Python Code Verification and Improvement Using Large Language Models

2310402 Jongmin Lee

The explosive growth of artificial intelligence (AI) has propelled large language models (LLMs) from academic curiosities to everyday development companions. By translating natural language descriptions directly into source code, LLMs promise unprecedented boosts in programmer productivity and faster delivery cycles. Yet, this promise is overshadowed by persistent concerns. Empirical studies show that automatically generated code frequently embeds critical security weaknesses, subtle logic flaws, and mismatches between user intent and program behavior. These shortcomings can translate into exploitable vulnerabilities, unexpected production outages, and costly maintenance overhead. Traditional manual reviews cannot keep pace with the volume and velocity of LLM output, underscoring the urgency for scalable, automated safeguards.

This thesis introduces CodeEnhancer, a fully automated, two stage framework that tackles these challenges holistically. Stage 1 combines best practice structured prompting with an iterative “generate-verify-refine” loop. Each candidate program is scrutinized by Pylint for syntax compliance, Bandit for CWE-mapped vulnerability patterns, and an LLM-based reasoning module that cross-checks functional intent captured in a mandatory docstring. Structured, tool-specific feedback is fed back to the LLM until the code passes all checks or a configurable iteration budget is exhausted, yielding a corpus of high quality, self-documented scripts with minimal human oversight.

Stage 2 shifts the focus from fixing problems after the fact to proactively ensuring code quality. In this phase, GPT-4o is fine-tuned using two complementary datasets. Secure code examples authored by experts from the LLMSecEval benchmark, and refined outputs from Stage 1. This combined training approach equips the model with both best practice secure coding techniques and practical remediation strategies, allowing it to recognize and prevent common vulnerabilities during code generation.

Comprehensive experiments on the security-oriented LLMSecEval and SecurityEval benchmarks confirm CodeEnhancer’s effectiveness. A GPT-4o baseline produced vulnerable code in 43.6% of LLMSecEval tasks. A GPT-4o variant further trained with the CodeEnhancer pipeline (framework-tuned GPT-4o) lowered this rate to 18.4% prior to refinement and to only 8% after one refinement cycle. On SecurityEval, the same framework-tuned GPT-4o cut the final vulnerability rate. Functional correctness exhibited similar improvements, with 98.7% prompt adherence after refinement while preserving or enhancing execution efficiency. Importantly, the framework-tuned GPT-4o achieved these gains in fewer iterations than both the baseline and a fine-tuned model by secure code written by experts. Underscoring the synergistic value of CodeEnhancer’s automatically generated training data.

Beyond empirical metrics, CodeEnhancer offers practical benefits. It integrates seamlessly into CI/CD pipelines, requires no language specific test harnesses, and remains agnostic to future advances in static code analysis (SAST) tools. Nevertheless, limitations persist. Current evaluations target Python and static analysis. Extending coverage to languages with stronger type systems and incorporating dynamic or formal constitutes next steps. Additionally, broader datasets that capture domain specific security requirements will help generalize the approach.

In summary, CodeEnhancer demonstrates that pairing LLMs with iterative static analysis, verification, and feedback informed fine-tuning creates a virtuous cycle that progressively decreases vulnerabilities and increases trustworthiness. The framework lays a scalable foundation for safely harnessing AI powered code generation across modern software engineering workflows, paving the way for future research on multi language support, dynamic analysis, and continuous self-improvement loops.

Keywords: Large Language Models (LLMs), Code Generation, Software Security, Static Analysis, Functional Correctness, Fine-Tuning