134

Engineering

Scan for abstract

136

Health and Life Sciences

The 2025 Undergraduate EXHIBITION

Rong Feng

Engineering

134

PennState College of Engineering

Symbolic Execution Without Execution: AI-Powered Constraint Prediction

Rong Feng, Vanisha Gupta, Viroopaksh Reddy Ernampati, Vivek Patel, and Dr. Suman Saha
The Department of Computer Science and Engineering
The Pennsylvania State University, University Park, PA 16802, USA

Abstract

Symbolic execution helps check programs by exploring different paths based on symbolic inputs. Tools like KLEE are commonly used because they can automatically detect bugs and create test cases. In this project, we wanted to see if a large language model like GPT 4o could simulate the kinds of outputs that KLEE generates. The idea was to explore whether LLMs could one day replace parts of symbolic execution to save time and resources. One specific goal was to have GPT-4o identify the most constrained path in a program—this is the execution path with the most symbolic conditions. These paths are especially important because they often represent edge cases that are harder to test and more likely to contain deep bugs. However, figuring this out usually requires fully running KLEE, which can be expensive. So, we tested whether GPT-4o could predict the KLEE outputs and the most complex path using a dataset of 100 C programs. Our results showed about 20% accuracy in generating KLEE-like outputs and identifying the most constrained path. While not highly accurate, this early work helps show what current LLMs can and can't do when it comes to simulating symbolic execution.

Data Collection and Processing

The process of fine-tuning an LLM to approximate symbolic execution relies heavily on the quality and structure of the training data. Our dataset is derived from 100 C programs executed using KLEE, which generates multiple output files. Each component of the dataset serves a specific purpose in understanding program execution paths and constraint solving. Below, we describe the data collection process, data structure, and preprocessing steps to prepare for fine-tuning the LLM.

Data Collection

...re self-contained with a main() function and free of ...r-input functions like scanf() or fgets(). A Python script ...atible code to ensure KLEE compatibility.

...ically executed using KLEE, which produces multiple ...e then categorized into input and output components

Data Processing

...KLEE and Used as the Model Target): The following ... program execution:

...tistics): Contains execution statistics, including:

...uted → Represents program complexity.
...ken → Indicates path divergence.
...ures constraint-solving complexity.

...straints Representation)
...traints imposed on symbolic variables.
...ch is compatible with solvers like Z3

...Test Cases)
...or symbolic variables that satisfy

...ternal Query Representation)
...nts in KLEE's custom format.

... Most constrained execution path
...he most constrained execution path. They reflect ...entation. We counted the number of conditions in ...th the highest count. On average .kquery files had ...m stood out with 180 .kquery files and a peak ...ile being only 13 lines long, showing how small ...ighly complex execution paths during symbolic

Purpose of Each Data Components

Component	Description
run.istats	• Helps determine execution complexity. • Useful for identifying the longest path (critical path). • Aids in predicting solver efficiency in symbolic execution.
test000001.smt2	• Defines constraints on symbolic variables (x in this case). • Determines feasible program paths based on constraints. • Helps the model learn the relationship between program logic and execution paths.
test000001.ktest	• Provides a real input example that follows symbolic constraints. • Helps verify correctness by replaying execution paths. • Allows the LLM to learn how test cases map to constraints.
test000001.kquery	• Represents the actual constraint system solved by KLEE. • The LLM is trained to predict kquery from the C program. • The most complex part of symbolic execution, as it captures all execution paths.

Table 1: Purpose of each data components from KLEE

Approach

Run KLEE on Training Programs:

- Execute KLEE on 100 C programs.
- Collect (run.istats, smt2, ktest, kquery) for each program.
- Identity the most constrained path from .kquery

Fine-Tune GPT-4o:

- Used 80 programs for training and 20 for testing
- Each example was automatically structured as a conversion by using a Python script

Testing on Unseen Programs:

- Provide twenty new C programs to the fine-tuned model.
- Generate predicted (istats, smt2, ktest, kquery, most constrained path).

Comparison & Evaluation:

- Measure similarity between predicted vs. actual KLEE outputs.

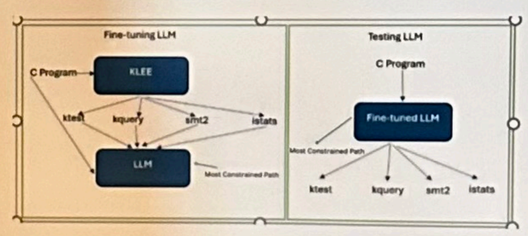

Figure 2: Using LLM to identify to generate Symbolic Execution components

Results

Component	Accuracy
# of files match	20%
Correct Path Predicted	20%
Most Constrained Path Complexity	50%

Table 2: Accuracy rate of each of the predicted components

Future Work

While this research demonstrates the feasibility of using GPT-4o to approximate KLEE's symbolic execution outputs, there are several areas for improvement and expansion. Future work will focus on enhancing prediction accuracy, scalability, and integration with existing program analysis tools. Below are the key directions for future research:

1. Hybrid Approach: Combining Partial KLEE Execution with LLM Predictions

Generate only istats and smt2, skipping full constraint solving (kquery). Use the LLM to predict kquery, reducing symbolic execution time.

2. Applying LLM-Powered Symbolic Execution to Security Analysis

Use LLM-predicted constraints for automated test case generation. Focus on buffer overflows, integer overflows, and null pointer dereferences.

Malware analysis & reverse engineering: Apply symbolic execution to obfuscated binaries to extract logic flows.

3. Scaling Up to Large-Scale Real-World Programs

Benchmark performance on large-scale software, such as: Open-source projects (e.g. Linux kernel drivers, cryptographic libraries). Real-world C/C++ applications used in security research.

4. Extend to static analysis tools

Integrate LLM-based constraint prediction into Clang static analyzer or LLVM sanitizers. Reduce false positives in taint analysis and vulnerability detection.

Conclusion

In this work, we explored whether a large language model like GPT-4o could simulate the outputs of KLEE. We trained the model using 100 C programs and their corresponding KLEE outputs to see if GPT-4o could learn to produce similar results without actually running KLEE. While the model only achieved about 20% accuracy in file generation and identifying the most constrained execution path, this early attempt shows that LLMs may still have potential in this area. Even though the results weren't strong, this project gave us a clearer view of how far off current models are from handling tasks that require precise symbolic reasoning. It also showed that using an LLM could reduce the time and resources needed for symbolic execution, which could be helpful for scaling analysis across large codebases.