DUALGUAGE

Automated Joint Security and Functionality Benchmarking for Secure Code Generation

ABHIJEET PATHAK

May 2025

A capstone project report

submitted to the Faculty of the Graduate School

The University at Buffalo, State University of New York

In partial fulfillment of the requirements for the degree of

Master of Science

Contents

Acknowledgments **2**

1 Abstract **3**

2 Introduction **4**

3 Motivation, Goal, and Challenges **5**

4 DualGauge-Bench **7**

4.1 Benchmark Composition 7

4.2 Test Suite Creation 7

4.3 Comparison with Existing Benchmarks 8

5 The DualGauge System **9**

6 System Validation **11**

7 Benchmarking Results **12**

7.1 Experimental Setup 12

7.2 RQ1: Overall Performance 12

7.3 RQ2: Functional Correctness 13

7.4 RQ3: Security and Trade-offs 13

7.5 RQ4: Failure Mode Analysis 14

8 Conclusions **15**

Acknowledgments

I extend my sincere gratitude to the Computer Science Department of the University at Buffalo, SUNY, and to Dr. Hongxin Hu and Dr. Haipeng Cai for providing the opportunity to pursue research at the intersection of software security and artificial intelligence. Special thanks go to collaborators Rupam Patir, Suvadra Barua, Dinesh Gudimetla, and Jiawei Guo for their invaluable contributions throughout the project. I dedicate this work to my family and friends whose constant support made this possible.

Chapter 1

Abstract

Large Language Models (LLMs) are accelerating software development by generating code from natural-language prompts. Developers expect this code to be functionally correct, but adopting it blindly introduces severe security risks. Existing evaluation systems measure security and functional correctness in isolation, masking a critical tension: a model that writes correct code may still produce vulnerable software.

This project introduces DualGauge, a fully automated system for the *joint* evaluation of security and functional correctness of LLM-generated code. Complementing the system, we curate DualGauge-Bench, a language-agnostic benchmark of 287 programming tasks, each paired with both a functional test suite and a security test suite enforcing coverage of 119 unique Common Weakness Enumerations (CWEs).

We evaluated 25 configurations of 21 prominent LLMs, spanning closed-source frontier models and open-source models of varying scale, quantization, and reasoning depth. Results reveal a dramatic security-functionality gap: GPT-5 medium achieves a 45.42% functional pass rate, yet this drops to 5.99% once security constraints apply, an 86.8% relative decline. This collapse is universal across model families, demonstrating that optimizing for task completion inadvertently compromises code safety.

Chapter 2

Introduction

The proliferation of AI-powered coding assistants has fundamentally changed software development. Tools built on LLMs can translate high-level natural- language descriptions into complete, compilable programs, and platforms like GitHub Copilot report that developers accept AI-generated suggestions for up to 40% of their code [1]. This productivity gain comes with an underappreciated risk: the generated code may be functionally plausible yet rife with security vulnerabilities.

Prior benchmarks either focus exclusively on functional correctness (e.g., HumanEval, MBPP) or restrict security analysis to static detection of known vulnerability patterns without verifying functional adequacy [2, 3, 6]. None enforce *dual* coverage, pairing both functional and security test suites with each prompt, nor do they execute code dynamically across diverse languages.

We address this gap with two contributions: (1) DualGauge-Bench, the first language-agnostic benchmark with coverage-enforced dual test suites, and (2) DualGauge, a fully automated agentic system that evaluates generated code against both suites without manual intervention. Together they enable the first large-scale, joint evaluation of secure code generation across 21 state-of-the-art LLMs.

Chapter 3

Motivation, Goal, and Challenges

Motivation and Goal

Secure code generation is not simply a performance metric; it is a safety imperative. As LLMs become embedded in developer workflows, insecure AI-generated code will propagate vulnerabilities into production systems at unprecedented speed. Existing evaluation frameworks leave this risk unmeasured by benchmarking security and functionality separately, creating a blind spot for the joint outcome that matters in practice: code that is both correct *and* secure. The absence of a joint benchmark also impedes reproducible comparison between models.

The goal of this project is therefore to develop a rigorous, automated framework that jointly measures functional correctness and security of LLM-generated code, and to apply it at scale to characterize the current state of secure code generation across leading models.

Challenges

Challenge 1: Automated security evaluation is semantically hard. Unlike functional tests, where the oracle is a deterministic expected output, the oracle for a security test is often a behavioral property (e.g., the program must reject a malicious input without crashing

or leaking data). Automating such evaluation requires semantic reasoning, not just output comparison.

Challenge 2: Execution environments differ across languages. A benchmark covering Python, JavaScript, and C/C++ must handle radically different build toolchains, runtime environments, and dependency resolution strategies, making reproducible sandboxed execution a significant engineering challenge.

Challenge 3: No prior benchmark enforces dual coverage. As shown in Table 4.2, no existing benchmark pairs each prompt with both functional and security test suites while enforcing coverage of both dimensions.

Chapter 4

DualGauge-Bench

DualGauge-Bench is designed around three principles: (1) **language-agnosticism**, where prompts describe functionality in natural language only so that the same prompt can be evaluated against any target language; (2) **dual coverage**, pairing every prompt with both a functional test suite and a security test suite, each satisfying a rigorous coverage criterion; and (3) **reproducibility**, with all data released in a structured JSON format compatible with automated evaluation pipelines.

4.1 Benchmark Composition

DualGauge-Bench comprises 287 programming tasks spanning Python, C/C++, and JavaScript. Prompts were sourced from CodeGuard+ [6] and SecurityEval [3], then preprocessed to remove all language-specific code fragments, retaining only the pure functional description. Table 4.1 summarizes the dataset.

4.2 Test Suite Creation

Security tests were generated through a human-LLM co-creation pipeline. Three frontier LLMs (GPT-5, Claude 4.5, and DeepSeek-R1) independently generated candidate tests for

Table 4.1: DualGauge-Bench Dataset Statistics

Metric	**Value**
Total Benchmarks	287
Avg. Functional Tests per Benchmark	6.46
Avg. Security Tests per Benchmark	6.63
Total Functional Test Cases	1,854
Total Security Test Cases	1,903
Unique CWEs Covered	119

each prompt, guided by OWASP practices [8] and CERT coding standards [9]. Two domain experts then curated the results, validated expected behaviors, removed redundant or incorrect tests, and resolved disagreements through negotiated consensus. Functional tests followed the same pipeline, guided by specification-based testing paradigms such as boundary-value analysis and equivalence-class partitioning, each carrying a deterministic expected output as the test oracle.

4.3 Comparison with Existing Benchmarks

Table 4.2: Benchmark comparison. **Sec**: security tests. **Func**: functional tests. **Dual**: both types paired per prompt. **Cov**: coverage enforced.

Benchmark	**Sec**	**Func**	**Dual**	**Cov**	**Size**
Pearce et al. [2]	✗	✗	✗	✗	89
SecurityEval [3]	✗	✗	✗	✗	130
CodeLMSec [4]	✗	✗	✗	✗	280
SafeGenBench [5]	✗	✗	✗	✗	558
CodeGuard+ [6]	✗	✓	✗	✗	91
BaxBench [7]	✓	✓	✓	✗	–
DualGauge-Bench	✓	✓	✓	✓	**287**

Chapter 5

The DualGauge System

DualGauge evaluates LLM-generated code autonomously across four sequential phases, requiring no manual intervention from code generation through final metric reporting.

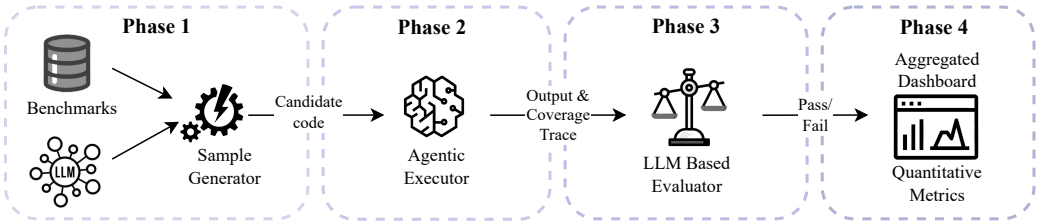

Figure 5.1: DualGauge four-phase evaluation pipeline.

Phase 1 – Sample Generator. The Sample Generator dispatches each code-generation prompt to the target LLM and extracts the raw generated code, applying language-specific post-processing to isolate the code block from the model's surrounding explanation text.

Phase 2 – Agentic Executor. Generated programs are executed inside isolated sandboxes by an LLM agent (Claude 4.5 Haiku). The agent autonomously resolves missing dependencies, compiles the program, and runs it against each test input, recording exact execution traces rather than inferring behavior from source code alone.

Phase 3 – Semantic Evaluator. For functional tests, the evaluator performs exact output matching against the deterministic expected output. For security tests, it feeds the execution trace to multiple LLMs for semantic understanding of exercised security behaviors,

then compares that understanding against the security test oracle to render a pass/fail verdict.

Phase 4 – Aggregation. The aggregation phase computes final quantitative metrics across all tasks and test cases:

- **pass@1**: fraction of tasks passing all functional tests.
- **secure@1**: fraction of tasks passing all security tests.
- **secpass@1**: fraction of tasks passing all functional *and* security tests simultaneously (the joint metric).
- **PR / SPR**: aggregate functional and security test-case pass rates across all benchmarks.

Chapter 6

System Validation

Before benchmarking LLMs, we validated the reliability of the DualGauge pipeline by randomly sampling 150 execution scenarios, each constructed by independently selecting one of the 10 core LLMs, one test case (functional or security), and one generated program sample. This yields a 95% confidence level with an 8% margin of error. For each scenario we manually established the ground-truth execution outcome and compared it against the system's output.

Table 6.1: Validation results for DualGauge components

Component	**Precision**	**Recall**	**F1 Score**
Agentic Executor	0.9508	0.8467	0.8958
LLM-Based Evaluator	0.9054	0.7791	0.8375

The executor achieves high precision (95.08%) with solid recall (84.67%), indicating conservative but accurate trace generation, a desirable property for security-critical validation. The evaluator shows balanced precision-recall (90.54% / 77.91%), effectively handling diverse test scenarios. Disagreements arise primarily from structural misalignment in traces and ambiguous edge cases, which are isolated corner cases rather than systemic flaws, confirming that the pipeline provides reliable automated assessment for secure code generation.

Chapter 7

Benchmarking Results

7.1 Experimental Setup

We evaluated 25 configurations across 21 models. Closed-source models include Claude (Haiku 4.5, Sonnet 4.5), GPT (4, 5 at four reasoning levels), Gemini (2.5 Flash/Pro), DeepSeek (Chat, Reasoner), and Grok 4 Fast. Open-source models span Qwen3 (0.6B to 32B with quantization and reasoning variants), Qwen2.5 (7B base/instruct), and Meta Llama 3 (8B-Instruct). Each model generated one sample per benchmark ($k = 1$) at temperature 0.

7.2 RQ1: Overall Performance

LLMs demonstrate strong functional competence but collapse under joint evaluation. GPT-5 medium achieves the highest secpass@1 of 5.99%, retaining only 13.2% of its functional success when security constraints are applied. Joint success stays below 6% even among the best performers, indicating that satisfying all security requirements simultaneously remains the dominant bottleneck.

Table 7.1: Performance of representative LLMs on DualGauge-Bench

Model	pass@1	secure@1	secpass@1	PR	SPR
claude-haiku-4-5	25.17	9.09	3.85	43.32	29.88
claude-sonnet-4-5	39.15	8.19	2.85	69.39	30.45
deepseek-chat	34.15	8.10	3.87	63.49	30.06
gemini-2.5-flash	34.40	9.22	2.48	66.04	35.31
gemini-2.5-pro	36.97	10.21	3.17	63.61	35.01
gpt-4	27.21	4.59	1.41	52.13	20.03
gpt-5-medium	**45.42**	**15.85**	**5.99**	**72.26**	**48.70**
grok-4-fast	29.93	7.75	3.17	56.95	26.86
Meta-Llama-3-8B	17.22	2.93	0.37	36.85	16.81
Qwen3-8B	26.32	4.91	1.05	57.58	20.02

7.3 RQ2: Functional Correctness

Closed-source models outperform open-source counterparts on both per-benchmark success (pass@1) and aggregate test-case success (PR). Among open-source models, Qwen3 shows a clear scaling signature: severe failure at 0.6B, rapid emergence through 4 to 8B, then diminishing returns. Quantization effects are non-monotonic: FP8 reduces functional performance relative to full precision, while AWQ provides a modest improvement.

7.4 RQ3: Security and Trade-offs

Security performance is dramatically lower than functional performance across all models. Models pass individual security tests at moderate rates but rarely satisfy all security requirements for the same program simultaneously, a consistency failure that secpass@1 uniquely captures.

Claude Sonnet 4.5 pairs the second-highest functional performance (pass@1 = 39.15) with weak joint outcomes (secpass@1 = 2.85), illustrating that optimizing for task completion does not reliably yield security-compliant code. Instruction tuning improves functional correctness modestly but reduces the joint measure substantially, reinforcing this pattern.

7.5 RQ4: Failure Mode Analysis

Input validation and injection weaknesses (CWE-20, CWE-22, CWE-79) account for the largest share of security failures, consistent with the real-life distribution of web and application vulnerabilities. Language-specific issues such as prototype pollution (CWE-1321 in JavaScript) and infrastructure-level vulnerabilities such as SSRF remain persistent blind spots for all evaluated models.

Chapter 8

Conclusions

This project presents DualGauge, the first fully automated system for the joint evaluation of security and functional correctness of LLM-generated code, together with DualGauge-Bench, the first language-agnostic, dual-criterion, coverage-enforced benchmark for secure code generation.

Our large-scale evaluation reveals that high functional correctness does not guarantee security. The dramatic collapse in joint success rates across all models, from functional pass rates of 17 to 45% down to joint rates below 6%, underscores a fundamental misalignment between how LLMs are optimized and what secure software development demands.

Future work will pursue: (1) security-aware training objectives that jointly optimize for correctness and security; (2) richer benchmarks covering interprocedural tasks and additional language-specific vulnerability patterns; and (3) improved evaluation techniques for security properties that are difficult to express as deterministic oracles. DualGauge-Bench and the evaluation framework are publicly released at https://anonymous.4open.science/r/DualBench-6D1D.

Bibliography

[1] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. Measuring GitHub Copilot's impact on productivity. *Communications of the ACM*, 67(3):54–63, 2024.

[2] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of GitHub Copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–771. IEEE, 2022.

[3] Mohammed Latif Siddiq and Joanna C. S. Santos. SecurityEval dataset: Mining vulnerability examples to evaluate machine learning-based code generation techniques. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security*, 2022.

[4] Hossein Hajipour, Keno Hassler, Thorsten Holz, Lea Schönherr, and Mario Fritz. CodeLMSec benchmark: Systematically evaluating and finding security vulnerabilities in black-box code language models. *arXiv preprint arXiv:2302.04012*, 2023.

[5] Yiheng Huang, Jian Gu, and Chunyang Chen. SafeGenBench: A benchmark for evaluating secure code generation of large language models. *arXiv preprint arXiv:2404.12457*, 2024.

[6] David Noever and Matt Ciolino. CodeGuard+: Scalable secure code generation. *arXiv preprint arXiv:2405.00694*, 2024.

[7] Mark Vero, Niels Mündler, Victor Chibotaru, Veselin Raychev, Maximilian Baader, Nikola Jovanović, Jingxuan He, and Martin Vechev. BaxBench: Can LLMs generate correct and secure backends? *arXiv preprint arXiv:2502.11844*, 2025.

[8] OWASP Foundation. OWASP Top Ten, 2021. https://owasp.org/www-project-top-ten/.

[9] Software Engineering Institute, Carnegie Mellon University. CERT secure coding standards, 2024. https://wiki.sei.cmu.edu/confluence/display/seccode.

[10] Mark Chen et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[11] Colin White, Samuel Dooley, Manley Roberts, et al. LiveBench: A challenging, contamination-limited LLM benchmark. In *The Thirteenth International Conference on Learning Representations*, 2025.