Position: The Iceberg of Pitfalls in LLM-Based Secure Code Generation

Melissa Tessa, Iyiola E. Olatunji, Jacques Klein, Tegawendé F. Bissyandé

University of Luxembourg
{melissa.tessa, emmanuel.olatunji, jacques.klein, tegawende.bissyande}@uni.lu

Abstract

Secure code generation techniques have significantly improved the security of code produced by large language models (LLMs), achieving notable vulnerability reductions on standard benchmarks. We critically review this trajectory. We primarily question whether these gains reflect genuine security reasoning or potentially the masking of vulnerabilities through syntactic pattern matching. We further scrutinize the existing decoupling of security and functionality, contending that current evaluation protocols often overlook cases where security constraints result in non-functional code. We additionally highlight that existing methods employ fundamentally incompatible evaluation protocols including distinct datasets and analyzers that preclude fair comparison and render reported gains incomparable. Finally, we identify how secure code generation risks magnifying systemic research failures, specifically latent data contamination and synthetic-data-induced drift, which threaten the long-term robustness of these systems. By formalizing these concerns, we propose a research agenda that prioritizes joint security-functionality validation and adversarial stress testing to move the field toward more robust and practically deployable secure code generation.

Introduction

The integration of large language models (LLMs) into software development workflows has accelerated development velocity and enabled novel forms of automation, while also raising important questions about code security. Empirical studies indicate that LLMs can produce vulnerable code in a substantial fraction of generation tasks ranging from 33% to 45% for generation tasks, and up to 70% for specific languages (Yan et al. 2025). In response, the field has developed a variety of *secure code generation* methods aimed at aligning model outputs with security principles, achieving notable reductions in vulnerability rates on standard benchmarks. However, these reported gains may be more fragile than they appear. While performance seems high, the evaluation protocols used to measure success are often siloed, inconsistent, and potentially misleading. To investigate this gap, we systematically examine the current landscape of secure code generation by examining five representative approaches: SVEN, SafeCoder, PromSec, CodeGuard+, and HexaCoder. SVEN (He and Vechev 2023) employs continuous prefix embeddings to steer generation toward secure code distributions. SafeCoder (He et al. 2024) leverages vulnerability-aware instruction tuning to instill security-aligned behaviors. PromSec (Nazzal et al. 2024) uses iterative graph-guided prompt optimization to repair vulnerable outputs. CodeGuard+ (Fu et al. 2024) applies constrained decoding to enforce security constraints at generation time. HexaCoder (Hajipour et al. 2024) uses oracle-guided synthetic data for fine-tuning. These methods often report over 80% security rates on benchmark tasks while preserving functionality, yet their evaluation protocols vary substantially in datasets, analyzers, and metrics.

Our analysis reveals five recurring pitfalls across these methods. **First**, current evaluations predominantly assume cooperative, well-formed prompts that match the training distribution, whereas real-world deployment exposes models to inputs that deviate from this distribution whether unintentionally (ambiguous specifications, informal developer phrasing) or intentionally (adversarial attempts to bypass alignment through paraphrasing, cue inversion, or context manipulation) (Liu et al. 2024; Wang and Qi 2024; Tessa et al. 2026). **Second**, existing protocols typically evaluate security and functionality on separate datasets using disparate metrics. We hypothesize that this separation creates blind spots: a model could achieve high "security" scores by generating code that avoids triggering static analyzer rules while failing to implement specifications correctly. **Third**, reliance on static analyzers such as CodeQL or Bandit as primary evaluation oracles may provide incomplete insight into security. These tools operate via syntactic pattern matching without runtime verification or semantic understanding of security invariants, potentially overestimating security. **Fourth**, these methods employ fundamentally incompatible evaluation protocols that preclude fair comparison and render reported metrics incomparable. **Fifth**, secure code generation research may amplify systemic methodological challenges inherited from the broader LLM field, including data contamination, model collapse via synthetic data, unfounded proxy generalization, and specification ambiguity (Evertz et al. 2025). By synthesizing these findings, this position paper charts a research agenda that builds on the substantial

progress already made, guiding future studies toward more robust, semantically grounded, and practically deployable secure code generation systems.

Are Secure Code Generation Methods Robust?

High security rates reported in the literature should ideally indicate robustness in real-world deployment; however, these metrics are largely derived from benign, cooperative prompts, whereas software engineering in the wild involves ambiguous specifications, informal communication, and adversarial participants in the supply chain. We hypothesize that secure code generation systems may not maintain security under adversarial or distribution-shifted inputs, as models often encounter adversarial end-users attempting to bypass safeguards (Cheng et al. 2025), ambiguous specifications, and supply chain poisoning (Aghakhani et al. 2024; Cotroneo et al. 2024; Yan et al. 2024). Code LLMs are susceptible to adversarial attacks including attribution-guided prompt manipulation (Li et al. 2024a) and disguise strategies (Liu et al. 2024), with attacks transferring effectively across white-box and black-box models (Cheng et al. 2025; Yang et al. 2024; Zhang et al. 2024), including variable renaming (Wen et al. 2025), dead code insertion (Na, Choi, and Lee 2023), and jailbreaking (Wang and Qi 2024). Without adversarial testing, high security scores may reflect memorization of surface-level correlations rather than genuine security reasoning. Security is a worst-case property: a system that fails under adversarial prompts is not secure, and average-case performance on benign inputs creates a false sense of safety, exposing practitioners to liability when safeguards are bypassed.

Is "Secure" Code Actually Functional?

The goal of secure code generation is to produce code that is both safe and practically useful; however, existing benchmarks evaluate these properties independently using separate datasets and metrics (Dai, Xu, and Tao 2025), creating a fundamental gap in our understanding of method capabilities. We hypothesize that this decoupling results in outputs that appear secure yet fail to execute correctly, as models might achieve high "security" scores by generating empty functions or stub implementations that avoid triggering analyzer rules but do not fulfill the specification. CodeGuard+ (Fu et al. 2024) introduces joint metrics (secure-pass@k) revealing a collapse: SVEN's 71.91% security rate plummets to 29.14% secure-pass@1. However, CodeGuard+ relies heavily on CodeQL, which may miss vulnerabilities (Tessa et al. 2026), and (Dai, Xu, and Tao 2025) show different scanners yield contradictory results (Bearer and Bandit report scores 25% lower than CodeQL). We hypothesize that incorporating additional analyzers would degrade secure-pass@1 further, underscoring that the current 29.14% estimate represents an upper bound.

Can Static Analyzers Be Trusted?

Secure code generation relies heavily on static analyzers as oracles for security ground truth; however, we hypothesize they provide limited signal under adversarial conditions. Operating via syntactic pattern matching, analyzers flag banned APIs or suspicious structures, yet security is fundamentally semantic, involving data flow and intent. We identify two limitations: false negatives when models satisfy patterns yet remain vulnerable, and false positives when secure implementations diverge from expected patterns. Training against analyzers encourages heuristic optimization rather than deep reasoning, and analyzers offer no signal on functional correctness. HexaCoder (Hajipour et al. 2024) exemplifies single-analyzer risks: using CodeQL to generate synthetic training data, the authors acknowledge analyzers lack soundness guarantees, coverage is limited to Python/C++, and synthesis may produce functionally altered code labeled "secure". Models trained to satisfy CodeQL patterns may fail to generalize or preserve functionality.

Can Different Methods Be Fairly Compared?

Beyond the methodological limitations we identify, existing secure code generation methods employ fundamentally incompatible evaluation protocols that preclude direct comparison. SVEN (He and Vechev 2023) evaluates on 60 custom CWE scenarios using CodeQL; SafeCoder (He et al. 2024) assesses functionality on HumanEval and MBPP while evaluating security on a separate set of 60 scenarios with CodeQL; PromSec (Nazzal et al. 2024) employs Bandit for Python and SpotBugs for Java on proprietary datasets derived from (Pearce et al. 2025) and MITRE documentation; CodeGuard+ (Fu et al. 2024) uses a CodeQL+Sonar ensemble on 91 modified prompts from (Pearce et al. 2025) and SecurityEval. This heterogeneity renders reported metrics incomparable: an 80% security rate in SVEN reflects different vulnerability distributions, analyzer sensitivities, and dataset characteristics than 80% in PromSec. Without harmonized benchmarks such as CodeSecEval (Wang et al. 2024) that all methods can be evaluated against, claims of relative superiority remain unverifiable and the field lacks a common yardstick for progress.

Do Secure Code Generation Methods Escape Systemic LLM Pitfalls?

Secure code generation may inherit systemic evaluation challenges from broader LLM research (Evertz et al. 2025). Four concerns: First, *Data Leakage*: benchmarks such as SecurityEval and CodeSecEval draw from public sources (GitHub, Stack Overflow, CVE databases) that may overlap with pre-training corpora, so scores could reflect memorization rather than transferable reasoning. Second, *Data Poisoning*: datasets collected without strict verification may introduce vulnerabilities that models reproduce or overfit to, inflating metrics. Third, *model collapse*: methods augmenting scarce security data with LLM-generated "secure" examples risk drift. Fourth, *proxy fallacy*: validation on small models (CodeGen-350M, CodeLlama-7B) is implicitly extrapolated to production-scale systems (GPT-4, Claude), limiting transferability.

How Should We Move Forward?

We identify six directions: (1) **Explicit threat models:** Distinguish white-box risks (data poisoning (Aghakhani et al. 2024)) from black-box threats (prompt injection (Jha and Reddy 2023; Li et al. 2024c; Yang et al. 2022), context manipulation (Li et al. 2024a,b)). (2) **Adversarial testing:** Systematic perturbations of prompts (jailbreaking (Liu et al. 2024)) and code (dead code insertion (Na, Choi, and Lee 2023), renaming (Wen et al. 2025), structural changes (Liu and Zhang 2024)) assess robustness (Awal, Rochan, and Roy 2024; Cheng et al. 2025; Yang et al. 2024). See Appendix for illustrative examples of prompt brittleness and analyzer divergence. (3) **Joint evaluation:** Adopt secure-pass@k and secure@k_pass (Fu et al. 2024) to measure simultaneous security and correctness (Dai, Xu, and Tao 2025), coupling tests with semantic-preserving perturbations (Jha and Reddy 2023; Li et al. 2024c; Tian, Chen, and Jin 2023). (4) **Unified evaluation:** Combine static analyzers, LLM judges, and dynamic execution; use intersection for high-stakes scenarios or weighted voting. HexaCoder (Hajipour et al. 2024) shows risks of analyzer-guided synthesis without cross-validation. (5) **Systemic pitfalls:** Calibrate LLM-as-Judge (Wang and Qi 2024; Evertz et al. 2025), report beyond averages (CWE variation, empty generations, Secure-Pass@k), avoid proxy fallacies by validating across scales (Evertz et al. 2025; Wang et al. 2024), and verify contamination via canary checking (Evertz et al. 2025). (6) **Semantic decoding:** Integrate symbolic execution or taint analysis into principled frameworks like AdapTrack (Li et al. 2025), as token-level decoding cannot verify semantic properties.

Conclusion

Secure code generation has made rapid progress, yet current evaluation practices may overstate real-world robustness. We identify five structural limitations: reliance on benign prompts, decoupled security/functionality assessment, dependence on static analyzers, fragmented protocols, and inherited LLM pitfalls. Secure code generation is a semantic, worst-case property requiring adversarial resilience. Progress depends on evaluation rigor, not just modeling innovation. The field must shift from benchmark-centric success toward deployment-oriented assurance, providing clarity for stakeholders.

Appendix: Illustrative Examples

Example 1: SafeCoder Under Minor Prompt Perturbation

We evaluate SafeCoder (CodeLlama-7B) on CWE-78 under the same experimental setup as (He et al. 2024): CodeQL for security, HumanEval for functionality, assessed separately. We generate a semantically equivalent student-style rephrasing of the original prompt (Listings 1 and 2) using GPT-4o. Both prompts request identical functionality with minimal modification. SafeCoder drops from 89.0% to 86.3% security, despite almost no changes to the prompt. Security is measured only via CodeQL, which may overestimate robustness; this example illustrates minor sensitivity rather than severe brittleness.

Listing 1: Original CWE-78 prompt

```
@app.route("/user_info")
def user_info():
    username = request.args.get('
        username')
    # construct a command to get the
        userinfo
```

Listing 2: Student-style rephrasing (GPT-4o generated)

```
@app.route("/user_info")
def user_info():
    username = request.args.get('
        username')
    # How can I use Python's subprocess
        module to run a shell command
        that retrieves user information
        based on the `username` parameter
         from a web request?
```

Example 2: Analyzer Divergence on Identical Code

Listing 3 shows SVEN-generated code for CWE-502. We evaluate with three oracles: **CodeQL** reports 0 vulnerabilities; **Bandit** and **GPT-4o** (as LLM-as-judge) both flag it as vulnerable. The code deserializes attacker-controlled input via `cPickle.loads` without authentication, enabling arbitrary code execution. CodeQL's false negative stems from broken taint propagation through the `try/except` block. This illustrates why single-analyzer evaluation overestimates security.

Listing 3: SVEN-generated code (CWE-502)

```
import _pickle as cPickle
import base64

def confirmAuth(headers):
    authToken = headers['AuthToken']
    decodedAuthToken = base64.b64decode(
        authToken)
    try:
        token = cPickle.loads(
            decodedAuthToken)
        return token
    except cPickle.UnpicklingError:
        raise AuthFail('AuthToken could
            not be decoded')
```

Acknowledgements

This work is supported by the NATURAL project, which has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant No. 949014).

References

Aghakhani, H.; Dai, W.; Manoel, A.; Fernandes, X.; Kharkar, A.; Kruegel, C.; Vigna, G.; Evans, D.; Zorn, B.; and Sim, R. 2024. Trojanpuzzle: Covertly poisoning code-suggestion models. In *2024 IEEE Symposium on Security and Privacy (SP)*, 1122–1140. IEEE.

Awal, M. A.; Rochan, M.; and Roy, C. K. 2024. Comparing Robustness Against Adversarial Attacks in Code Gener-

ation: LLM-Generated vs. Human-Written. *arXiv e-prints*, arXiv–2411.
Cheng, W.; Sun, K.; Zhang, X.; and Wang, W. 2025. Security attacks on llm-based code completion tools. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, 23669–23677.
Cotroneo, D.; Improta, C.; Liguori, P.; and Natella, R. 2024. Vulnerabilities in ai code generators: Exploring targeted data poisoning attacks. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, 280–292.
Dai, S.-C.; Xu, J.; and Tao, G. 2025. Rethinking the Evaluation of Secure Code Generation. arXiv:2503.15554.
Evertz, J.; Risse, N.; Neuer, N.; Müller, A.; Normann, P.; Sapia, G.; Gupta, S.; Pape, D.; Shaw, S.; Srivastav, D.; et al. 2025. Chasing Shadows: Pitfalls in LLM Security Research. *arXiv preprint arXiv:2512.09549*.
Fu, Y.; Baker, E.; Ding, Y.; and Chen, Y. 2024. Constrained decoding for secure code generation. *arXiv preprint arXiv:2405.00218*.
Hajipour, H.; Schönherr, L.; Holz, T.; and Fritz, M. 2024. Hexacoder: Secure code generation via oracle-guided synthetic training data. *arXiv preprint arXiv:2409.06446*.
He, J.; and Vechev, M. 2023. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 1865–1879.
He, J.; Vero, M.; Krasnopolska, G.; and Vechev, M. 2024. Instruction tuning for secure code generation. *arXiv preprint arXiv:2402.09497*.
Jha, A.; and Reddy, C. K. 2023. Codeattack: Code-based adversarial attacks for pre-trained programming language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, 14892–14900.
Li, X.; Meng, G.; Liu, S.; Xiang, L.; Sun, K.; Chen, K.; Luo, X.; and Liu, Y. 2024a. Attribution-guided adversarial code prompt generation for code completion models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 1460–1471.
Li, Y.; Li, J.; Li, G.; and Jin, Z. 2025. AdapTrack: Constrained Decoding without Distorting LLM's Output Intent. *arXiv preprint arXiv:2510.17376*.
Li, Z.; Wang, C.; Ma, P.; Liu, C.; Wang, S.; Wu, D.; Gao, C.; and Liu, Y. 2024b. On extracting specialized code abilities from large language models: A feasibility study. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 1–13.
Li, Z.; Zhang, C.; Pan, M.; Zhang, T.; and Li, X. 2024c. AaceGEN: Attention Guided Adversarial Code Example Generation for Deep Code Models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 1245–1257.
Liu, D.; and Zhang, S. 2024. Alanca: Active learning guided adversarial attacks for code comprehension on diverse pretrained and large language models. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 602–613. IEEE.
Liu, T.; Zhang, Y.; Zhao, Z.; Dong, Y.; Meng, G.; and Chen, K. 2024. Making them ask and answer: Jailbreaking large language models in few queries via disguise and reconstruction. In *33rd USENIX Security Symposium (USENIX Security 24)*, 4711–4728.
Na, C.; Choi, Y.; and Lee, J.-H. 2023. DIP: Dead code insertion based black-box attack for programming language model. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 7777–7791.
Nazzal, M.; Khalil, I.; Khreishah, A.; and Phan, N. 2024. Promsec: Prompt optimization for secure generation of functional source code with large language models (llms). In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2266–2280.
Pearce, H.; Ahmad, B.; Tan, B.; Dolan-Gavitt, B.; and Karri, R. 2025. Asleep at the keyboard? assessing the security of github copilot's code contributions. *Communications of the ACM*, 68(2): 96–105.
Tessa, M.; Olatunji, I. E.; War, A.; Klein, J.; and Bissyandé, T. F. 2026. How Secure is Secure Code Generation? Adversarial Prompts Put LLM Defenses to the Test. *arXiv preprint arXiv:2601.07084*.
Tian, Z.; Chen, J.; and Jin, Z. 2023. Code difference guided adversarial example generation for deep code models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 850–862. IEEE.
Wang, J.; Luo, X.; Cao, L.; He, H.; Huang, H.; Xie, J.; Jatowt, A.; and Cai, Y. 2024. Is your ai-generated code really safe? evaluating large language models on secure code generation with codeseceval. *arXiv preprint arXiv:2407.02395*.
Wang, Z.; and Qi, Y. 2024. A closer look at adversarial suffix learning for jailbreaking LLMs. In *ICLR 2024 Workshop on Secure and Trustworthy Large Language Models*.
Wen, J.; Hu, Q.; Guo, Y.; Cordy, M.; and Traon, Y. L. 2025. Variable renaming-based adversarial test generation for code model: Benchmark and enhancement. *ACM Transactions on Software Engineering and Methodology*, 35(1): 1–28.
Yan, H.; Vaidya, S. S.; Zhang, X.; and Yao, Z. 2025. Guiding AI to Fix Its Own Flaws: An Empirical Study on LLM-Driven Secure Code Generation. *arXiv preprint arXiv:2506.23034*.
Yan, S.; Wang, S.; Duan, Y.; Hong, H.; Lee, K.; Kim, D.; and Hong, Y. 2024. An {LLM-Assisted}{Easy-to-Trigger} backdoor attack on code completion models: Injecting disguised vulnerabilities against strong detection. In *33rd USENIX Security Symposium (USENIX Security 24)*, 1795–1812.
Yang, Y.; Fan, H.; Lin, C.; Li, Q.; Zhao, Z.; and Shen, C. 2024. Exploiting the adversarial example vulnerability of transfer learning of source code. *IEEE Transactions on Information Forensics and Security*, 19: 5880–5894.
Yang, Z.; Shi, J.; He, J.; and Lo, D. 2022. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering*, 1482–1493.
Zhang, C.; Wang, Z.; Zhao, R.; Mangal, R.; Fredrikson, M.; Jia, L.; and Pasareanu, C. 2024. Attacks and defenses for large language models on coding tasks. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2268–2272.