Catching Vulnerabilities in AI Generated Code at Scale

Kalit Inani Keshav Kabra Ziyi Yang

Georgia Institute of Technology, School of Computer Science

CS8803 SAI

Problem Statement

- Developers are integrating **AI-generated code** (Claude Code, Github Copilot) into production systems at an unprecedented scale that **traditional testing tools cannot match**
- Safety-critical applications (like algorithmic trading platforms, flight software) risk deploying code that is functionally correct but **algorithmically flawed**, leading to timeouts, memory exhaustion, and crashes
- Current fuzzers rely on **uniform resource allocation** (T_{fixed}), blindly treating all testing targets as equally risky
- This approach wastes cycles on safe code while failing to **catch deep algorithmic complexity bugs** (e.g., $O(n^2)$ logic) in high-risk targets, making exhaustive testing prohibitively expensive

Background & Related Work

- LLM-Based Unit Test Generation (ChatUniTest, FSE'24)
  - Employs LLMs with adaptive context and validation-repair mechanisms
  - Generated tests often violate functional correctness requirements (invalid)
  - *Our experiments*: 21% invalid tests; 0-40% validity on complex problems
  - Limited applicability in catching timeouts, overflows, and crashes
- Formal Verification (Dafny)
  - Requires domain experts to specify invariants and handle iterative refinement
  - LLMs struggle with generating correct loop invariants (50-60% success rate)
  - Static analysis and SMT-based verifiers cannot capture algorithmic complexity mismatches (e.g., O(n²) when n=10⁶)
  - Impractical at scale: requires 5-10+ verification attempts per program with manual intervention when verification fails
- LLM-Guided Fuzzing
  - **OSS-Fuzz-Gen**: Uses LLMs to auto-generate fuzz harnesses via agent-based exploration of projects, build scripts, and validation
    → *Limitation*: Generates generic harnesses; ignores problem-specific constraints, requires fixed fuzzing time without prioritization
  - **Green Fuzzing** (ISSTA): Addresses when to stop fuzzing campaigns to save resources; uses ML on static features to predict vulnerable functions; stops when coverage of predicted vulnerable code saturates
    → *Limitation*: Relies on static features, not semantic reasoning; blind to algorithmic complexity

Opportunities and Challenges

- **Resource Constraints**: Exhaustive testing with fixed timeouts leads to inefficient resource usage. Requires long fuzzing campaigns for non-critical code.
  → *Opportunity*: Intelligent prioritization of high-risk code enables scalable, focused testing.
- **The Semantic Gap**: Static metrics (LOC, Cyclomatic, Halstead) provide limited vulnerability discrimination and are incomplete.
  → *Evidence*: They fail to capture **algorithmic complexity mismatches** (e.g., $O(n^2)$ logic vs. $O(n)$ constraints).
- **The Oracle Challenge**: Current agents often generate generic harnesses and do not capture the oracle constraints, leading to explosion in search space.
- **Prompt Sensitivity**: Semantically equivalent prompts yield different code behaviors.
  → *Opportunity*: System must account for prompt variation to ensure robust verification.

Proposed Solution

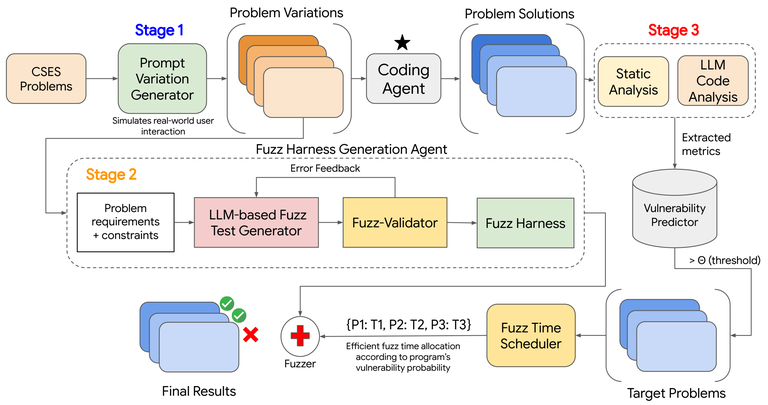

Figure 1. Overview of our proposed architecture

Stage 1: Prompt Variation Generation

Generate variations per CSES problem (e.g., overflow emphasis, reordered presentation, examples-only, specific template requirement) to simulate diverse user interactions with coding agents.

Stage 2: LLM-Based Fuzz Harness Generation

- LLM agent extracts input constraints and generates exhaustive fuzz harnesses
- Generate user-defined oracles: timeout, crash, determinism, overflow detection
- Validator iteratively refines harnesses with compilation error feedback

Stage 3: Vulnerability Prediction & Adaptive Allocation

3.1 Vulnerability Prediction: In addition to static analysis metrics, an LLM agent acts as a complexity judge. Its fed in the problem constraints and code under test (CUT). The agent analyzes and extracts semantic scores similar to how a human would comprehend code. A trained classifier uses these features to predict vulnerability probability $p_i \in [0, 1]$; achieves 0.943 ROC-AUC.

- Static analysis: LOC, cyclomatic and cognitive complexity, Halstead metrics
- **LLM-based features**: Risks of timeouts, logic errors, integer overflows, array out of bounds, null pointer access and handling edge cases.

3.2 Adaptive Allocation: Filter-out CUTs with $p_i < \theta$ (where θ is the vulnerability threshold); The scheduler allocates user-defined time budget (T_{budget}) proportionally among the remaining CUTs:

$$t_i = \frac{p_i}{\sum_{j \in S} p_j} \cdot T_{\text{budget}}, \text{ where } S = \{j : p_j \geq \theta\}$$

3.3 Early-Stopping Scheduling: The scheduler dynamically monitors coverage of running fuzzer. If coverage becomes stagnant within a pre-defined time window, the fuzz test is preempted and the next test is scheduled.

Implementation details:
• *Dataset*: 96 CSES algorithmic problems (sorting, searching, dynamic programming, graph algorithms, mathematics) with official test suites.
• *Eval*: Human-written time/space-optimal reference solutions for evaluation
• *Coding Agent*: Qwen/Qwen3-Coder-480B-A35B-Instruct-Turbo
• *Java Fuzzer*: Jazzer • *ML Model*: Random Forest Classifier

Evaluation Results

Model Discrimination Quality

Our hybrid model better identifies vulnerable code, enabling more aggressive filtering with higher precision:

- GreenFuzz (static-only): Filters 113 (88 clean, 25 buggy) → 77.9% precision
- **Ours**: Filters 231 (198 clean, 33 buggy) → **85.7% precision**
- **Filters 2× more code while missing only 8 additional bugs**
- Better discrimination → more fuzzing resources allocated to high-risk targets

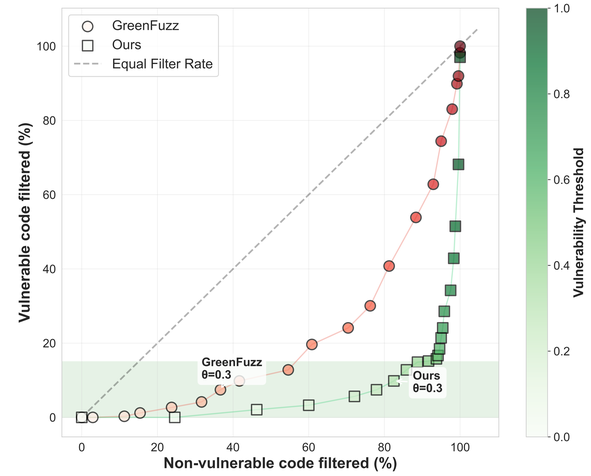

Figure 2. Comparison of our vulnerability predictor vs GreenFuzz baseline

End-to-End Bug Detection Performance

Approach	Bugs Found (out of 336)	Time (mins)	Recall	Accuracy
ChatUniTest	163	68.6	48.6%	63.0%
Fuzzing (Fixed)	**245**	336.5	**72.92%**	**82.12%**
GreenFuzz	212	242.9	63.1%	76.74%
Ours	228	174.9	67.86%	80.38%
Ours + Early Stopping	216	**141.3**	64.29%	78.30%

Key Findings:

- ChatUniTest cannot catch timeout and overflow bugs—unit tests fail to stress-test algorithmic complexity mismatches
- Fixed fuzzing is thorough but exhaustive without intelligent prioritization
- **Adaptive resource allocation** enables substantial time savings while maintaining comparable bug detection
- Our hybrid approach detects **similar bugs (~65% recall) as fuzzing baselines with significantly less time (1.3x - 1.7x faster)**

Future Work

- We plan to extend the framework scope from algorithmic vulnerabilities to more generic software engineering CVEs (e.g. SQL injection and auth issues)
- Fuzz-testing is not effective in verifying functional correctness of programs. We intend to draw ideas from unit-test generation and explore how they can be complemented with LLM-guided fuzz-testing