1328

PennState
College of Engineering

Can LLMs Recover Program Semantics? A Systematic Evaluation with Symbolic Execution

Rong Feng and Dr. Suman Saha
The Department of Computer Science and Engineering
The Pennsylvania State University, University Park, PA 16802, USA

Abstract

In this work, we investigate whether LLMs, when fine-tuned with symbolic execution artifacts, can effectively deobfuscate programs and restore analyzability. We construct a benchmark by applying four widely studied transformations across diverse C programs from TUM Obfuscation Benchmarks, the LLVM test suite, and algorithmic repositories. We then compare three state-of-the-art LLMs under two training configurations: baseline fine-tuning on obfuscated/original code pairs, and enhanced fine-tuning with additional KLEE. Our evaluation examines syntactic correctness (compilation success), semantic fidelity (behavioral equivalence under symbolic execution), and code quality (readability and structure). Results show that GPT-4.1-mini achieves the strongest deobfuscation overall, and that incorporating KLEE artifacts consistently improves semantic preservation and compilation success across models. These findings highlight deobfuscation as a broader software engineering concern, demonstrating that combining LLMs with symbolic execution can strengthen automated testing, static analysis, and program comprehension in the presence of obfuscation.

Data Collection and Processing

Data Collection

We constructed our dataset from three complementary sources to capture programs of varying size and complexity. The first source was the TUM repository, which provides small but well-structured C programs widely used in software engineering benchmarks. The second source was the LLVM test suite, which contains medium-sized programs covering diverse computational patterns. Finally, we included programs from a public Algorithms repository, consisting of implementations of common algorithms such as sorting, searching, and dynamic programming. Together, these sources provided a broad set of code fragments suitable for evaluating deobfuscation.

Data Filtration

Since obfuscation can drastically increase program size and symbolic reasoning costs, we applied two filtering steps. First, we excluded programs that led to path explosion in KLEE. In practice, this removed a significant number of programs where symbolic execution generated an intractable number of execution paths. Second, we filtered programs based on context size, discarding those that produced inputs exceeding the maximum token window of our target LLMs. After filtering, we retained a balanced set of programs across the three repositories.

Obfuscation Transformations

We applied four widely studied obfuscation transformations. These transformations were chosen because they distort program structure at both the control-flow and data-flow levels, and they are commonly implemented in tools such as Obfuscator-LLVM. Below we illustrate each transformation with their definition:

- Control-Flow Flattening (CFF): CFF replaces structured branching with a dispatcher loop and a switch statement, obscuring the natural flow of execution.
- Opaque Predicates (OP): OP introduces conditions that always evaluate to the same outcome but appear complex, misleading analysis tools.
- Arithmetic Encoding (AE): AE rewrites simple arithmetic expressions into longer but equivalent forms.
- Branch Encoding (BE): BE hides branch conditions by encoding them through bit manipulations or tables.

KLEE Outputs and Fine-Tuned Model Details

To analyze program paths and generate training data, KLEE produces several key artifacts during symbolic execution:

- SMT: These are constraint files representing the program's symbolic execution paths, used by solvers (like Z3) to check if paths are feasible.
- kquery: KLEE's internal constraint format. It's a low-level representation of symbolic constraints before conversion to SMT-LIB format.
- istats: Execution statistics files generated by KLEE. They record metrics like instructions executed, path counts, and time usage for performance analysis.
- ktest: Test case files produced when KLEE finds concrete inputs satisfying the path constraints. Each .ktest corresponds to one program path and input set.

For the fine-tuning stage, we compare different LLMs to understand model capacity and scalability. The table below (Table 1) illustrates the context length for each model used.

Model	Model ID	Context Length
GPT	GPT-4.1-mini-2025-04-14	1,047,576
Ministral	Ministral-8B-latest	128,000
Codestral	Codestral-latest	256,000

Table 1. Model Context Length

Approach

Our approach combines program obfuscation, symbolic execution, and large language model (LLM) fine-tuning in a unified workflow (Figure 1). We begin with source programs drawn from three repositories and apply four widely studied obfuscation techniques to create challenging inputs. In parallel, we use KLEE on the original programs to produce symbolic artifacts, such as constraints, path statistics, and test cases, which serve as semantic ground truth. These artifacts are then paired with the obfuscated code and used to fine-tune LLMs under two configurations: a baseline setting that relies only on code, and an enhanced setting that incorporates symbolic artifacts. At inference time, the models are prompted with obfuscated programs and expected to generate deobfuscated outputs. We then evaluate these outputs using syntactic, semantic, and quality measures.

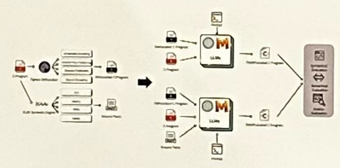

Fig. 1. Overview of our approach.

Results

To evaluate the impact of KLEE artifacts on fine-tuning performance, we compare models trained with KLEE vs. without KLEE across three key metrics: compilation success, semantic accuracy, and overall code quality. The following graphs and tables summarize the findings.

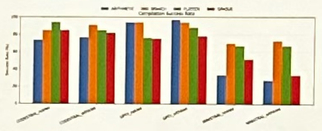

Fig. 2. Compilation Success Rate for All Models and Transformations (No-KLEE vs. KLEE)

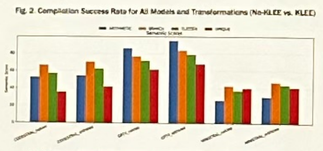

Fig. 3. Semantic Scores for All Models and Transformations (No-KLEE vs. KLEE)

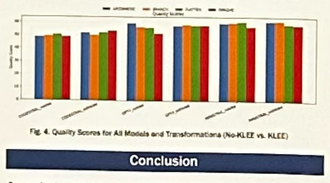

Fig. 4. Quality Scores for All Models and Transformations (No-KLEE vs. KLEE)

Conclusion

Our results demonstrate that incorporating KLEE-generated artifacts during fine-tuning consistently improves compilation success, semantic correctness, and in many cases quality scores across all obfuscation transformations.

- GPT-4.1-mini with KLEE achieved the highest compilation rates (up to 96.9%) and semantic fidelity (up to 94.7%), showing strong reliability under complex obfuscations.
- Codestral models also benefited moderately from KLEE, while Ministral improvements were smaller and less consistent.
- Overall, the addition of KLEE artifacts provides a robust signal for preserving program correctness and semantics during code generation.

These findings highlight the potential of symbolic execution artifacts in enhancing LLM performance on code understanding and deobfuscation tasks.

RONG
Rong Feng
The Pennsylvania State University