Catching Vulnerabilities in AI Generated Code at Scale

Kalit Inani
Georgia Institute of Technology

Keshav Kabra
Georgia Institute of Technology

Ziyi Yang
Georgia Institute of Technology

Vima Gupta
Georgia Institute of Technology

Anand Iyer
Georgia Institute of Technology

Abstract

With the increase in popularity of AI-coding assistant tools, developers are increasingly checking-in AI generated code into production systems. Yet, current testing frameworks cannot keep up with this pace. Traditional fuzz testing techniques allocate resources uniformly and lack semantic awareness of potential vulnerable targets. Hence, they waste resources testing safe code and miss to catch actual vulnerabilities efficiently. Further, they ignore prompt constraints in harness generation exploding the search space. To address these issues, we present a hybrid testing framework that employs LLM-guided adaptive fuzzing to efficiently detect vulnerabilities. It combines prompt-based behavioral diversification, LLM-guided fuzz harness generation with problem-specific oracles, and a LLM-based vulnerability predictor to enable adaptive resource allocation and dynamic early stopping. We evaluate our system on a set of CSES algorithmic problems. Our approach improves vulnerability discrimination precision from 77.9% to 85.7% compared to GreenFuzz. It filters out nearly 2x more non-vulnerable code, while achieving comparable bug detection recall at 1.7x lower time cost. These results show a scalable path towards resource-efficient fuzz testing to catch vulnerabilities in AI-generated code.

1 Introduction

Large Language Models (LLMs) have become ubiquitous across diverse domains, from question answering and retrieval systems to performing scientific discoveries. One of the most popular applications has been in software development, where LLM-powered coding agents such as Claude Code [3] and GitHub Copilot [13] are changing how developers write code. These tools allow developers to express intent in natural language and thus enable rapid experiments with product features. As they become popular, the amount of AI-generated code is increasing in production systems across several safety-critical domains.

However, current testing approaches have not evolved to match the pace and scale of the code output by these agents. Traditional testing approaches like unit test generation, formal verification, fuzz testing were built to verify human-written code. However, AI-generated code can show unexpected behaviors, such as algorithmic complexity mismatches, resource exhaustion vulnerabilities, and prompt-induced behavioral variations. As a result, shipping untested AI-generated code in safety-critical systems such as algorithmic trading platforms, flight control software, and medical devices raises risks where seemingly correct programs may fail under adversarial input.

Existing testing approaches have several limitations. LLM-based unit test generation [8] achieves high code coverage but cannot stress-test algorithmic complexity or detect timeout and overflow behaviors. Fuzz Testing [5, 21], a popular technique to identify crashes and security vulnerabilities, is exhaustive in its approach but computationally expensive. Traditionally, it allocates uniform fuzz time budgets to all programs regardless of risk, wasting resources testing safe code while ignoring potentially vulnerable targets. Formal verification [18] provides strong correctness guarantees. However, it requires domain experts for invariant generation as LLMs struggles with increasing code complexity and do not scale. This creates a gap in scale testing AI generated code for algorithmic vulnerabilities.

We address this gap with a hybrid LLM-guided fuzzing framework that combines semantic reasoning with adaptive resource allocation. Our insight is that LLMs can act as semantic analyzers capable of predicting vulnerability risk by reasoning about algorithmic properties that static analysis cannot capture. Our system integrates three components: (1) prompt variant generation to explore user behavioral diversity, (2) LLM-guided fuzz harness generation that extracts problem-specific constraints and generate effective oracles, and (3) a hybrid vulnerability predictor that combines static code metrics with LLM-extracted semantic features to enable intelligent time allocation and early stopping.

Our evaluation of 96 CSES algorithmic tasks [17] shows that our approach improves the precision of vulnerability discrimination from 77.9% to 85.7% compared to GreenFuzz [19], and improves resource allocation by reducing over-

all fuzzing time up to 1.7x while maintaining strong recall. Further, our fuzz-harness generation agent outputs effective harnesses and scales linearly with problems. These results show that LLM-guided semantic analysis enables a scalable path toward efficient vulnerability detection.

2 Background and Motivation

This section examines the current landscape of testing approaches for AI-generated code and identifies key limitations that motivate our work. We analyze three main categories: LLM-based unit test generation, formal verification approaches, and existing fuzzing techniques.

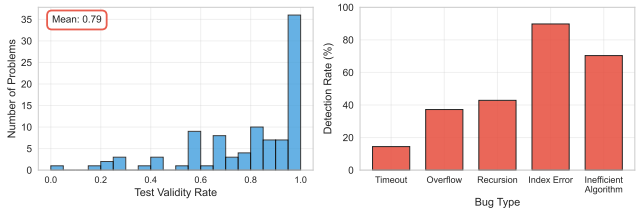

Figure 1: Limitations of ChatUniTest. Left: Validity rate of the generated Unit tests across the problems. Right: Bug Detection rate across the different bug buckets.

2.1 LLM-Based Unit Test Generation

Unit tests are widely used to evaluate the functional correctness of code in software engineering. Recently, LLM-based unit test generation tools such as ChatUniTest [8] have become popular. These tools can often generate both test prefixes and test oracles and ensure high coverage of the unit under test. Specifically, ChatUniTest introduces a generation-validation-repair pipeline, which includes a feedback loop that iteratively optimizes generated test cases, making sure they are compilable and error-free during runtime. ChatUniTest also implements adaptive focal context generation, which extracts only the most relevant code context to minimize the number of LLM tokens. Overall, ChatUniTest achieves excellent line coverage, but it also often produces invalid tests and fails for complex programs. In our evaluation in Figure 1, we see an average 21% invalid tests generated per problem (left figure). By invalid, we mean that the expected output of a function (asserted in its unit test) does not match the actual output for the given input. Moreover, these tests fail to capture timeouts, memory overflows, or algorithmic complexity mismatches (right figure), making it difficult to detect resource-bounded vulnerabilities.

2.2 Formal Verification and SMT-based Analysis

Formal verification mathematically proves the program correctness by checking if it adheres to the provided formal specification. Tools like Dafny [18] leverage SMT solvers to automatically verify programs annotated with preconditions, postconditions, and loop invariants. Overall, it is the strongest way to prove the correctness and is complete.

However, formally verifying AI-generated code at scale is challenging. It requires precise specifications provided by domain experts. Recent work explored using LLMs to generate loop invariants [1, 29], but LLMs struggle with good invariant generation and repairing incorrect invariants. Verification also requires manual effort and iterative refinement upon failure and is computationally expensive, requiring minutes per program. This makes it infeasible to scale for frequently verifying AI-generated code.

2.3 LLM-Guided Fuzzing

Fuzzing [21] is a popular testing technique that checks programs with randomly generated inputs to discover security vulnerabilities. Traditional fuzzing approaches target programs from their public entry points, but struggle with low coverage on deeply nested code paths. Coverage-guided gray-box fuzzers like AFL [11] address this by using code coverage feedback to guide input mutation, but still face challenges reaching functions deep in the call graph.

Fuzzing relies on good harnesses for targeted testing. There have been recent improvements using LLMs to automate fuzz harness generation. OSS-Fuzz-Gen [20] uses LLMs in a multi-agent system to automatically generate fuzz harnesses for target functions. Their system comprises of harness generation, refinement, coverage analysis, and feedback loops to iteratively improve harnesses. While this approach reduces manual effort, the generated harnesses can report false positive crashes [2] and fail to account for the problem specific constraints. This leads to exploding the input fuzzing search space and the detection of general crashes. However, for our use case involving algorithmic code, we want to focus on detecting complexity violations and resource exhaustion.

Another problem with traditional fuzzing is resource allocation and determining the stopping criterion. GreenFuzz [19] tries to address this by using machine learning to predict vulnerable functions and stop fuzzing when the coverage of the predicted vulnerable code saturates. They extract features from static analysis tools and software metrics and use the trained classifier to predict vulnerability probability and filter programs. In their evaluation, they are able to terminate campaigns 6-12 hours earlier and miss fewer than 0.5 bugs on average. Their vulnerability predictor achieved ROC-AUC scores of 0.8 (0.827 in evaluation on our dataset). However, GreenFuzz's approach has several limitations for our domain. It relies solely on static features and cannot capture algorithmic properties. From Figure 4 of our evaluation, we can see that there is still a gap in the quality of vulnerability discrimination of Green Fuzz. It is not able to filter out most of the non-vulnerable targets at lower thresholds. As a result, resources are still spent on testing safe code. Moreover, it uses a fixed saturation window and does not adapt fuzzing time

based on vulnerability probability, allocating equal time to all targets that pass the vulnerability threshold.

3 System Design

To address the limitations identified in the previous work and efficiently detect vulnerabilities in AI-generated code, our system employs a three-stage pipeline architecture that utilizes intelligent resource allocation for fuzz testing. The framework combines LLM-guided semantic analysis with traditional fuzzing to achieve high precision and scalability.

3.1 System Overview

The system takes the natural language descriptions of a set of programming problems as input and assesses the coding agent-generated solutions for vulnerabilities. The pipeline consists of three main stages: (1) Prompt Variant Generation creates diverse formulations of each problem to simulate real-world usage patterns, (2) LLM-Based Fuzz Harness Generation produces problem-specific test harnesses with semantic oracles, (3) Vulnerability Prediction and Adaptive Allocation uses a hybrid ML model to predict vulnerability risk and intelligently distribute fuzzing resources.

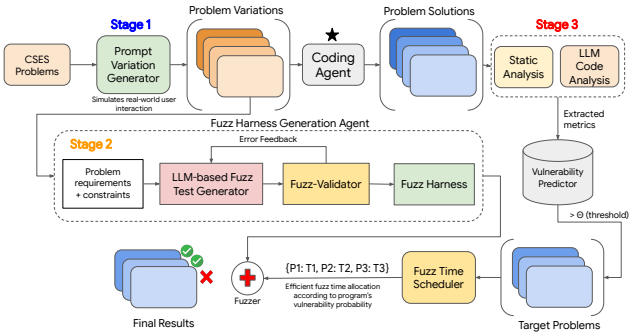

Figure 2: Overview of our proposed architecture

3.2 Prompt Variant Generation

Previous research has shown that semantically equivalent prompts can trigger different code generation behaviors in LLMs [7,23,26], which leads to solutions with different types of vulnerabilities. To explore this finding and simulate diverse real-world user interactions, we generated multiple prompt variants for each problem.

Our variant generation strategy is detailed in Table 1. For each variant, we preserve the core problem semantics while prompting with various styles, which instructs coding agents to produce different solutions. Each solution is evaluated against the official CSES test suite to mark whether it is buggy. The buggy rate in the table shows that prompting the same problem in different ways leads to variation in vulnerabilities.

Table 1: Prompt variants generated for each problem. Each problem generates 12 variants: 1 original, 5 semantic variations, and 6 buggy variations with intentionally injected vulnerability instructions.

Variant Name	Description	% Buggy
Original & Semantic Variations		
Original	Original problem format	29.17
Overflow Emphasis	Highlights edge cases with large numbers	27.08
Reordered Presentation	Reorders constraints and examples first	29.17
Examples Only	Minimal format relying on examples	35.42
Iterative Approach	Explicitly requests iterative loops	29.17
Edge Case Focus	Emphasizes boundary value testing	28.12
Buggy Variations (Intentional Vulnerabilities)		
Integer Overflow	Forces int instead of long data types	62.50
Timeout/Inefficient	Suggests inefficient algorithms	93.75
Heavy Recursion	Encourages deep recursion patterns	94.79
Array Indexing	Induces off-by-one errors	91.67
Greedy Implementation	Suggests wrong algorithmic approach	92.71
Incorrect Logic	Introduces subtle logical errors	86.46

3.3 LLM-Based Fuzz Harness Generation

3.3.1 Harness Generator Agent Design

Our fuzz harness generator employs structured prompt engineering to guide an LLM agent in generating Jazzer [9] fuzz tests. The generator receives three inputs: the problem description, the solution code, and the extracted `solve()` target function signature. The LLM performs semantic analysis to understand problem constraints, algorithmic properties, and potential failure scenarios.

Constraint Extraction: The LLM parses problem descriptions to extract numerical bounds (e.g., $1 \leq n \leq 10^6$), type requirements, and structural constraints. These constraints guide input generation to remain within valid ranges and avoid exploding the search space. For example, in a graph problems with n nodes and m edges, the generator respects these bounds during edge creation.

Weighted Input Generation: Rather than uniform random sampling, we employ weighted distributions to bias toward stress-inducing inputs. For recursion-heavy problems (DFS, backtracking), we bias toward maximum recursion depth by generating large grids. For memory-intensive problems (involving dynamic programming), we bias toward maximum input sizes. The generator creates inputs spanning boundary

values at constraint limits, edge cases like empty arrays or single elements, patterns like all-identical elements, and overflow triggers using maximum representable values.

3.3.2 Oracle Generation

Generic fuzz harnesses often generate inputs without considering problem constraints, leading to a large search space that wastes resources on invalid inputs. We address this by generating problem-specific oracles according to constraints from the problem description. These oracles focus fuzzing on valid inputs and detect violations of expected program behavior.

The timeout oracle detects infinite loops and excessive recursion by running the solution in a separate thread with timeout monitoring. The crash oracle identifies errors such as null pointer dereferencing and array index violations. The determinism oracle catches code with uninitialized memory and ensures that identical inputs lead to identical outputs. Finally, the overflow oracle is responsible for looking for incorrect type choices. For example, operations on large positive numbers that yield a negative value can indicate overflow issues.

3.3.3 Validation Loop

LLM-generated code can contain syntax errors, type mismatches, or incorrect API usage. Drawing inspiration from OSS-Fuzz-Gen [20], we implement a two-stage validation loop with compilation-driven feedback to improve harness quality.

When fuzzing on the generated harness, the agent attempts to compile with the dependencies of the fuzzing framework. The compilation checker captures error messages and parses them to identify common issues like missing dependencies, incompatibility between harness and target function, inappropriate usage of fuzzer API, etc. In case of failures, it uses a retry mechanism that includes the original problem, the previously generated harness code, and the compilation error output. This feedback guides the LLM perform targeted fixes rather than regenerating from scratch.

3.4 Vulnerability Prediction and Adaptive Allocation

Traditional fuzzing allocates uniform time budgets to all targets, wasting resources on safe code while missing complex vulnerabilities in high-risk programs. Our approach addresses this through ML-based risk prediction and proportional resource allocation. The system comprises of feature extraction combining static metrics with LLM semantic analysis, vulnerability prediction, and adaptive time allocation with early stopping.

3.4.1 Feature Extraction

Accurate vulnerability prediction requires features that capture both structural complexity and algorithmic behavior. We extract 14 features organized into two categories.

Static Analysis Features: Building upon the metrics used in GreenFuzz [19], we compute six static code complexity metrics using SciTools Understand [27] and custom analyzers, including total lines of code (LOC), total cyclomatic complexity, total cognitive complexity, and three Halstead metrics (volume, difficulty, effort). These metrics provide baseline indicators of code size and structural complexity. For example, high cyclomatic complexity often indicates deeply nested control flow that may contain edge case bugs.

LLM Semantic Features: Static metrics cannot distinguish between algorithmically safe and unsafe code. A program with low cyclomatic complexity may still implement $O(n^2)$ logic where constraints require $O(n \log n)$, or use `int` datatype where `long` is needed to avoid overflow. To address this, we prompt an LLM to analyze each program against its problem specification as a human would do, and assign risk scores (0-10) across eight vulnerability categories shown in Table Table 2.

Table 2: LLM semantic vulnerability features (scored 0-10)

Category	Detection Focus
Array bounds risk	Unchecked array accesses, missing bounds validation
Integer overflow	`int` where `long` needed (e.g., summing $n = 10^6$ values)
Null pointer risk	Dereferencing without null checks, uninitialized variables
Edge case handling	Handling of $n = 0$, $n = 1$, empty input, maximum constraint values
Off-by-one error	For a 0-indexed array of size n, accessing the `n+1`th element
Input validation	Validation against problem constraints
Logic error risk	Algorithm correctness (e.g., greedy where DP is needed)
Timeout risk	Algorithmic complexity vs constraints (e.g., $O(n^2)$ with $n = 10^6$)

Our LLM prompt includes the problem statement and the agent code, along with a detailed scoring rubric for each category. For example, the integer overflow rubric specifies: 0 points for correct use of `long`, 3 points for potential overflow in edge cases, 7 points for `int` used in intermediate calculations that can overflow given constraints, and 10 points for definite overflow like `int*int` multiplication without casting. This prompting approach with guided rubric reduces LLM hallucination and ensures consistent scoring across problems.

These semantic features complement static metrics by reasoning about problem-specific risks. For instance, two programs with identical cyclomatic complexity may receive vastly different timeout risk scores if one uses exponential recursion while the other uses iteration.

3.4.2 Vulnerability Predictor Model

We train a Random Forest [6] classifier to predict the probability of vulnerabilities from the 14 extracted features. Among different classification approaches such as logistic regression, SVM, and neural network, our experiments show that Random Forest serves best in terms of performance and simplicity.

We used 100 decision trees with a maximum depth of 10 and balanced class weights to handle the imbalanced distribution of buggy versus clean code in our dataset. The model is trained on a 50-50 stratified split of our CSES dataset with 10-fold stratified cross-validation to ensure robust performance estimates. For each program i, the model outputs vulnerability probability $p_i \in [0,1]$ representing the estimated likelihood of containing vulnerable bugs.

3.4.3 Adaptive Time Allocation

Given the vulnerability probabilities for all programs, we allocate fuzzing time to maximize discovering bugs under resource constraints. This allocation involves filtering and proportional distribution.

Threshold-Based Filtering: Programs with vulnerability probability below threshold θ are excluded from fuzzing. Let $S = \{i : p_i \geq \theta\}$ denote the set of included programs. This filtering strategy aggressively reduces the fuzzing search space by focusing resources on high-risk targets. By using $\theta = 0.3$, we achieve 85.7% precision while retaining 90.2% of true bugs. We discuss this further in our evaluation.

Proportional Time Allocation: For included programs, we allocate time proportional to vulnerability probability:

$$t_i = \begin{cases} \frac{p_i}{\sum_{j \in S} p_j} \cdot T_{\text{budget}} & \text{if } i \in S \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where T_{budget} = is the total budget given by a user. This ensures that the total fuzzing time scales linearly with the number of included programs while allocating proportionally more time to higher-risk targets within that set.

Example Calculation: Consider three programs with probabilities $p_1 = 0.8$, $p_2 = 0.4$, $p_3 = 0.2$ and $\theta = 0.3$, $T_{\text{budget}} = 120$s. Program 3 is filtered ($p_3 < \theta$), leaving $S = \{1,2\}$. As per our allocation scheme, Program 1 receives $t_1 = 80$s, and program 2 receives $t_2 = 40$s for fuzzing.

3.4.4 Early-Stopping Scheduler

Studies on fuzzing have shown that coverage grows rapidly initially, and then plateaus as the fuzzer exhausts reachable states [4]. Continuing to fuzz after saturation wastes time, which should have been better spent on testing other targets. To address this, our scheduler implements dynamic early stopping based on coverage stagnation detection.

During fuzzing, we parse Jazzer's real-time output to extract combined coverage metrics (edge coverage + feature coverage) at regular intervals. We maintain a timestamp of the last coverage increase for each target. When current coverage remains unchanged for a saturation window w, the scheduler preempts the running fuzz instance and schedules the next one.

4 Implementation

We implement our system in Python using scikit-learn [24] for ML model training, Jazzer [9] for coverage-guided fuzzing, and DeepInfra [10] for LLM inference. Our code is organized into modules, including LLM clients, fuzz harness generator, static analysis extractors, vulnerability predictor, and adaptive fuzzing orchestrator. Below we describe our implementation choices.

LLM Infrastructure: We implement a unified LLM client supporting three back-ends: Ollama [22] for local inference, vLLM [16] for high-throughput serving, and DeepInfra [10] for cloud-based API access. We initially attempted local inference with vLLM across 2-4 GPUs (L40S, H200, H100) on Georgia Tech's PACE cluster, but transitioned to DeepInfra due to GPU availability constraints. For all experiments, we use Qwen/Qwen3-Coder-480B-A35B-Instruct-Turbo [25] for its strong code understanding capabilities.

Fuzzing Framework: We use Jazzer [9], a coverage-guided fuzzer built on libFuzzer [28] with JVM integration. Our adaptive scheduler parses Jazzer's output dynamically using regex to extract coverage metrics and detect saturation. Crashes are saved with SHA-1 hashed filenames for automatic de-duplication.

Feature Extraction Pipeline: Static metrics are extracted using SciTools Understand [27] for LOC, cyclomatic complexity, and Halstead metrics, and a custom javalang-based analyzer for cognitive complexity. LLM semantic features are extracted via inference through DeepInfra with structured prompts.

Hardware and Deployment: Fuzzing experiments run on a Linux machine with an Intel Core i7-10750H (6-core, up to 5.0 GHz) and 16GB RAM. We run each Jazzer instance sequentially due to memory constraints. Despite limited hardware, Jazzer's enables effective fuzzing. Static analysis and ML training are performed on the same machine.

5 Evaluation

We evaluate our approach on CSES algorithmic problems to answer four key questions: (1) How does our hybrid model compare to static-only approaches in vulnerability discrimination? (2) What is the end-to-end bug detection performance and resource savings versus baselines? (3) How does the time budget allocation affect the number of bugs detected? (4) How does the fuzz harness generation scale?

5.1 Experimental Setup

5.1.1 Dataset

Our dataset consists of 96 algorithmic problems from the CSES benchmark [17]. The problems cover a wide range of topics, including sorting, greedy algorithms, dynamic programming, graphs, trees, range queries, and mathematics. We manually write an optimal solution (to serve as ground truth) for each problem and verify it against the official test suite.

We generate 12 variants of each problem using our prompt variant generator. An LLM coding agent then generates solutions for all variants, resulting in a total of 1,152 solutions. We run these solutions against official test suites and label each solution as either buggy or clean. For the vulnerability predictor, we use a 50-50 stratified train-test split. The test set contains 336 buggy and 240 clean solutions. Bug categories include timeouts due to algorithmic complexity violations, integer overflows, off-by-one errors, stack overflows, memory out-of-bounds accesses, logic errors, and non-deterministic behavior.

5.1.2 Baselines

We compare against four approaches: (1) ChatUniTest, a state-of-the-art LLM-based unit test generator, (2) Fixed-time baseline fuzzing that allocates equal fuzzing time to all fuzz targets, (3) GreenFuzz, an ML-based approach using only static features for vulnerability prediction, (4) Our approach. All baselines run on identical hardware to ensure fair comparison. Moreover, the last three approaches used the same fuzz harnesses generated by our harness generation agent.

5.1.3 Evaluation Metrics

We measure the number of bugs caught, recall (% of actual bugs detected), accuracy (% of solutions correctly identified as buggy or clean) and time cost (total minutes for complete fuzzing campaign).

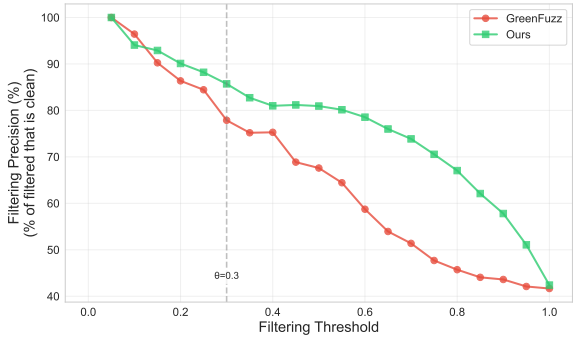

Figure 3: Filtering precision vs vulnerability threshold for GreenFuzz and our approach

5.2 Model Discrimination Quality (RQ1)

Our model significantly improves vulnerability discrimination over GreenFuzz's model using only static features. We achieve a mean cross-validation ROC-AUC of 0.943 and outperform the baseline (0.827 ROC-AUC). Figure 3 shows filtering precision across different vulnerability thresholds. At $\theta = 0.3$, GreenFuzz filters 113 programs (88 clean, 25 buggy) achieving 77.9% precision, while our approach filters 231 programs (198 clean, 33 buggy) with 85.7% precision.

Figure 4 further visualizes this discrimination capability. Points far below the diagonal "Equal Filter Rate" line indicate better performance, that is, filtering more clean code and retaining buggy code for fuzzing. At $\theta = 0.3$, our approach filters 82.5% of clean code while filtering only 9.8% of buggy code, compared to GreenFuzz which filters 36.7% clean and 7.4% buggy. By filtering more than twice as many programs (231 vs 113) while missing only 8 additional bugs, our model allows for aggressive resource savings for fuzzing without significantly affecting bug detection.

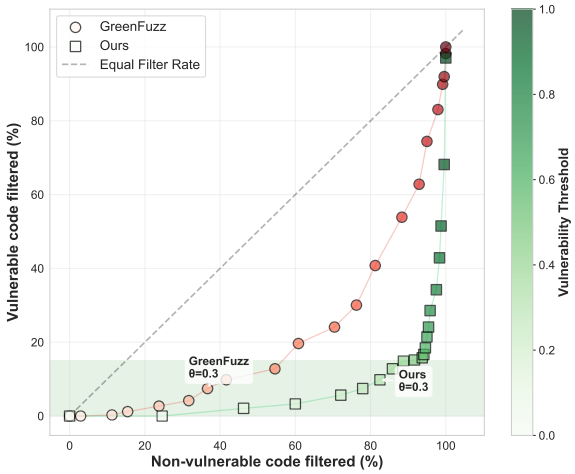

Figure 4: Discrimination capability: clean code filtered vs buggy code filtered across different thresholds

The improvement can be attributed to the reason that LLM semantic features capture algorithmic complexity mismatches that static metrics miss. For example, static analysis cannot detect $O(n^2)$ solutions to $O(n)$-constrained problems, but our timeout risk feature identifies such mismatches by comparing implementation complexity against problem requirements. Similarly, our overflow risk feature can identify when a program performs long summations, and a `long` data type should have been used instead of `int`. Static analysis cannot predict whether an integer overflow will occur. However, LLMs can parse the code like a human, understand the constraints of the problem, and infer whether the input values can cause an overflow. The hybrid model correctly filters out programs that appear complex by static metrics but are algorithmically correct.

Table 3: End-to-end bug detection performance across approaches. Our approach achieves strong recall with significant time savings.

Approach	Bugs (336)	Clean Filtered	Fuzz Time (min)	Recall (%)	Acc. (%)
ChatUniTest	163	0	68.6	48.6	63.0
Fixed Fuzz	**245**	0	336.5	**72.9**	76.7
GreenFuzz	232	88	242.9	69.1	76.9
Ours	226	198	**141.3**	67.3	**78.9**

Further, to assess the generalizability of our model beyond CSES, we evaluated it on a sample of problems from the LeetCode dataset. Without any additional training, our model achieved 0.897 ROC-AUC indicating strong transfer learning capability. However, the model's prediction ability is currently limited to algorithmic problems. Extending to general software codebases or predicting common CVEs would require retraining with domain-specific vulnerability datasets.

5.3 End-to-End Bug Detection Performance (RQ2)

Table 3 shows end-to-end fuzzing performance in all approaches. ChatUniTest detects only 163 bugs out of 336 known bugs (48.6% recall) because unit tests cannot stress-test algorithmic complexity and have limited ability in catching timeouts and overflows. Fixed fuzzing achieves the highest recall (81.9%) but requires a fuzzing campaign of 336.5 minutes, which is very large considering the algorithmic problems and simply does not scale in large codebases.

GreenFuzz improves over ChatUniTest in terms of catching vulnerabilities. It improves over fixed fuzzing in terms of time for fuzzing campaign by filtering out potentially non-vulnerable code. But, it allocates equal resources to the chosen problems irrespective of their vulnerability score. For example, the problems `elevator_rides_buggy_timeout` (buggy) and `elevator_rides_original` (clean) get an equal amount of fuzzing time (60s). Our hybrid approach detects 226 bugs in 141.3 minutes, achieving 1.7× speedup over green fuzzing with comparable effectiveness. For the same set of problems, our approach allocates `elevator_rides_buggy_timeout` (buggy) 71 secs and `elevator_rides_original` (clean) 27 secs for fuzzing. The speedup gain can be justified by the usage of an improved discrimination model and efficient resource allocation.

The results demonstrate that semantic features enable intelligent prioritization, focusing fuzzing resources on high-risk targets while safely filtering likely-clean code. Overall, our approach achieves the best time-recall tradeoff.

5.4 Time Budget-Recall Tradeoff (RQ3)

A general observation in fuzzing is that we tend to find more bugs when fuzzing for longer time on a given harness. The question we want to answer is: how much recall do we sacrifice when choosing a specific time budget? We experimented with a range of time budgets: 2s, 5s, 15s, 30s, 45s, 60s, 75s, and 90s (time per problem), collected all true positive bugs, and showed the relationship in Figure 5. The overall trend is increasing, and the rate of increasing is larger with smaller total fuzz time while plateauing after we go beyond 175 minutes. The mild fluctuations from the last few data points could be caused by different machines running these experiments. This experiment demonstrates that after investing a certain amount of fuzz time, the marginal gain in recall becomes minimal. This result provides us with valuable empirical guidelines for setting the fuzz time budget.

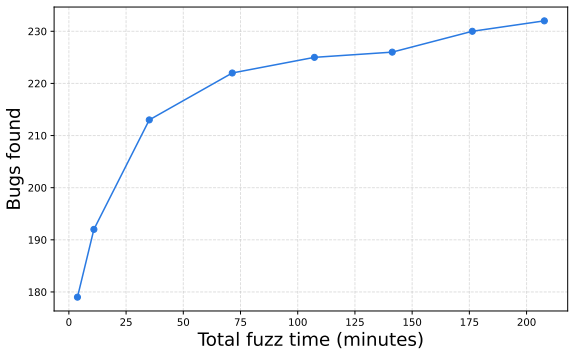

Figure 5: Tradeoff between time budget and bugs caught

5.5 Scalability of Harness Generation Agent (RQ4)

To investigate how well our fuzz harness generation agent scales, we ran experiments on 288 problems randomly selected from our test set. As Figure 6 shows, the cumulative time curve (purple) increases at a constant slope, showing that the agent scales linearly as the workload increases. We also show a breakdown of time taken per problem, where most problems spend less than 20 seconds and either succeed at initial generation or go through the retry and fix loop. Occasionally, the total time spikes, which corresponds to outlier problems that take longer time than average, primarily due to complexity. The limited retry mechanism makes sure that the generation is bounded and does not blow up in exponential time. Overall, we see a good linear relation in this scalability curve, with a few outliers but no cascading slowdowns.

6 Related Works

LLM-Guided Fuzzing: Recent work has explored using LLMs to improve fuzzing efficiency and coverage. Fuzz4All [30] introduced the first universal fuzzer that uses LLMs as input generators. They employ autoprompting to encode problem descriptions into effective prompts and improve the test coverage. Although powerful, Fuzz4All is limited to testing compilers and interpreters. OSS-Fuzz-Gen [20]

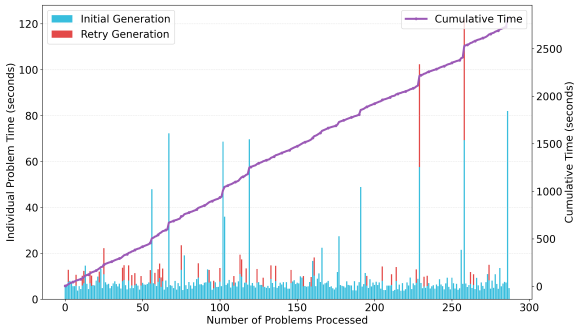

Figure 6: Assessing scalability of harness generation

automates fuzz harness generation using multi-agent LLM systems with feedback loops. However, the generated harnesses are generic and often produce false positive crashes due to incorrect input constraints. FalseCrashReducer [2] addresses this with constraint-based drivers that reduce the number of spurious crashes. Our work differs by generating problem-specific oracles that embed semantic constraints directly from problem descriptions. As a result, we reduce false positives and catch vulnerabilities that generic harnesses miss. GreenFuzz [19] proposes ML-based stopping criteria using prediction on static features, terminating fuzzing campaigns 6-12 hours earlier. We extend this approach with LLM semantic features that capture algorithmic properties, improving the filtering precision and better allocation of fuzzing resources..

Agentic Code Auditing: Recent research in AI safety has triggered a shift from static benchmarks to dynamic agent-based auditing [12, 14, 15]. RepoAudit [15] employs LLM agents for static analysis in repository-level code auditing to detect bugs such as null pointer dereferences and memory leaks. Tools like Anthropic's Petri [12] and Microsoft's RedCodeAgent [14] utilize adversarial agent interactions to test models for unintended behaviors and security jailbreaks. They focus on invoking harmful outputs or unsafe code execution through multi-turn conversations. Our framework complements these efforts by specifically targeting algorithmic vulnerabilities through semantic-aware fuzzing and stress-testing. It fills the gap where traditional static auditing and safety red-teaming fail to detect timeout, overflow, and complexity-violation bugs.

7 Future Work

Our evaluation reveals limitations of fuzzing-only approaches, which we plan to address as a part of our future work. While our adaptive fuzzing achieves a good recall, even fixed-time fuzzing with maximum resource allocation (336 minutes) catches only 245 of 336 bugs. The remaining bugs are either functionally incorrect or have subtle vulnerabilities that fuzzing cannot catch through random input generation alone. Unit test generators are better at functional correctness validation but miss performance and resource-based bugs. A hybrid approach that combines LLM-guided fuzzing for algorithmic vulnerabilities with LLM-based unit test generation for functional correctness could provide comprehensive coverage.

Our current implementation focuses on Java and algorithmic vulnerabilities, including timeouts, crashes, overflows, and logic errors. Expanding to general software Common Vulnerabilities and Exposures (CVEs), such as SQL injection, authentication bypasses, etc., would require developing CVE-specific oracles. Additionally, it would require training vulnerability predictors on diverse bug datasets, and adapting fuzz harness generation to target code with external dependencies. Supporting multiple programming languages, such as Python and C++, would broaden our system's scope. It would require integrating language-specific fuzzing frameworks and static analyzers.

8 Conclusion

In our work, we introduce a hybrid LLM-guided fuzzing framework to catch algorithmic vulnerabilities in AI-generated code at scale. Our system uses prompt variations to mimic user interactions, an LLM-guided harness generator that effectively captures problem-specific constraints and stress-tests inputs, and a vulnerability predictor to enable intelligent, adaptive time allocation. Our approach shows improved vulnerability discrimination, filters 2x more non-vulnerable targets, and achieves a 1.7x speedup with similar recall to the baselines.

Acknowledgments

We thank Professor Anand Iyer and Vima Gupta for their guidance and continuous feedback. We acknowledge CSES for providing the algorithmic problem data set and test suites.

References

[1] Mostafijur Rahman Akhond, Saikat Chakraborty, and Gias Uddin. Llm for loop invariant generation and fixing: How far are we?, 2025.

[2] Paschal C. Amusuo, Dongge Liu, Ricardo Andrés Calvo Méndez, Jonathan Metzman, Oliver Chang, and James C. Davis. FalseCrashReducer: Mitigating false positive crashes in OSS-Fuzz-Gen using agentic ai. In *arXiv preprint arXiv:2510.01234*, 2025.

[3] Anthropic. Claude code: Ai-powered coding assistant. https://www.claude.com/product/claude-code, 2025.

[4] Marcel Böhme and Brandon Falk. Fuzzing: On the exponential cost of vulnerability discovery. In *Proceedings of the 28th ACM Joint Meeting on European Software*

Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pages 713–724, 2020.

[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1032–1043, 2017.

[6] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, October 2001.

[7] Junkai Chen, Li Zhenhao, Hu Xing, and Xia Xin. Nlperturbator: Studying the robustness of code llms to natural language variations. *ACM Trans. Softw. Eng. Methodol.*, July 2025. Just Accepted.

[8] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 572–576, 2024.

[9] Code Intelligence. Jazzer: Coverage-guided fuzzing for the jvm. https://github.com/CodeIntelligenceTesting/jazzer, 2021.

[10] DeepInfra. Deepinfra: Llm models and infrastructure. https://deepinfra.com/, 2025.

[11] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++: combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies*, WOOT'20, USA, 2020. USENIX Association.

[12] Kai Fronsdal, Isha Gupta, Abhay Sheshadri, Jonathan Michala, Stephen McAleer, Rowan Wang, Sara Price, and Samuel R. Bowman. Petri: Parallel Exploration Tool for Risky Interactions, 2025. Open-source auditing tool to accelerate AI safety research.

[13] GitHub. Github copilot: Your ai pair programmer. https://github.com/features/copilot, 2025.

[14] Chengquan Guo, Chulin Xie, Yu Yang, Zhaorun Chen, Zinan Lin, Xander Davies, Yarin Gal, Dawn Song, and Bo Li. RedCodeAgent: Automatic red-teaming agent against diverse code agents. *arXiv preprint arXiv:2510.02345*, 2025.

[15] Jinyao* Guo, Chengpeng* Wang, Xiangzhe Xu, Zian Su, and Xiangyu Zhang. Repoaudit: An autonomous llm-agent for repository-level code auditing. In *Proceedings of the 42nd International Conference on Machine Learning*, 2025. *Equal contribution.

[16] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

[17] Antti Laaksonen. Cses problem set. https://cses.fi/problemset/, 2025.

[18] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[19] Stephan Lipp, Daniel Elsner, Severin Kacianka, Alexander Pretschner, Marcel Böhme, and Sebastian Banescu. Green fuzzing: A saturation-based stopping criterion using vulnerability prediction. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 127–139, New York, NY, USA, 2023. Association for Computing Machinery.

[20] Dongge Liu, Oliver Chang, Jonathan Metzman, Martin Sablotny, and Mihai Maruseac. Oss-fuzz-gen: Automated fuzz target generation. https://github.com/google/oss-fuzz-gen, 2024.

[21] {Valentin J.M.} Manes, Hyungseok Han, Choongwoo Han, {Sang Kil} Cha, Manuel Egele, {Edward J.} Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, November 2021. Publisher Copyright: © 1976-2012 IEEE.

[22] Ollama. Ollama: Run large language models locally. https://ollama.com/, 2025.

[23] Andrei Paleyes, Radzim Sendyka, Diana Robinson, Christian Cabrera, and Neil D. Lawrence. Prompt variability effects on llm code generation, 2025.

[24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[25] Qwen Team. Qwen3 technical report, 2025.

[26] Laboni Sarker, Mara Downing, Achintya Desai, and Tevfik Bultan. Syntactic robustness for llm-based code generation, 2024.

[27] SciTools. Scitools understand: Static code analysis tool. https://scitools.com/, 2025.

[28] Kostya Serebryany. libfuzzer – a library for coverage-guided fuzz testing. In *LLVM Developers' Meeting*, 2016.

[29] Anjiang Wei, Zhiyuan Yan, Hanjun Dai, Yuxiang Wei, Jingxuan He, Maolin Wei, and Xujie Si. Invbench: Can llms accelerate program verification with invariant synthesis?, 2025.

[30] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4All: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, 2024.