LLM-Assisted Fuzzing and Symbolic Execution Harness Generation

Mahima Chaudhary, Vishva Arasan, Sahasra Boddula, Daniil Novak, Wendy Contreras Martinez
Advisors: Professor Bultan, Md Shafiuzzaman, Professor Matni

January 4, 2026

1 Research Context and Problem Statement

Today's software plays a role in running every major system in the world: from banking to health care. These systems have to be as reliable as possible and even the smallest mistake can be catastrophic. The average cost of a data breach in the US is $4.4 million USD and that's not even accounting the possibility of lawsuits being pursued [5]. To prevent these events from happening, it is important that every possible bug or vulnerability is addressed before the software is published. There are a variety of tools that can efficiently find these bugs before they are published into the actual software. The aim of such tools is to explore as many vulnerabilities in the code-base as possible.

A common way to find bugs more effectively is to use symbolic execution [2]. When a program normally runs, it uses concrete values that are fixed once the program is compiled. Symbolic execution works differently. Instead of assigning a single value to a variable, the tool treats the variable as symbolic, meaning it can represent many possible values at the same time. As the symbolic execution engine runs, it records the logical conditions that must be true for the program to take each branch. These conditions are called path constraints.

Consider this if-else statement:

```
if (x > 10) {
 // branch A
} else {
 // branch B
}
```

If x is symbolic, the tool does not pick one branch. It considers both. One path has the constraint ($x > 10$) while the other has ($x \leq 10$). An SMT (Satisfiability Modulo Theories) solver then checks whether these logical conditions are satisfiable and produces concrete values that would cause the program to follow each path [8]. This lets the tool reach parts of the code that ordinary testing might never hit. Although this aims for high code coverage, symbolic execution needs careful setup, and as programs grow larger, the number of possible paths can explode.

KLEE is one of the most widely used symbolic execution engines for C programs [2]. To use KLEE, the user must write a harness. A harness is a small driver program that tells KLEE what to analyze. In practice, a harness usually does two things:

- Marks certain function inputs as symbolic.

- Calls these target functions with the new symbolic inputs.

This matters because symbolic execution cannot automatically explore an entire large codebase. The harness decides where the engine begins, which inputs can vary, and what part of the program KLEE should treat as the main focus. If the harness is vague or poorly structured, KLEE may waste time exploring irrelevant paths or miss bugs entirely. This need for careful manual setup is one of the main reasons symbolic execution is hard to apply at scale.

Static analysis tools help by examining the code without running it and pointing out areas that might be risky and likely to contain real vulnerabilities [7]. These tools search for patterns such as unchecked memory access, pointer misuse, or functions that depend heavily on external input. However, even with this help, writing the harness still requires technical skill, and small mistakes can significantly reduce the effectiveness of symbolic execution.

Fuzzing is another common way to find bugs [4]. In its simplest form, black-box fuzzing, it sends random or mutated inputs into a program without knowing how the code works. It can trigger crashes and find simple bugs, but it struggles with deeper logic because it cannot see which paths exist in the code. White-box fuzzing improves this by pairing fuzzing with symbolic execution. It starts by mutating inputs like a normal fuzzer, and when it stops finding new paths, it uses symbolic reasoning to push further. SAGE is a well-known example. It showed that symbolic reasoning can uncover serious bugs in large real-world software, but it also revealed key limitations: complex constraints slow down solving, and path explosion remains a major obstacle.

In modern times, tools such as STASE attempt to reduce manual work by automatically generating harness components using static analysis and templates, but they still require humans to write or fix drivers [7].

With the advances in the sphere of LLMs and Generative AI, a myriad of possibilities have been opened in the field of symbolic execution, but there are many issues that persist, such as: limited context windows, hallucinations, and inconsistent output.

The goal of the project is to show that LLMs can be taught and assisted to understand symbolic execution. If our results are successful, then LLMs can be a significant step toward solving the problem that restricts symbolic execution from being truly scalable. If we are able to make symbolic execution accessible, then many new possibilities in software verification research could arise and lead to a better understanding of the subject as a whole.

2 Related Work

2.1 Overview

Our project builds on three areas of previous work:

1. Symbolic execution and how tools explore program behavior.
2. Research that tries to guide or automate symbolic execution.
3. Work that uses large language models for testing and code generation.

Together, these show why harness generation still depends on manual effort and why it is a good target for automation.

2.2 Symbolic Execution and Guided Program Analysis

Symbolic execution explores the behavior of a program by treating inputs as variables that can represent many values at once. As the program runs, the tool collects the conditions needed to reach each branch, and a solver then produces concrete inputs that satisfy those conditions. This allows symbolic execution to reach paths that regular testing usually misses.

2.3 Large scale symbolic execution: SAGE

SAGE is one of the most influential systems in this space [4]. It was developed at Microsoft as a white-box fuzzer that uses symbolic reasoning instead of random mutations. SAGE found many bugs in large production software and showed that symbolic execution can work at scale.

It also exposed two core limitations in the field

- Path explosion. The number of paths grows too quickly to explore.
- Expensive solver calls. The solver must work through complex logical conditions.

These limits connect directly to why the harness matters for our project. The harness controls how much of the program symbolic execution tries to explore. If it is vague or includes too much surrounding code, the tool is more likely to hit path explosion and spend time following branches that do not lead anywhere useful. A focused and well built harness keeps the analysis on track and helps symbolic execution move toward the parts of the code that actually matter.

2.4 Symbolic execution tools: KLEE

KLEE is a widely used symbolic execution engine for C programs that produces high coverage tests by exploring many execution paths [2]. KLEE operates at the function level and relies on the harness to set up the analysis. A harness typically does two main things:

- It marks specific inputs as symbolic, usually parameters that influence how the program moves between branches or interact with code that seems risky.
- It calls the target function with those symbolic inputs that earlier analysis or testing suggested were worth examining more closely.

KLEE cannot analyze an entire program at once. Because of that, the harness shapes the entire search space. This is one of the main reasons our project focuses on automating harness generation.

2.5 Guided symbolic execution: STASE

Recent work shows that symbolic execution benefits when it is guided by lighter forms of program analysis. STASE is a good example [7]. It uses static analysis to find code regions that look risky or error prone. These often include:

- unusual pointer usage
- unchecked buffer operations
- functions that handle external data

STASE first identifies these regions and then directs symbolic execution toward them. This reduces wasted exploration and increases the chance of uncovering meaningful bugs. STASE generates some harness structure through templates. This works when code follows known patterns, but it struggles with cases

that fall outside these patterns. Our project explores whether an LLM can generate complete, ready to use harnesses that do not depend on fixed templates.

2.6 Automating the Symbolic Execution Pipeline

Symbolic execution requires a fair amount of setup before it can run. A developer must write a harness that specifies what to test and how to expose inputs to the symbolic executor. This setup strongly affects the outcome. If the harness is unclear or incomplete, symbolic execution wastes time on irrelevant paths or fails to reach the behavior that actually matters. STASE reduces some of this work, but it still relies on rules and templates that do not adapt well to different code bases [7]. This leaves a gap in the field. Tools can guide symbolic execution, but they cannot create a full harness that is tailored to the target program. This is where language models offer a new direction.

Recent studies show that large language models can understand program structure and create testing code that follows real usage patterns. This makes them promising for tasks that require both reasoning and code generation.

2.7 Prompt based testing code

Prompt Fuzzing uses an LLM to produce fuzz drivers based on structured prompts [6]. A fuzz driver prepares the environment for testing an API. This is similar to how a harness prepares KLEE for symbolic execution. Existing work has shown that language models can read an API, understand how it is meant to be used, and produce working test code.

2.8 Solver related test generation

On a lower level, LLMs were able to generate formulas for SMT solvers, which are the same solvers used inside symbolic execution engines [8]. Their work shows that LLMs can produce logical structures that fit into solver driven workflows. This supports the idea that they can help with symbolic execution tasks.

2.9 Zero shot input generation

Recent developments in other fields of research, such as ML, have proven how LLMs can act as a zero shot fuzzers for deep learning libraries [3]. The model generated inputs that revealed real bugs even without task specific training. This suggests that LLMs can reason about how code behaves and generate meaningful tests on their own.

2.10 Structured code generation challenges

There have also been discoveries of new challenges and flaws in current approaches to using LLMs, for example how when LLMs generate parser tests various difficulties arise such as: keeping outputs well structured and designing prompts that avoid invalid code [1]. These findings specifically shaped the prompt templates used in our project.

2.11 Synthesis and Research Gap

Past work shows that symbolic execution works best when the harness is clear and well structured. Systems like STASE reduce some of the manual effort, but they rely on templates that cannot adapt to patterns outside their preset rules. Research on LLM based testing shows that models can write test code effectively, yet none of this work looks at generating harnesses for symbolic execution. This is where our project fits in. We use an LLM to produce full harnesses for KLEE, with the hope of lowering the amount of hand written setup needed to run symbolic execution. The idea is to create harnesses that adjust to the structure of the target program and make symbolic execution easier to use. If successful, it provides a more flexible and accessible way to apply symbolic execution in practice.

3 Proposed Solution

Although previous findings help create a general idea of how LLM Assisted Symbolic Execution should work, they do not directly implement it. Our solution uses these previous findings to create a pipeline that can be deployed on more than one dataset and be used on software that has been publicly published. The pipeline uses most of the fundamentals of Symbolic Execution as a foundation.

To automate harness generation and program simplification, our team uses advanced large language models (LLMs), such as Deepseek V3.2. The reasoning behind this decision is to make the LLM understand the relevant parts of the codebase and produce the harness components needed for symbolic execution through its advanced reasoning capabilities. However, LLMs have a crucial limitation: they cannot load an entire large codebase into their context window, or the maximum amount of text they can process at a time, at once. To work around this, we use a guided reasoning approach: the model breaks the task into smaller steps and then processes the necessary parts of the code at each step. The smaller steps include generating instrumented files by inserting assert statements at detected vulnerability locations and generating drivers that make inputs symbolic and call the instrumented files. This approach is more token efficient and doesn't over-rely on the LLM's reasoning abilities. We break the

Another major challenge is helping the LLM understand complex data structures, such as Linked Lists, Dictionaries, and Arrays. These values can't be directly understood through their values and the LLM needs some idea of their internal components. To address this we convert these structures into JSON representations. The LLM is given a clean and structured description of each object, making it easier for the model to reason about program behavior and generate accurate harness code.

A central part of our solution is to create prompt templates that allow the LLM to automatically generate both driver programs and instrumented files for many types of vulnerabilities. These prompts act as an example the model can follow, eliminating the need for developers to manually craft drivers or write specialized harness code. This greatly reduces the level of expertise required, since the LLM can fill in the correct structure, stubs, symbolic inputs, and function calls based on the template.

Traditionally, users of symbolic execution tools must manually:

- stub unsupported operations,
- write driver programs to make inputs symbolic, and
- prepare the final code so tools like KLEE can run successfully.

Our pipeline automates all of these steps. We begin by using lightweight tools—such as static analysis and fuzzing—to locate "error hotspots," or areas of code that are most likely to contain bugs. Once these hotspots are identified, we feed the context to the LLM so it can edit or restructure the relevant code and automatically generate a suitable harness so that KLEE can execute the program and explore the paths that matter most.

By combining static analysis, guided LLM reasoning, JSON-based program simplification, and prompt-based harness templates, our approach aims to create a fully automated symbolic execution pipeline. With this added assistance the pipeline should ideally be able to reliably use the LLM to have a near identical performance to STASE.

4 Evaluation and Implementation Plan

4.1 Evaluation Plan

The pipeline has two major pieces: a script that calls the LLM to create the harnesses, and a symbolic execution step that runs those harnesses using KLEE.

To test how well this works, we'll start testing our pipeline with known datasets like the Juliet test suite, which contains examples with intentionally planted bugs. We will evaluate whether our pipeline is effective in finding these bugs. Afterwards we will use published parser libraries to test our tool on more complex examples. For comparison, we'll also run two other tools: Vanilla KLEE, which is the basic version that uses symbolic execution, and STASE, an optimized symbolic execution tool developed in our lab.

We will evaluate all tools using three main metrics: Precision, Recall, and Overhead. The precision metric helps us see how many reported bugs are real versus false alarms. Recall tells us how many of the "ground-truth" bugs—bugs we already know are in the code—our pipeline successfully finds. The overhead of each tool is computed through the amount of time it takes to complete symbolic execution, RAM usage, and CPU usage.

We aim to have a minimum working version (MVP) by Week 9 of Winter Quarter and the start of Spring at the latest. We will start running experiments and computing metrics in Spring Quarter. To compute our metrics, we'll check:

- whether each reported bug is real or harmless,
- how many bugs each tool finds, and
- how long each tool takes to run.

We will evaluate the tools with the Juliet test suite first in Winter quarter during Weeks 7-10, because Juliet test suite's programs are simpler and clearly label bugs are present. This will simplify our initial testing process. Then, we will make some improvements to out pipeline based on our testing results. When we start running experiments in Spring Quarter, we will evaluate all three tools with the Juliet test suite again and then evaluate with more complex examples such as published parser libraries. With each experiment, we will collect the mentioned metrics and compare the results from all three tools. We will also collect information on how each tool performs with different types of vulnerabilities. Then, we will evaluate whether our tool is an improvement over already existing tools.

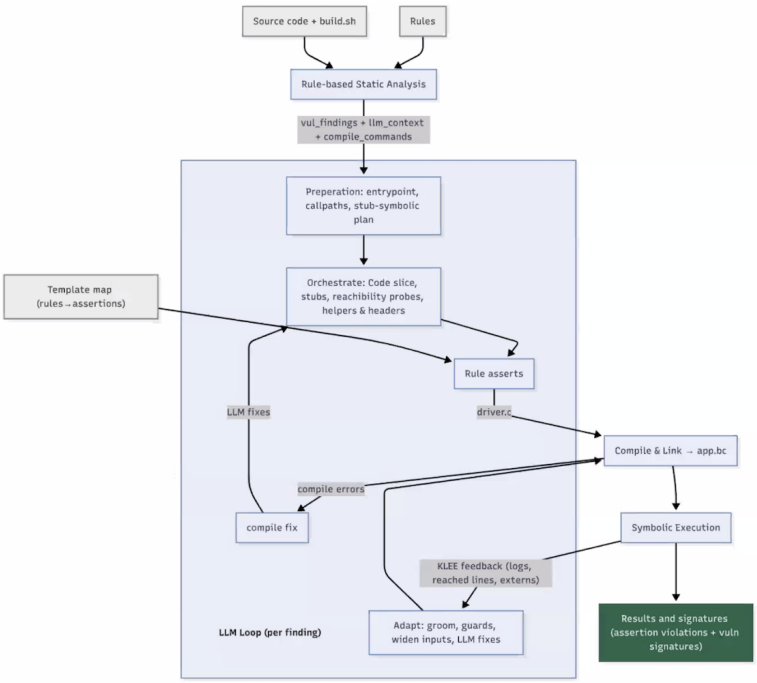

Figure 1: Evaluation Plan

4.2 Timeline

By the end of Fall quarter, we plan to finish reading the core background material and write setup notes and small case studies so we're comfortable with the tools. During the first week of Winter Quarter, we'll choose the specific types of programs we want to analyze and build the pipeline so it can symbolically execute common datasets from start to finish.

Our goal is to have the pipeline ready for real experiments—with data we can actually measure—by the end of Winter Quarter. In Spring Quarter, we'll run experiments, make small changes to improve the pipeline, and prepare our poster.

Overall, we hope to show that an LLM-assisted approach can perform at the level of existing symbolic execution tools while being easier to understand for people outside our research group. A successful outcome would have high precision, low overhead, and at least the same number of bugs reported as existing tools.

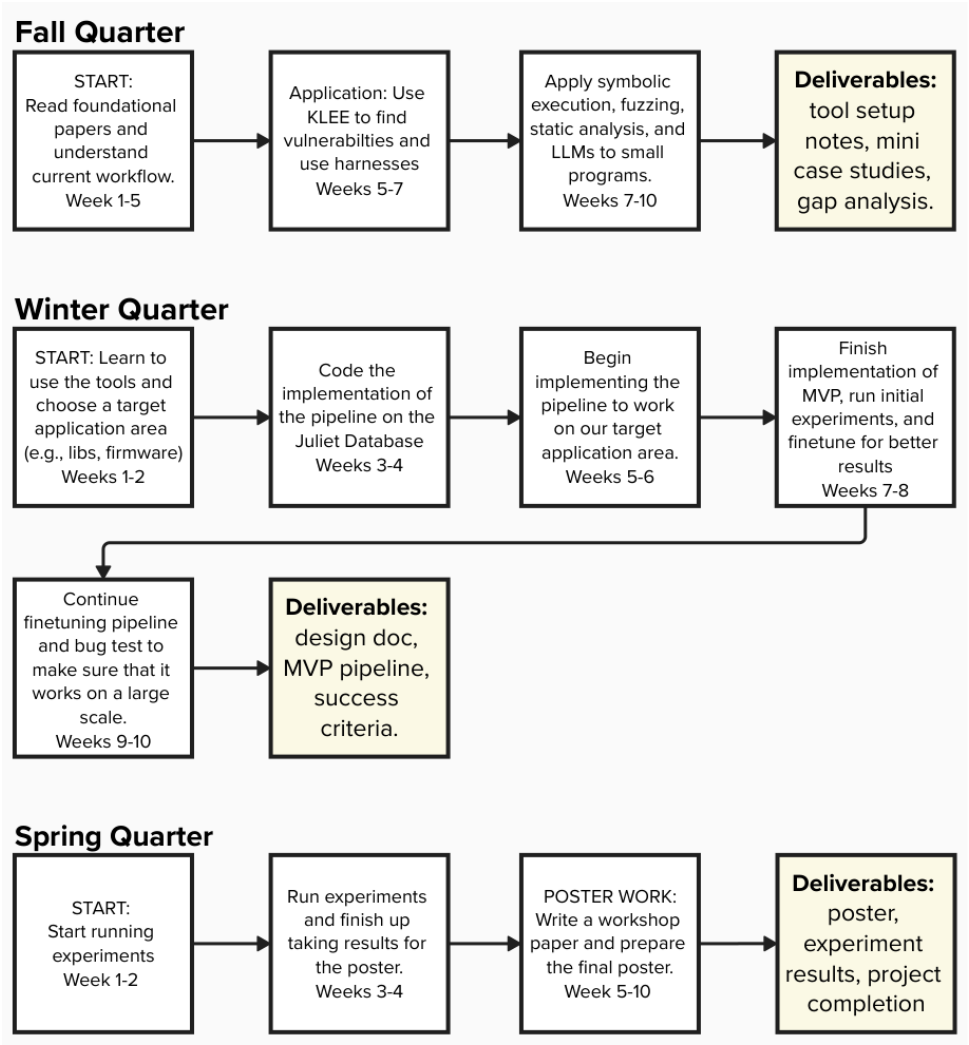

Figure 2: Timeline Plan

5 Proposal Review Feedback

- Introduction: include more paragraphs in introduction to tie high-level context with more technical details
- Literature Review: some parts felt like rough draft in related work and aren't connected to other paragraphs. We don't need to include researchers' names when writing this section.
- Evaluation Plan: Little more detail on benchmarks; what will experiment eventually be defined by (high-level)
- Timeline: more detail in timeline (include details for 2 week increments)
- Tech Details: A lot of terms aren't fully defined
- Other feedback: List advisors on title page on top; Improve overall writing style

References

[1] Joshua Ackerman and George Cybenko. Large language models for fuzzing parsers (registered report). In *Proceedings of the 2nd International Fuzzing Workshop*, FUZZING 2023, New York, NY, USA, 2023. Association for Computing Machinery.

[2] Christian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56, 2013.

[3] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 23)*, pages 423–435. ACM, 2023.

[4] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[5] IBM Security and Ponemon Institute. Cost of a data breach report 2025. https://www.ibm.com/reports/data-breach, 2025. Accessed: 2025-12-01.

[6] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. Prompt fuzzing for fuzz driver generation. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*, pages 3793–3807. ACM, 2024.

[7] Md Shafiuzzaman, Achintya Desai, Laboni Sarker, and Tevfik Bultan. Stase: Static analysis guided symbolic execution for UEFI vulnerability signature generation. *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE 24)*, page 1783–1794, 2024.

[8] Maolin Sun, Yibiao Yang, Yang Wang, Ming Wen, Haoxiang Jia, and Yuming Zhou. SMT solver validation empowered by large pre-trained language models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1288–1300. IEEE, 2023.