompts per model). The headline result is that PPO with partial credit (continued from PPO-simple) achieves the highest syntax validity (60%) and is the only evaluated model with a non-zero test success signal (5% at-least-one-test-pass) under my protocol, while remaining 100% Bandit-clean.

A detail worth emphasizing is that the absolute test success rate is still low. In this evaluation, "5%" corresponds to one prompt for which the model passed at least one test. That said, the fact that *only* the partial-credit PPO-continue variant crosses zero is consistent with the original motivation: binary rewards can keep PPO stuck, and partial credit can help it reach the first rung of functional success.

B. Security results: Bandit vs. heuristic findings

All evaluated generations were Bandit-clean, so R_{sec} did not differentiate models in this pilot. I also tracked a lightweight regex-based heuristic counter during experiments; those heuristic findings increased in PPO-continue, which I interpret as a byproduct of longer/more varied generations triggering simplistic patterns. For reporting security in the paper, I treat Bandit as the reference and heuristics as debugging tools.

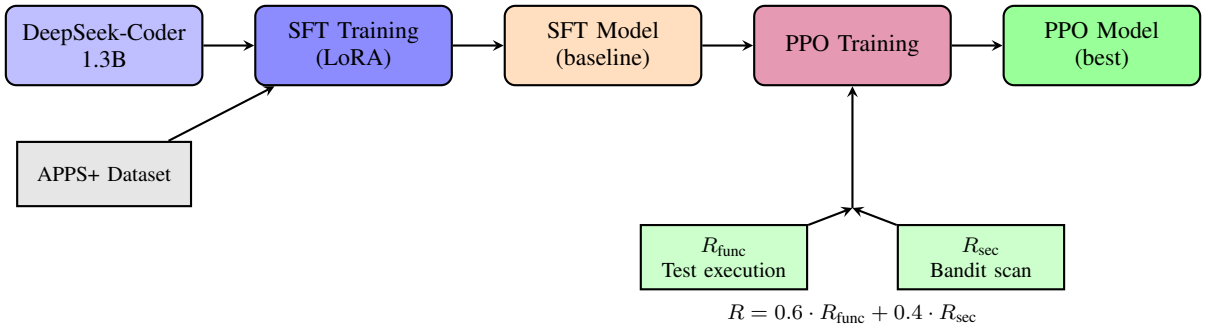

Fig. 1: SecureCodeRL pipeline: SFT (LoRA) initializes a policy, then PPO optimizes a combined reward. Functional and security rewards merge into a single signal before updating the PPO policy.

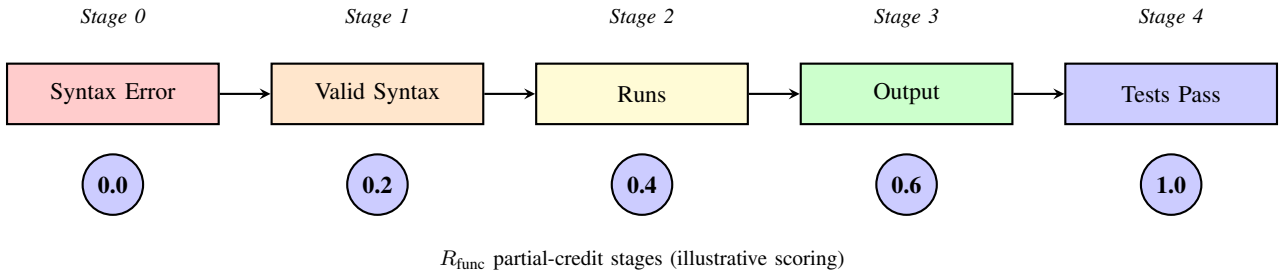

Fig. 2: Partial-credit functional reward: instead of binary test pass/fail, I assign intermediate scores for syntax validity, successful execution, and producing output, with full credit reserved for passing tests.

VII. Discussion

A. Why partial credit helped in this setting

The partial-credit design follows directly from the failure taxonomy in Table II. If a model reads input and computes a value but fails to print it, a strict test reward returns zero, even though the fix is small. Partial credit gives these programs a non-zero score (syntax $\rightarrow$ runs $\rightarrow$ output) and makes it easier for PPO to climb the ladder instead of restarting from scratch each episode.

Another practical benefit is that staged rewards help me debug training. When the reward is always zero, it is hard to tell whether the model is making progress on syntax, runtime stability, or I/O behavior. With partial credit, the reward itself becomes a diagnostic signal.

B. What I still need for stronger validation

The pilot results in Table III are enough to validate the pipeline end-to-end and to show that partial credit changes PPO's behavior in the direction I intended: PPO-continue is the only variant in my evaluation that achieves non-zero test success, and it substantially improves syntax validity. At the same time, I do not want to over-interpret what a 20-prompt study can establish. To strengthen the empirical validation, I see three concrete upgrades.

First, I need a larger and more systematic evaluation. The most direct change is to scale from $N = 20$ prompts per model to at least a few hundred, stratified by APPS+ difficulty. With small N, a single success can move the test metric by multiple points, so the current estimate has high variance.

Second, I need uncertainty estimates and controlled ablations. I plan to report bootstrap confidence intervals for Syntax % and test success, and I will add comparisons that isolate each design choice: PPO with a binary functional reward vs. PPO with partial credit; partial credit with and without the security term; and security shaping with a binary functional reward. These ablations directly test whether the improvements come from reward design rather than initialization effects or noise.

Third, I need stronger baselines and broader security coverage. On the functionality side, I should compare against inference-time baselines such as best-of-k sampling with test-based selection, to separate the value of training from simply sampling more candidates. On the security side, Bandit is a good starting point for Python, but adding additional analyzers (or a curated vulnerability suite) would make the security claim more robust.

Finally, I will include a short qualitative appendix with paired examples that make the reward shaping behavior explicit: a missing-print program fixed into a correct one; an output-formatting failure corrected; a timeout avoided; and at least one case where R_{sec} prevents a risky shortcut. These examples make the mechanism legible and connect the quantitative metrics back to concrete programmer-visible behavior.

C. Limitations

There are several limitations to keep in mind:

- **Small evaluation set.** The reported 5% test success comes from 1/20 prompts in this pilot.
- **Security findings rarity (in pilot eval).** Bandit findings were absent in the evaluated samples, so the security component did not meaningfully differentiate models in this run.
- **Competitive-programming bias.** APPS+ emphasizes stdin/stdout and algorithmic correctness; results may not transfer directly to other coding tasks (APIs, libraries, applications).

VIII. Conclusion

This paper presents SecureCodeRL, a security-aware RL pipeline for code generation with a combined objective $R = \alpha R_{\text{func}} + \beta R_{\text{sec}}$ and a partial-credit functional reward to mitigate sparse test feedback. I first motivate the problem with a multi-model benchmark (Table I) and a compact failure taxonomy (Table II) that explains why binary rewards are brittle. In a small pilot evaluation on APPS+, PPO with partial credit (using a continued-training variant) achieves the strongest results among the variants I tested: higher syntax validity and the only non-zero test success signal, while remaining clean under Bandit static analysis. The main takeaway is not that the problem is solved; it is that reward shaping can meaningfully change whether PPO learns anything at all in strict unit-test environments. The next step is scaling evaluation and adding ablations to turn this proof-of-concept into a fully benchmarked result.

References

[1] D. Hendrycks, S. Basart, S. Kadavelil, V. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt, "Measuring Coding Challenge Competence with APPS," in *NeurIPS*, 2021.

[2] A. Blustrund, "APPS_Plus," GitHub repository, 2025. [Online]. Available: https://github.com/Ablustrund/APPS_Plus.
Direct data file used in my pipeline: https://raw.githubusercontent.com/Ablustrund/APPS_Plus/refs/heads/main/data/v1/data.json.

[3] D. Guo, Q. Yang, J. Zhang, Z. Wang, J. Li, and others, "DeepSeek-Coder: When the Large Language Model Meets Programming," *arXiv preprint arXiv:2401.14196*, 2024.

[4] B. Rozière, J. Gehring, G. Gloeckle, S. Soudry, and others, "Code Llama: Open Foundation Models for Code," *arXiv preprint arXiv:2308.12950*, 2023.

[5] R. Li and the BigCode Project, "StarCoder 2 and The Stack v2: The Next Generation," *arXiv preprint arXiv:2402.19173*, 2024.

[6] H. Le, Y. Li, H. Liang, and others, "CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning," in *NeurIPS*, 2022.

[7] J. Liu, Y. Liu, and others, "Reinforcement Learning from Unit Test Feedback," *arXiv preprint arXiv:2307.04349*, 2023.

[8] PyCQA, "Bandit: A tool designed to find common security issues in Python code," Documentation. [Online]. Available: https://bandit.readthedocs.io/.

[9] H. Pearce, B. Tan, B. Ahmad, R. Karri, and others, "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions," in *IEEE Symposium on Security and Privacy (S&P)*, 2022.