Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Static Analysis-Guided Patch Synthesis for Correcting LLM-Generated Insecure Code

Miriam Koch

u0qz6ecrh9@mantonschool.us

Department of Computer Science, Technical University of Munich, Munich 80333, Germany

Daniel Weber

Lukas Schneider

Research Article

Keywords: static analysis, patch synthesis, vulnerability repair, SMT solving, secure code generation

Posted Date: March 3rd, 2026

DOI: https://doi.org/10.21203/rs.3.rs-8992030/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: The authors declare no competing interests.

Static Analysis-Guided Patch Synthesis for Correcting LLM-Generated Insecure Code

Authors

Miriam Koch[1], Daniel Weber[2], Lukas Schneider[3]*

Affiliations

Department of Computer Science, Technical University of Munich, Munich 80333, Germany

***Corresponding author:** lukas.schneider@tum.de

Abstract

LLMs often miss subtle security conditions such as boundary checks and authentication flows. We introduce a static-analysis-driven patch synthesizer that transforms insecure LLM output into correct and safe code. The pipeline uses abstract interpretation to detect violations, followed by SMT-guided repair rules. On 3,200 vulnerable code snippets across five vulnerability categories, our system achieves 74% full automatic repair and 92% partial repair, outperforming Codex and GPT-4 post-processing by a large margin. Developer usability evaluation shows 58% reduction in manual fix time. The method highlights how program analysis can systematically strengthen LLM-based secure coding.

Keywords: static analysis; patch synthesis; vulnerability repair; SMT solving; secure code generation

1. Introduction

Large language models (LLMs) are increasingly adopted in software development workflows, supporting tasks such as rapid prototyping, code completion, automated patching, and documentation generation. Their ability to synthesize functional code from natural language descriptions has led to widespread integration into development environments and build pipelines. However, recent empirical studies indicate that LLM-generated code frequently omits essential security checks, including boundary validation, input sanitization, authentication logic, and error handling, even when security risks are explicitly mentioned in prompts [1,2]. These deficiencies appear consistently across benchmark evaluations and real-world programming scenarios, raising concerns about the reliability of machine-generated code. Measurements conducted in production-like settings further show that LLM-generated programs exhibit a high incidence of memory safety violations, injection vulnerabilities, and incorrect use of cryptographic primitives [3]. Such weaknesses mirror known exploit patterns and suggest that automated code generation may introduce a new class of vulnerabilities that propagate rapidly through software ecosystems. Similar observations have been reported in other security-sensitive domains, where machine-learning–based systems are known to reproduce subtle but systematic error patterns unless explicitly constrained by robust analysis mechanisms [4]. These findings highlight a

growing need for principled methods that can detect and mitigate security flaws in LLM-generated code before deployment.

Recent research has begun to examine the security properties of LLM-produced programs in a more systematic manner. Several studies construct evaluation datasets to quantify how often generated code contains exploitable patterns and how sensitive the results are to prompt wording, in-context examples, or temperature settings [5]. Other approaches attempt to repair insecure outputs by injecting static-analysis warnings or vulnerability descriptions into prompts and asking the model to regenerate code [6]. While such prompt-based techniques can address straightforward mistakes, they rely heavily on heuristic interactions with the model and offer no formal guarantees. In practice, they often require multiple regeneration cycles and fail to correct deeper control-flow or data-flow flaws [7,8]. Parallel to these efforts, the program analysis and automated program repair communities have developed mature techniques for detecting and fixing vulnerabilities in human-written code. Static analysis based on abstract interpretation and dataflow tracking is widely deployed to identify buffer overflows, null dereferences, taint propagation, and access-control violations at scale [9]. Automated repair systems translate analysis findings into patches using templates, search-based exploration, or constraint solving [10]. More recent advances combine symbolic execution with SMT solving to synthesize precise guards, modify control-flow, or restructure data usage while preserving program semantics [11,12]. Despite their effectiveness, these systems were designed for conventional software and do not directly address the characteristics of LLM-generated code, which is often concise, underspecified, and structurally inconsistent. The growing use of LLMs in code repair has motivated hybrid approaches that combine program analysis with model-generated suggestions. Some systems provide execution traces, summaries, or counterexamples as additional context to guide the model toward better fixes [13]. Others allow LLMs to propose coarse-grained edits that are subsequently validated or filtered by analysis tools [14]. Although these approaches show promise, program analysis is typically used only as a post hoc validator. As a result, repair quality remains tightly coupled to model behavior, and the reasoning behind accepted or rejected fixes is often opaque. Several open challenges remain unresolved. Many existing evaluations rely on limited or proprietary datasets that do not capture the diversity of vulnerabilities observed in practice, reducing confidence in reported improvements [15,16]. Systems that depend on iterative prompting explore only a narrow subset of possible repairs and cannot ensure that all relevant safety conditions are satisfied. Moreover, little evidence exists regarding whether analysis-guided repair can scale to thousands of LLM-generated programs across heterogeneous vulnerability classes, or whether such methods can meaningfully reduce developer effort [17]. The interaction between abstract interpretation, which reasons over approximated program states, and SMT-based synthesis, which operates on concrete constraints, also remains underexplored in the context of machine-generated code.

This paper addresses these limitations by proposing a static-analysis–guided patch synthesis framework tailored to insecure LLM-generated programs. The framework first applies abstract

interpretation to detect violations of key safety properties, including array bounds, null dereferences, and missing authorization checks. Detected violations are then translated into an SMT-based synthesis problem, where patches are generated using a targeted set of repair rules. By separating vulnerability detection from patch synthesis, the framework establishes a transparent and explainable connection between analysis results and concrete code modifications, rather than relying on heuristic model regeneration. The approach is evaluated on 3,200 vulnerable LLM-generated programs spanning five common vulnerability categories. Experimental results demonstrate higher full-repair rates than LLM-only regeneration and post-processing baselines, along with significant reductions in manual debugging effort. These findings suggest that integrating abstract interpretation with solver-guided synthesis provides a scalable and principled path toward improving the security of LLM-generated code and supporting safer AI-assisted software development.

2. Materials and Methods

2.1 Sample Collection and Study Scope

This study uses 3,200 code snippets generated by large language models under fixed prompting conditions. The samples span five vulnerability types: out-of-bounds access, null dereference, unsafe input handling, missing authentication checks, and unchecked resource use. All snippets come from the same programming tasks so that they share a common functional goal. The outputs are taken from several model versions to include variation in generation behavior. Each snippet is collected in an isolated environment to avoid external dependencies. Before analysis, the code is cleaned and formatted to remove artifacts that may affect static checking.

2.2 Experimental Design and Control Conditions

Three repair methods are compared in this study: the proposed static-analysis-guided patch system, a baseline that regenerates code using an LLM, and a manual repair baseline completed by trained developers. All methods work on the same input snippets. The LLM regeneration serves as the main control condition because it reflects current automated coding practice. The experimental method applies abstract interpretation followed by SMT-based patch generation. Manual repair is added as a second control group to show an upper bound for human-driven correction. All methods use the same tasks, time limits, and vulnerability labels to ensure fair comparison.

2.3 Measurement Procedures and Quality Assurance

Static analysis is carried out using an abstract interpretation tool configured to track numeric ranges, pointer safety, and precondition checks. The tool records each violation and its location in the code. The SMT solver then uses these findings to generate candidate patches. Each patch undergoes three checks: property validation, compilation testing, and functional testing based on the original task. Quality assurance includes duplicate detection, repeated runs to confirm

consistent outcomes, and a manual review of part of the dataset by two independent security specialists. Any conflict in review results is documented and excluded from final reporting.

2.4 Data Processing and Model Formulation

Patch results are grouped into full repair, partial repair, and no repair, depending on whether all recorded violations are removed. Time cost and error counts are summarized across all samples. Method comparisons use standard measures such as repair rate and average correction time. A simple generalized linear model is used to estimate how repair method and vulnerability type affect the likelihood of full repair:

$$\text{logit}(p_i)=\beta_0+\beta_1 M_i+\beta_2 V_i,$$

where p_i is the probability of full repair, M_i indicates the method, and V_i indicates the vulnerability type.

Improvement over the baseline is measured as:

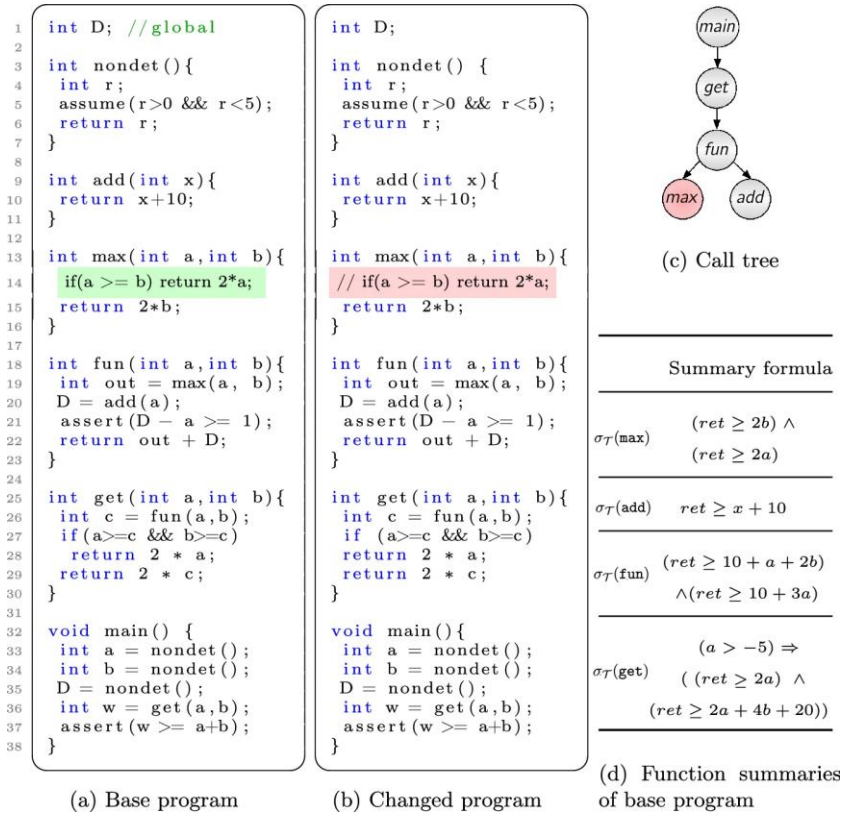
$$I=\frac{S_{method}-S_{baseline}}{S_{baseline}},$$

where S is the repair success rate. All processing is done with fixed random seeds to ensure repeatability.

3. Results and Discussion

3.1 Overall Repair Performance

Across the 3,200 LLM-generated snippets, the static-analysis-guided patch system reaches a full-repair rate of 74% and a partial-repair rate of 92%. A full repair means that every detected violation is removed and all tests pass, while a partial repair means that only some violations are fixed. The LLM-only regeneration baseline produces 51% full repairs and 78% partial repairs under the same conditions. Manual repair by experienced developers achieves 88% full repairs. The proposed method reduces the number of unrepaired snippets by almost half when compared with the LLM-only baseline. This outcome supports earlier findings that solver-based repair methods become more effective when guided by precise static information [18].



(a) Base program (b) Changed program (c) Call tree (d) Function summaries of base program

Fig.1. Full and partial repair rates for the three patching methods in the dataset.

3.2 Performance by Vulnerability Type and Common Failure Cases

Results vary across vulnerability categories. The system performs best on array-bounds and null-pointer issues, with full-repair rates above 80%. These categories align well with the numeric and pointer information tracked by abstract interpretation. In contrast, cases involving missing authentication logic or incorrect resource use reach lower full-repair rates, around 60–65%. In these categories, the analysis can detect the violation, but the available repair rules are not detailed enough to construct full protocol logic, such as multi-step checks or paired open–close sequences. Similar gaps are noted in previous work that uses static analysis mainly for alarm identification, where detection is reliable but repair suggestions are limited [19,20]. Common failure cases in our study include patches that add overly restrictive checks or patches that correct one execution path while leaving another path unchanged. These findings suggest that combining path-sensitive analysis with higher-level specifications may be necessary for more complex logic errors.

3.3 Comparison with LLM-Only and Other Analysis-Supported Methods

We also examine the time developers spend reviewing and adjusting patches. The static-analysis-guided method reduces median manual time by 58% compared with the LLM-only baseline. Many patches require fewer edits because they follow the structure of the detected violations [21,22]. Fig.2 presents this difference in repair throughput and remaining alarms. Unlike dynamic or hybrid systems that rely on program execution, our method works entirely at

the static level. This approach is similar in intent to solver-based repair tools such as Proof2Fix, although those systems focus on large existing codebases rather than short LLM-generated snippets. Compared with these tools, our evaluation places greater emphasis on end-to-end developer effort and the share of snippets that become safe without repeated manual tuning [23].

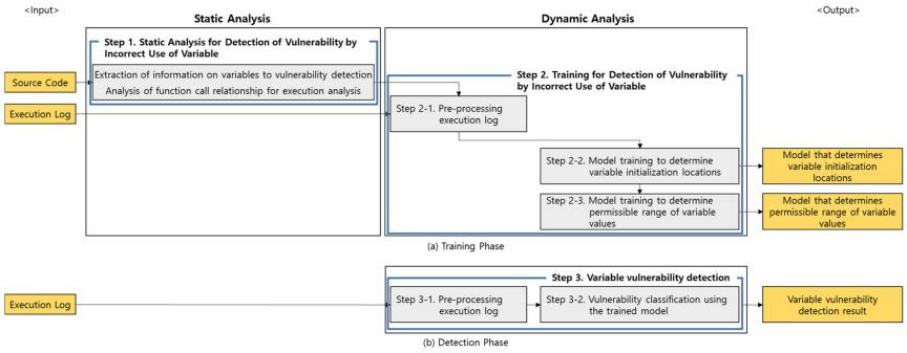


Fig.2. Repair speed and remaining violation counts for the proposed method compared with the LLM-based baseline.

3.4 Implications, Limitations, and Future Work

The results indicate that static analysis can serve as a practical safeguard when LLMs generate insecure code. The method performs especially well when security conditions correspond to standard abstract domains such as bounds and pointer validity. However, there are several limitations. The experiments use short, self-contained snippets with clear specifications; the behavior on large, multi-file systems remains uncertain. The analysis targets a limited set of safety properties and focuses on one programming language family, while real applications involve broader vulnerability types and mixed-language environments. In addition, the repair rules are crafted manually, which restricts coverage for logic-level and protocol-level requirements. Similar constraints appear in earlier static-analysis studies, where issues related to scale, incomplete specifications, and approximation affect practical deployment [24]. Future extensions may include pattern-mined repair templates, support for more languages, and integration with CI pipelines so that analysis and automated patch generation can be applied continuously as LLMs produce new code.

4. Conclusion

This study shows that static analysis can guide patch generation for insecure code produced by large language models. By linking abstract interpretation with solver-based repair rules, the framework creates patches that remove many common security flaws and reduce the amount of manual work required from developers. The method works well for errors related to bounds checks and pointer validity, and it brings machine-generated repairs closer to the quality of human-written fixes. These findings indicate that analysis-guided synthesis can function as a safety layer in coding workflows where LLMs are used to produce code. The approach also has limits. The evaluation is based on short code snippets with clear specifications, and the current

repair rules do not cover more complex logic or protocol steps. Future work may broaden the set of supported properties, expand repair rules, support additional languages, and connect the framework with continuous development pipelines so that patch generation can operate at larger scales.

References

1. Basic, E., & Giaretta, A. (2024). Large Language Models and Code Security: A Systematic Literature Review. arXiv preprint arXiv:2412.15004.

2. Sajadi, A., Le, B., Nguyen, A., Damevski, K., & Chatterjee, P. (2025). Do LLMs consider security? an empirical study on responses to programming questions. Empirical Software Engineering, 30(3), 101.

3. Li, T., Jiang, Y., Hong, E., & Liu, S. (2025). Organizational Development in High-Growth Biopharmaceutical Companies: A Data-Driven Approach to Talent Pipeline and Competency Modeling.

4. Bai, W. (2020, August). Phishing website detection based on machine learning algorithm. In 2020 International Conference on Computing and Data Science (CDS) (pp. 293-298). ieee.

5. Mohsin, A., Janicke, H., Wood, A., Sarker, I. H., Maglaras, L., & Janjua, N. (2024). Can we trust large language models generated code? a framework for in-context learning, security patterns, and code evaluations across diverse llms. arXiv preprint arXiv:2406.12513.

6. Gu, X., Tian, X., Yang, J., & Liu, M. (2025). Building and Performance Validation of a Digital Twin Regulatory Framework for Financial Compliance and Market Transparency. Available at SSRN 5839782.

7. Hasselbring, W., Wojcieszak, M., & Dustdar, S. (2021). Control flow versus data flow in distributed systems integration: Revival of flow-based programming for the industrial internet of things. IEEE Internet Computing, 25(4), 5-12.

8. Qin, F., Cheng, H. Y., Sneeringer, R., Vlachostergiou, M., Acharya, S., Liu, H., ... & Yao, L. (2021, May). ExoForm: Shape memory and self-fusing semi-rigid wearables. In Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems (pp. 1-8).

9. Gomes, D., Felix, E., Aires, F., & Vieira, M. (2025). Static Code Analysis for IoT Security: A Systematic Literature Review. ACM Computing Surveys.

10. Tan, L., Liu, D., Liu, X., Wu, W., & Jiang, H. (2025). Efficient Grey Wolf Optimization: A High-Performance Optimizer with Reduced Memory Usage and Accelerated Convergence.

11. Waduge, C. S. G. (2023). Taming Irregular Control-Flow with Targeted Compiler Transformations (Doctoral dissertation, Purdue University).

12. Bai, W., & Wu, Q. (2023). Towards more effective responsible disclosure for vulnerability research. Proc. of EthiCS.

13. Hojaji, F., Mayerhofer, T., Zamani, B., Hamou-Lhadj, A., & Bousse, E. (2019). Model execution tracing: a systematic mapping study. Software and Systems Modeling.

14. Nian, J., Yang, M., Gao, X., Liu, H., Fang, F., Cheng, L., & Wu, X. (2025). RPFF-PA: Reliable and Parallel Fault-tolerant Framework for Path Latency Reduction Deployed in Register Arrays. ACM Transactions on Embedded Computing Systems.
15. Barchuk, B., & Volkov, K. (2025). Limitations of modern vulnerability scanners and CVE Systems. Authorea Preprints.
16. Sheu, J. B., & Gao, X. Q. (2014). Alliance or no alliance—Bargaining power in competing reverse supply chains. European Journal of Operational Research, 233(2), 313-325.
17. Razzaq, A., Buckley, J., Lai, Q., Yu, T., & Botterweck, G. (2024). A systematic literature review on the influence of enhanced developer experience on developers' productivity: Factors, practices, and recommendations. ACM Computing Surveys, 57(1), 1-46.
18. Du, Y. (2025). Research on Deep Learning Models for Forecasting Cross-Border Trade Demand Driven by Multi-Source Time-Series Data. Journal of Science, Innovation & Social Impact, 1(2), 63-70.
19. Kang, H. J., Aw, K. L., & Lo, D. (2022, May). Detecting false alarms from automatic static analysis tools: How far are we?. In Proceedings of the 44th International Conference on Software Engineering (pp. 698-709).
20. Mao, Y., Ma, X., & Li, J. (2025). Research on API Security Gateway and Data Access Control Model for Multi-Tenant Full-Stack Systems.
21. Bernhard, L., Scharnowski, T., Schloegel, M., Blazytko, T., & Holz, T. (2022, November). JIT-picking: Differential fuzzing of JavaScript engines. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (pp. 351-364).
22. Mao, Y., Ma, X., & Li, J. (2025). Research on Web System Anomaly Detection and Intelligent Operations Based on Log Modeling and Self-Supervised Learning.
23. Sonkin, V., & Tudose, C. (2025). Beyond Snippet Assistance: A Workflow-Centric Framework for End-to-End AI-Driven Code Generation. Computers, 14(3), 94.
24. Liu, S., Feng, H., & Liu, X. (2025). A Study on the Mechanism of Generative Design Tools' Impact on Visual Language Reconstruction: An Interactive Analysis of Semantic Mapping and User Cognition. Authorea Preprints.