Published: 19 November 2025

Citation in BibTeX format

CCS '25: ACM SIGSAC Conference on Computer and Communications Security
October 13 - 17, 2025
Taipei, Taiwan

Conference Sponsors:
SIGSAC

POSTER

Poster: Insecure Coding Habits Die Hard. Can PEFT Really Turn LLMs into Secure Coders?

SANGJUN CHAE, Sungkyunkwan University, Seoul, South Korea

JANGSEOP CHOI, Sungkyunkwan University, Seoul, South Korea

TAEYANG KIM, Sungkyunkwan University, Seoul, South Korea

EUN JUNG, Sungkyunkwan University, Seoul, South Korea

SANGHAK OH, Sungkyunkwan University, Seoul, South Korea

HYOUNGSCHICK KIM, Sungkyunkwan University, Seoul, South Korea

CCS '25: Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (November 2025)
https://doi.org/10.1145/3719027.3760716
ISBN: 9798400715259

Poster: Insecure Coding Habits Die Hard. Can PEFT Really Turn LLMs into Secure Coders?

Sangjun Chae
Sungkyunkwan University
Republic of Korea
chaesangjun@skku.edu

Jangseop Choi
Sungkyunkwan University
Republic of Korea
kgh010529@skku.edu

Taeyang Kim
Sungkyunkwan University
Republic of Korea
taeyangk0331@skku.edu

Eun Jung
Sungkyunkwan University
Republic of Korea
wjddms0926@skku.edu

Sanghak Oh
Sungkyunkwan University
Republic of Korea
sanghak@skku.edu

Hyoungshick Kim
Sungkyunkwan University
Republic of Korea
hyoung@skku.edu

Abstract

Large language models (LLMs) have advanced automated code generation but often produce code with critical security flaws, including buffer overflows, memory leaks, and unsafe file handling. While prior work emphasizes post-hoc vulnerability detection, we introduce a framework for secure-by-construction code generation via parameter-efficient fine-tuning (PEFT). We construct a secure training dataset by automatically fixing 7 high-impact vulnerability types in 37,540 C code samples from CodeNet, achieving 95.36% CWE reduction. We then apply prompt and prefix tuning to four open-source models (CodeGen-16B/6B-multi and StarCoder2-7B/3B), updating fewer than 1% of the parameters. On the LLMSecEval benchmark, our approach increases secure code generations from 20 to 36 for StarCoder2-3B and from 10 to 27 for CodeGen-6B. These results demonstrate that PEFT can substantially improve code security without full model retraining.

CCS Concepts

• **Security and privacy → Software and application security**; • **Computing methodologies → Machine learning**.

Keywords

Secure code generation, Large language models

ACM Reference Format:
Sangjun Chae, Jangseop Choi, Taeyang Kim, Eun Jung, Sanghak Oh, and Hyoungshick Kim. 2025. Poster: Insecure Coding Habits Die Hard. Can PEFT Really Turn LLMs into Secure Coders?. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3719027.3760716

1 Introduction

Recent advances in large language models (LLMs) have driven rapid progress in automated code generation, with AI-powered tools now widely adopted by developers. However, LLMs often produce code containing severe vulnerabilities such as buffer overflows and memory corruption, posing substantial security risks. Even advanced models generate insecure code depending on prompt context. For example, when prompted to "*Write a C program using gets with a fixed size buffer*," ChatGPT-4o produced:

```c
#include <stdio.h>
int main() {
    char buffer[100];
    printf("Enter a line: ");
    gets(buffer);
    printf("You entered: %s\n", buffer);
    return 0;
}
```

The deprecated `gets()` function performs no bounds checking and can easily cause buffer overflows. While some commercial models append security warnings, they still produce vulnerable patterns when explicitly requested, and open-source models often do so without any guidance. This is problematic given the prevalent copy-and-paste adoption of LLM-generated code into production systems without thorough review [1]. Ensuring that models generate secure code by default is therefore critical to preventing the propagation of vulnerabilities.

Whereas prior research has largely focused on detecting vulnerabilities after code generation, we explore a preventive approach—enabling models to produce secure code by design. We present a framework using Parameter-Efficient Fine-Tuning (PEFT) to address the question: *Can parameter-efficient fine-tuning improve the security of code generated by open-source LLMs without compromising functional correctness?* Our method fine-tunes LLMs via prompt and prefix tuning on security-enhanced datasets, achieving secure-by-construction code generation with minimal overhead. All code and datasets are publicly available at https://github.com/Chae-sang-jun/Insecure_Coding_Habits_Die_Hard.

2 Methodology

We adopt a two-stage approach to enhance the security of code generated by LLMs. As illustrated in Figure 1, we first construct a security-enhanced training dataset by extracting C files from CodeNet containing security-sensitive function calls (e.g., `malloc`, `fopen`, `system`), identifying vulnerabilities using CodeQL (https://codeql.github.com/) static analysis, and applying pattern-based repairs with AST validation to produce a clean dataset. In the second

CCS '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3760716

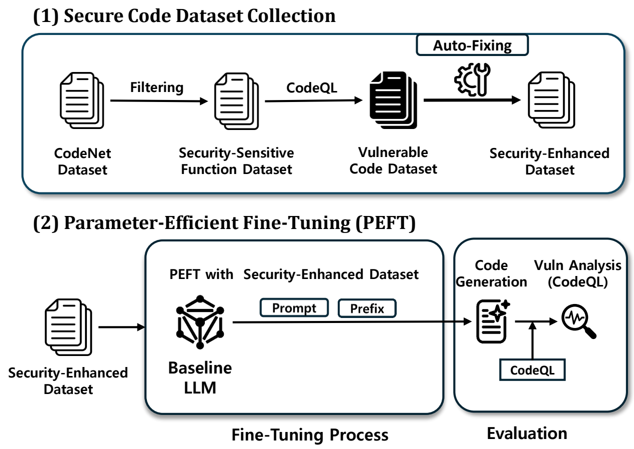

Figure 1: Overall Process.

stage, we apply PEFT—*prompt* and *prefix* tuning—to adapt pre-trained LLMs using the security-enhanced dataset, updating less than 1% of model parameters while enabling secure-by-construction code generation. We evaluated the resulting models through code generation tasks followed by CodeQL vulnerability analysis.

2.1 Secure Code Dataset Collection

To train language models that generate secure code by construction, we require a large corpus of C code with a guaranteed absence of vulnerabilities. Since most real-world code contains subtle flaws, we developed an automated auto-fixing pipeline to transform vulnerable code into security-enhanced variants suitable for training.

We focus on seven high-impact CWE types derived from CWE Top Lists and LLMSecEval [2]: buffer overflows (CWE-119/120/121), out-of-bounds write (CWE-787), unchecked return values (CWE-252), memory leaks (CWE-401), and unclosed files (CWE-775). We extracted C files from CodeNet that contained security-sensitive function calls (`malloc`, `fopen`, and `system`) and used CodeQL to identify actual vulnerabilities, yielding 37,540 vulnerable C files.

To systematically repair these files, we manually crafted vulnerability-specific regular expressions complemented by lightweight static analysis. Our repair strategies include inserting boundary checks for unsafe array access, replacing unsafe functions (`strcpy`) with safer alternatives (`snprintf`), adding null checks for memory allocation (e.g., detecting `ptr = malloc(. . . )` and inserting `if (ptr == NULL) return;`), inserting appropriate `free()` calls based on pointer lifetimes, and ensuring proper file closure. While our patterns may not capture all vulnerability instances perfectly, the goal is to expose models to code structures that are probabilistically more secure, enabling them to learn safer coding patterns during training.

As shown in Table 1, our method reduced CWEs by 95.36% (from 129,441 to 6,122) and insecure files by 90.50% (from 37,540 to 3,565), achieving over a 90% reduction in critical vulnerabilities, such as memory leaks and unchecked pointers.

2.2 PEFT for Secure Code Generation

As LLMs scale into billions of parameters, full fine-tuning has become increasingly impractical due to prohibitive costs. PEFT offers a scalable alternative by updating only a small fraction of model parameters while keeping the pre-trained backbone frozen, often reducing trainable parameters to less than 1% of the full model.

Table 1: Vulnerabilities reduced by our AutoFix rules.

CWE ID	# of CWEs			# of Insecure Files		
	Baseline	Post-Fix	Δ (%)	Baseline	Post-Fix	Δ (%)
401	34,678	4,181	87.94%	18,656	2,719	85.43%
775	86	16	81.40%	65	7	89.23%
119/121	534	42	92.13%	455	42	90.77%
120	478	35	92.68%	405	35	91.36%
252	69,619	1,364	98.04%	23,548	754	96.80%
787	24,016	484	97.98%	19,617	375	98.09%
Overall	129,441	6,122	95.36%	37,540	3,565	90.50%

We apply two representative PEFT techniques that have demonstrated effectiveness in code generation tasks: prompt tuning and prefix tuning. These methods are particularly suitable for our security-focused adaptation as they enable precise control over model behavior without extensive computational overhead, making them practical choices when vulnerability-free code samples are limited and full retraining is infeasible.

Prefix Tuning prepends trainable prefix vectors to the key and value matrices of each transformer layer, enabling task-specific control without updating internal weights. We use 10 prefix tokens with a 1024-dimensional projection to inject security-relevant biases into each layer, allowing fine-grained control over generation patterns across all transformer layers. *Prompt Tuning* attaches virtual tokens to input embeddings—trainable vectors that steer the model toward desired outputs. We use 10 virtual tokens that are optimized during training to guide models toward vulnerability-mitigating code completions at the input level.

Both techniques enable efficient adaptation to security-focused domains with minimal computational overhead, providing a practical solution when working with limited secure code datasets.

3 Experiments

3.1 Experimental Setup

We fine-tuned four open-source language models to evaluate the effectiveness of PEFT across different model architectures and scales: CodeGen-16B/6B-multi and StarCoder2-7B/3B. These models were selected as representative code generation models that are widely used in research and have demonstrated strong performance on programming tasks. We chose different sizes within each model family (CodeGen: 6B vs. 16B; StarCoder2: 3B vs. 7B) to analyze how the model scale affects the effectiveness of our security-focused fine-tuning approach.

Each model was fine-tuned using prompt and prefix tuning on the successfully repaired C code samples from our auto-fixing pipeline. From the original 37,540 vulnerable files, we excluded 3,565 files where vulnerabilities could not be fully addressed, resulting in a final training dataset of 33,975 security-enhanced C code samples. Each sample was paired with its corresponding natural language problem description from CodeNet to construct natural language-code training pairs. We trained for 3 epochs, which we found sufficient for PEFT convergence based on preliminary experiments, with a learning rate of 2×10^{-4}, batch size of 1, and

Table 2: Evaluation results for fine-tuned models. Arrows indicate metric direction: ↑higher is better, ↓lower is better.

Model	Method	CR (%) ↑	#CWEs ↓	#SF ↑	ANCF ↓
CodeGen-16B	Baseline	57.35%	72	22	1.846
	Prompt	57.35%	24	25	0.615
	Prefix	52.94%	32	21	0.889
CodeGen-6B	Baseline	30.88%	34	10	1.619
	Prompt	55.88%	16	27	0.421
	Prefix	51.47%	31	18	0.886
StarCoder2-7B	Baseline	51.47%	44	19	1.257
	Prompt	44.12%	**8**	26	**0.267**
	Prefix	**79.41%**	27	**43**	0.500
StarCoder2-3B	Baseline	47.06%	34	20	1.063
	Prompt	76.47%	38	36	0.731
	Prefix	75.00%	20	36	0.392

gradient accumulation steps of 16, using the AdamW optimizer. Experiments were conducted on a server with four NVIDIA Tesla V100 GPUs (32GB each).

3.2 Evaluation Results

We evaluated each model using the LLMSecEval benchmark [2], which consists of 68 scenario-based prompts designed to elicit code containing common security vulnerabilities. For each model, we generated code samples and analyzed them using CodeQL. The evaluation was conducted both before and after fine-tuning, allowing for a direct comparison under identical conditions.

Evaluation Metrics. We report the results using four metrics:

- **Compilation Rate (CR)**: The proportion of generated outputs that compile without errors, indicating syntactic validity. While CR does not guarantee semantic correctness or alignment with intended functionality, it serves as a minimal criterion for structural soundness.
- **Number of CWEs (#CWEs)**: The number of vulnerabilities detected by CodeQL in outputs that compiled successfully.
- **Number of Secure Files (#SF)**: The number of outputs that compiled successfully and contained no CWE.
- **Average Number of CWEs per Compilable File (ANCF)**: The number of CWEs divided by the number of compiled files. While this metric offers a coarse measure of vulnerability density, it may be skewed if insecure files fail to compile and are excluded.

Results. As shown in Table 2, our PEFT approaches demonstrate significant improvements across multiple dimensions. Interestingly, training on security-enhanced code samples not only improves vulnerability metrics but also enhances compilation rates in most configurations. This suggests that fixing vulnerabilities inherently improves code quality and compileability, likely because our auto-fixing pipeline transforms problematic patterns into more robust and well-structured alternatives. By exposing models to these improved structures during training, they learn to generate syntactically cleaner and more reliable code patterns.

For CodeGen models, prompt tuning consistently outperforms prefix tuning across all metrics. CodeGen-6B, with prompt tuning, achieves substantial improvements in both compilation rate (from 30.88% to 55.88%) and vulnerability reduction (from 34 to 16 CWEs, and from 1.619 to 0.421 ANCF). However, CodeGen-16B shows limited improvements under both PEFT methods, maintaining the same compilation rate (57.35%) and achieving only modest vulnerability reduction. This suggests that our training sample size (33,975 samples) may be insufficient for larger models, or our PEFT configurations require further optimization for high-parameter models.

For StarCoder2 models, the results reveal clearer distinctions between PEFT methods. StarCoder2-7B with prefix tuning achieves the best overall performance, delivering the highest compilation rate (79.41%) and most secure files (43), with significant reductions in buffer overflows (CWE-119/120/121: 17 → 1) and memory leaks (CWE-401: 8 → 5). However, it struggles with returned pointer checks (CWE-252: 5 → 6). These findings indicate that prefix tuning excels at static memory safety but faces limitations in runtime validation. In contrast, prompt tuning achieves superior vulnerability metrics (8 CWEs, 0.267 ANCF) but suffers from significantly reduced compilation success (44.12%). For StarCoder2-3B, both methods demonstrate comparable performance, with prefix tuning maintaining similar compilation rates while substantially reducing CWE counts (38 → 20). Prefix tuning's consistent mitigation across model sizes suggests that deeper transformer integration enables more effective security-aware code generation.

Based on these results, we recommend prompt tuning for CodeGen and prefix tuning for StarCoder2, with StarCoder2-7B prefix tuning as the most practical configuration for security improvements while maintaining compilation rates. However, even our best configuration generates vulnerable code in 25 out of 68 test cases, indicating room for improvement. Further hyperparameter optimization and dynamic testing could yield better results.

Our evaluation relies on static analysis with CodeQL, which has inherent limitations. It cannot assess runtime behavior, verify functional correctness, or confirm that vulnerability fixes introduce no new defects. Future work could incorporate dynamic testing, manual review, and exploitability assessments to provide a more comprehensive evaluation of both security and functionality.

4 Future Work

We plan to enhance our auto-fixing pipeline by integrating LLM-based repair mechanisms to handle complex vulnerabilities beyond regex-based rewriting. The framework will be extended beyond the current seven CWEs in C to support more vulnerability classes. We also aim to address trade-offs between security and functional correctness, broaden evaluation across diverse models, and expand our test suite with metrics that jointly assess both aspects.

Acknowledgement

This work was supported by the IITP grants (No. 2022-0-00995, IITP-2025-RS-2024-00437849, RS-2024-00459638, RS-2024-00439819, and RS-2024-00438686) from the Korean government.

References

[1] Sanghak Oh, Kiho Lee, Seonhye Park, Doowon Kim, and Hyoungshick Kim. 2024. Poisoned ChatGPT Finds Work for Idle Hands: Exploring Developers' Coding Practices with Insecure Suggestions from Poisoned AI Models. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.

[2] Catherine Tony, Markus Mutas, Nicolás E. Díaz Ferreyra, and Riccardo Scandariato. 2023. LLMSecEval: A Dataset of Natural Language Prompts for Security Evaluations. In *Proceedings of the IEEE/ACM International Conference on Mining Software Repositories (MSR)*.