Large Language Model Powered Symbolic Execution

YIHE LI, National University of Singapore, Singapore
RUIJIE MENG*, National University of Singapore, Singapore
GREGORY J. DUCK, National University of Singapore, Singapore

Large Language Models (LLMs) have emerged as a promising alternative to traditional static program analysis methods, such as symbolic execution, offering the ability to reason over code *directly* without relying on theorem provers or SMT solvers. However, LLMs are also inherently approximate by nature, and therefore face significant challenges in relation to the *accuracy* and *scale* of analysis in real-world applications. Such issues often necessitate the use of larger LLMs with higher token limits, but this requires enterprise-grade hardware (GPUs) and thus limits accessibility for many users. In this paper, we propose *LLM-based symbolic execution*—a novel approach that enhances LLM inference via a path-based decomposition of the program analysis tasks into smaller (more tractable) subtasks. The core idea is to generalize path constraints using a generic code-based representation that the LLM can directly reason over, and without translation into another (less-expressive) formal language. We implement our approach in the form of AutoBug, an LLM-based symbolic execution engine that is lightweight and language-agnostic, making it a practical tool for analyzing code that is challenging for traditional approaches. We show that AutoBug can improve both the accuracy and scale of LLM-based program analysis, especially for smaller LLMs that can run on consumer-grade hardware.

CCS Concepts: • **Theory of computation** → **Program analysis**; • **Computing methodologies** → *Knowledge representation and reasoning*.

Additional Key Words and Phrases: Program Analysis; Software Verification; Symbolic execution; Large language models (LLMs); Automated reasoning; Constraint solving

ACM Reference Format:
Yihe Li, Ruijie Meng, and Gregory J. Duck. 2025. Large Language Model Powered Symbolic Execution. *Proc. ACM Meas. Anal. Comput. Syst.* 9, OOPSLA2, Article 385 (October 2025), 29 pages. https://doi.org/10.1145/3763163

1 Introduction

Program analysis is a foundational discipline in computer science that aims to understand program behavior through various systematic techniques. Traditional forms of program analysis include *static* methods—such as *symbolic execution* [40], *abstract interpretation* [19, 20], and *model checking* [17]—that analyze the program without executing it, as well as *dynamic* methods—such as *fuzzing* [51, 77], *concolic execution* [30, 62], *instrumentation* [53, 63], and *profiling* [32]—that analyze program behavior based on observations of actual executions. In general, program analysis has many applications, such as program testing [11, 18, 30, 31, 34, 56, 58, 62, 72, 80], debugging [2, 25, 76], verification [28, 35], repair [43, 46], reverse engineering [52], and vulnerability detection [39]. A recent alternative to traditional program analysis methods has emerged in the form of *Large Language Models* (LLMs) [10, 14, 45, 54, 67]. Here, LLMs can make inferences over code directly

*Corresponding author

Authors' Contact Information: Yihe Li, National University of Singapore, Singapore, Singapore, yihe.li@u.nus.edu; Ruijie Meng, National University of Singapore, Singapore, Singapore, ruijie@comp.nus.edu.sg; Gregory J. Duck, National University of Singapore, Singapore, Singapore, gregory@comp.nus.edu.sg.

ACM 2476-1249/2025/10-ART385
https://doi.org/10.1145/3763163

with properties expressed in code or natural language and have become powerful enough to handle many traditional program analysis applications. As such, many LLM-based testing [15, 50], debugging [47], and repair [74, 78, 79] tools have recently emerged.

We contrast traditional program analysis (symbolic execution) with LLM-based program analysis. Here, symbolic execution is a static program analysis method based on the idea of executing a program with *symbolic inputs/values*, typically represented as logical formulae over some underlying theories (e.g., linear arithmetic, bit vectors, and arrays). Symbolic execution systematically transforms these symbolic values (a.k.a. the *symbolic state*) based on the program's statements, effectively executing multiple *concrete* paths simultaneously. These symbolic states can then be used for various analysis tasks, such as verifying the truth of an assertion or generating test cases, using *deductive inference* with the help of some underlying *theorem prover* or *SMT solver* [21]. In contrast, LLM-based program analysis involves engineering a *prompt* (or sequence of *prompts*) that queries an LLM as an oracle. A typical prompt consists of the relevant code (or code fragments), as well as instructions explaining the analysis task expressed in natural language. The LLM effectively uses a form of *approximate inference* (based on the training data) to solve the task, rather than the strict deductive inference used by a theorem prover. As such, LLM-based program analysis is typically ad hoc and tailored to specific tasks rather than general and principled.

Both traditional and LLM-based program analysis face significant challenges in practice. For example, traditional symbolic execution has several well-known limitations [6, 12], such as the handling of *unbounded* loops, the handling of external *environment*/libraries, and the handling of memory/*heap*-manipulating programs—all of which are common-place in real-world code. We consider how KLEE [11], a prominent symbolic execution engine, treats each iteration of a loop as a separate path, leading to non-termination for unbounded loops (e.g., `while(`i < *input*`) {...}`). In contrast, LLMs have the ability to directly reason over loops, environment, and heaps, avoiding problems such as non-termination. That said, LLMs face other challenges, such as *scalability* (e.g., the 8192 *token limit* for GPT-4 [10]) and *accuracy* [27, 44] due to approximate reasoning. Recent studies also correlate the accuracy of LLMs with prompt size [44], meaning that more concise and targeted prompts generally perform better.

In this paper, our aim is to improve the accuracy and scale of LLM-based program analysis. Our first main insight is that the strengths/weaknesses of traditional and LLM-based program analysis are *complementary*, meaning that a hybrid design can help address the limitations of either approach. To this end, we propose combining the path-based decomposition of symbolic execution with approximate inference via an LLM—a.k.a. *LLM-based symbolic execution*. The core idea is a principled decomposition of the original program analysis task into smaller subtasks based on paths in the original program—helping to mitigate some of the scalability and accuracy concerns with LLMs. Our second main insight is that since LLMs are primarily trained on code, the representation of symbolic states should also be in terms of code rather than logical formulae. To this end, we propose a generic path constraint representation of the form of a *strongest post-condition* (*sp*) predicate transformer [23] over *sub-programs* derived from the original program, where each sub-program represents a path. Since we represent path constraints as ordinary code, we can use an LLM prompting *directly*, disposing of any *verification conditions* (VCs) generated by the symbolic execution process. Our approach also avoids many of the limitations associated with the translation of paths into formulae for a theorem prover (e.g., environment and heaps). Our final insight is that our path constraint representation can be *generalized* into *sets* of paths (e.g., all iterations of an unbounded loop), ensuring that LLM-based symbolic execution will always terminate.

We study LLM-based symbolic execution in both theory and practice. We show that LLM-based symbolic execution mitigates many of the limitations of both traditional and LLM-based program analysis. We have implemented our approach in the form of AutoBug—an automated LLM-based symbolic execution engine. AutoBug uses a path-based decomposition of a program analysis task into smaller (more tractable) subtasks that are suitable for LLM inference. Based on the observation that LLMs are inherently approximate oracles, we propose a lightweight design of AutoBug that is language agnostic and does not rely on any heavyweight compiler infrastructure. AutoBug is designed to be a practical program analysis tool that can be applied to code that is difficult to analyze with traditional methods. In summary, the main contributions of this paper are:

- We introduce the concept of *LLM-based symbolic execution*—a symbolic execution methodology using LLMs for *direct* reasoning over the original programming language (e.g., `C`/Java/Python), rather than indirect reasoning via translation into some formal theorem prover language. We show that our approach avoids many well-known limitations of traditional symbolic execution, such as unbounded loops, external environment, heap manipulation, etc. Furthermore, our approach uses a path-based decomposition of program analysis tasks into more concise LLM prompts, improving the accuracy and scalability of LLM inference.
- We implement our approach in the form of AutoBug—a lightweight LLM-based symbolic execution engine that supports multiple programming languages without relying on heavyweight compiler infrastructure.
- We evaluate AutoBug against various program analysis tasks for `C`/Python/Java code. We show that AutoBug improves both *accuracy* and *scale*, especially for smaller models that can run on consumer-grade hardware.

2 Motivation

2.1 Background

2.1.1 Symbolic Execution. Symbolic execution [6, 40] is an established program analysis methodology based on running the program with *symbolic values* representing *sets* of concrete inputs (rather than a single concrete input for *normal* execution). For each symbolic input, symbolic execution will systematically explore multiple execution paths of the program, allowing for applications such as error detection, verifying correctness, or finding vulnerabilities.

Symbolic execution works with *symbolic states* that are traditionally represented as a set of *variables* (e.g., $\mathtt{x}$ and $\mathtt{y}$) subject to a *path constraint* (e.g., $\mathtt{x} < \mathtt{y}$). The symbolic state represents the set of *concrete* states (e.g., $\{(\mathtt{x}, \mathtt{y}) \mid \mathtt{x} < \mathtt{y}\}$) that is reachable from some symbolic input. Symbolic execution can be defined in terms of operations over symbolic states, e.g., symbolic execution over `C`-style increment statement (`x++`) can be represented as the Hoare triple [35]:

$$\{\mathcal{P}\}\ \ \mathtt{x{+}{+}}\ \ \{\exists z : \mathtt{x} = z + 1 \land \mathcal{P}[\mathtt{x} \mapsto z]\}$$

Given a pre-condition ($\mathcal{P}$), the triple describes the resulting post-condition after the execution of the increment operation. For example, given a path constraint $\pi_{pre} = (\mathtt{x} < \mathtt{y})$, then the path constraint after the operation will be $\pi_{post} = (\exists z : \mathtt{x} = z + 1 \land z < \mathtt{y})$, or equivalently, $\pi_{post} = (\mathtt{x} \leq \mathtt{y})$. Conditional statements (e.g., **if** c **then** T **else** E **end**) are typically handled by *forking* the path constraint (π_{pre}) into separate $\pi_{then} = (\pi_{pre} \land c)$ and $\pi_{else} = (\pi_{pre} \land \neg c)$ constraints, and then continuing execution along the individual *then*- and *else*-branches. A similar strategy is used for loops (e.g., **while** c **do** B **done**). Finally, for each path through the program, a final path constraint π will be generated. This can be used to prove properties over the path, such as whether a final *post-condition* (Q) holds, i.e., whether the *verification condition* (VC) of the form ($\pi \models Q$) holds or not. Such VCs are discharged with the help of *theorem provers*, such as SMT solvers [21]. By executing sets of concrete inputs at once, symbolic execution aims to exhaustively explore the space of program behaviors—something that cannot be achieved by concrete execution alone.

Example 2.1. Consider the simple program (**if** $x > y$ **then** $z := x + 2$ **else** $z := x * y$ **end**) and the post-condition $Q = (z > y)$. Then the following VC will be generated for the **else**-path:

$$\neg(x > y) \wedge z = x \times y \qquad \models \qquad z > y \tag{VC-Else}$$

A theorem prover shows that (VC-Else) is unsatisfiable, meaning that (Q) does **not** hold for all executions of the program. □

2.1.2 Large Language Models. The emergence of *Large Language Models* (LLMs) [10, 14, 45, 54, 67] presents a new alternative for reasoning over program code. Trained on extensive datasets, including millions of lines of real-world code written in multiple languages, LLMs can reason over C/Java/Python/etc. code directly, including many traditional program analysis tasks, such as testing [15, 50], debugging [47], and repair [74, 78, 79]. For example, instead of relying on symbolic execution, LLMs can reason over code using a suitable prompt expressed in natural language, such as "*there is a bug present in the following Python code segment, please suggest the possible root causes of the bug and corresponding fixes*". Most modern LLMs can analyze the code and generate possible suggestions and patches automatically, based on the understanding of code and defects present in the training set.

Unlike traditional program analysis methods, LLMs do not aim to be perfectly *precise*. Rather, LLMs can be thought of as *approximate* oracles that are sometimes incomplete or give the wrong answer. This is because LLMs fundamentally rely on learned patterns and approximate reasoning, rather than classical deductive reasoning used by traditional theorem provers. Despite the difference, LLMs are clearly useful, with an explosion of applications for real-world analysis problems.

2.2 Limitations of Symbolic Execution

While symbolic execution has many applications (e.g., bug detection, security analysis, debugging, and program repair), it also has well-known limitations, as we briefly summarize below:

2.2.1 Limitation: Handling Loops (and Recursion). Unbounded loops (and recursion) are a known problem for symbolic execution. Here, symbolic state *forking* (Section 2.1.1) can treat each loop iteration $\{0, 1, 2, ...\}$ as a new path, potentially leading to infinite unrolling if the loop is not bounded. Popular symbolic execution tools, such as KLEE [11], handle this problem using a *concrete iteration bound*—replacing potentially infinite exploration with a bounded, but incomplete, exploration.

An alternative is to use *loop invariants*. If known, the invariant allows symbolic execution to pass over a loop without explicit unrolling. Loop invariants can be manually provided, or discovered automatically, such as using *abstract interpretation* [19] over some known domain, *constraint based* (CBMC [29]), or machine learning (Code2Inv [65]). However, loop invariant discovery may only work for simple loops, and the general case is either computationally hard or undecidable.

2.2.2 Limitation: Handling External Environment. Another known problem is the handling of external functions (e.g., calls to third-party libraries without source code) and/or external inputs (e.g., `recv` from a socket), collectively the *external environment*. Since the underlying symbolic-execution theorem prover uses *deductive reasoning*, a precise specification of all external operations and/or inputs is usually required. As such, the environment is usually handled through a combination of stubs, modeling, or *concretization*. For example, the user can manually model an external function call by implementing a replacement *stub* function that specifies the necessary specification using `klee_assume()`. However, this approach is manual, and modeling arbitrary code or inputs can require significant effort, meaning that the approach tends to rarely scale.

Another approach is concretization, where the symbolic execution algorithm assigns concrete values to some symbolic variables, allowing external functions to be executed with these values. However, concretization can also lead to an incomplete exploration of program behavior.

(a) original program

```
#define SIZE 100
typedef struct NODE {
    const char *key, *val;
    struct NODE *next;
} NODE;
NODE *db = NULL;
pthread_mutex_t mutx = PTHREAD_MUTEX_INITIALIZER;

void handle_client(int s) {
  char buf[SIZE], rep[SIZE];
  while (true) {
    ssize_t r = recv(s, buf, SIZE-1, 0);
    if (r <= 0) break;
    buf[r] = '\0';
    char cmd[SIZE] = {0}, key[SIZE], val[SIZE];
    sscanf(buf, "%s %s %s", cmd, key, val);
    if (strcmp(cmd, "GET") == 0) {
      snprintf(rep, SIZE, "ERROR\n");
      NODE *n, *p = NULL;
      pthread_mutex_lock(&mutx);
      for (n = db; n != NULL; p = n, n = n->next) {
        if (strcmp(n->key, key) != 0)
          continue;
        if (p != NULL) p->next = n->next;
        else           db = n->next;
      }
      pthread_mutex_unlock(&mutx);
      if (n != NULL) {
        snprintf(rep, SIZE, "%s\n", n->val);
        free(n->key); free(n->val); free(n);
      } else
        snprintf(rep, SIZE, "ERROR\n");
    } else if (strcmp(cmd, "PUT") == 0) {
      NODE *n = (NODE *)malloc(sizeof(NODE));
      pthread_mutex_lock(&mutx);
      n->key = strdup(key);
      n->val = strdup(val);
      n->next = db; db = n;
      pthread_mutex_unlock(&mutx);
      snprintf(rep, SIZE, "OK\n");
    } else
      snprintf(rep, SIZE, "ERROR\n");
    send(s, rep, strlen(rep), 0);
  }
  close(s);
}
```

(b) slice for "mutx is unlocked"

```
pthread_mutex_t mutx = PTHREAD_MUTEX_INITIALIZER;

void handle_client(int s) {
  while (true) {
    if (...) break;
    if (...) {
      pthread_mutex_lock(&mutx);
      pthread_mutex_unlock(&mutx);
    } else {
      pthread_mutex_lock(&mutx);
      pthread_mutex_unlock(&mutx);
    }
  }
}
```

(c) slice for "db->key ≠ NULL"

```
#define SIZE 100
typedef struct NODE {
    const char *key;
} NODE;
NODE *db = NULL;

void handle_client(int s) {
  char buf[SIZE];
  while (true) {
    ssize_t r = recv(s, buf, SIZE-1, 0);
    if (...) break;
    buf[r] = '\0';
    char cmd[SIZE] = {0}, key[SIZE], val[SIZE];
    sscanf(buf, "%s %s %s", cmd, key, val);
    if (...) {
      NODE *n = (NODE *)malloc(sizeof(NODE));
      n->key = strdup(key);
      db = n;
    }
  }
}
```

Fig. 1. An example program (a) that implements a simple key-value server. The example program includes an unbounded loop (`while (true) ...`), interaction with the external environment (`recv`/`send`), and heap-manipulating data structures (`NODE`). In addition, slices (b) and (c) corresponds to the post-conditions "`mutx` is unlocked" (always holds) and "`db->key ≠ NULL`" (may not hold) respectively.

2.2.3 *Limitation: Heap Manipulating Programs.*

Traditional theorem provers and SMT solvers tend to have limited support for reasoning over (mutable) data structures with complex structural invariants, such as *singly*- and *doubly*-linked lists, binary trees, *red-black* trees, and DAGs. By extension, traditional symbolic execution tools inherit these limitations. Some tools are based on *Separation Logic* [59], which does support reasoning over heap-manipulating programs, as used by VeriFast [36] and Infer [24]. However, VeriFast is manual and annotation-heavy, and Infer uses heuristics to infer common structural invariants, such as linked lists, but not arbitrary heap shapes.

Discussion. Figure 1 (a) is an example of a program that exhibits all three limitations, including unbounded loops (`while (true) ...`), interaction with the external environment (`recv`/`send`), and is a heap manipulating program (`NODE`). Although this program is relatively simple, it still presents a significant challenge for traditional symbolic execution tools such as KLEE.

2.3 Limitations of LLMs

LLMs are trained on a huge corpus of data, and do not necessarily have the same limitations as symbolic execution. For example, an LLM can reason over Figure 1 (a), and can answer queries about the program. That said, LLMs also have known limitations, as summarized below:

2.3.1 Limitation: Scale. LLMs typically have a limited ability to reason over large and complex code bases. For example, the *token limit* imposes a maximum number of tokens for the given input (and output), and a medium-to-large code base can easily exceed this limit.

2.3.2 Limitation: Approximate Oracles. LLMs rely on *approximate reasoning*, i.e., answers based on statistical patterns and heuristic generalization, rather than the formal deductive reasoning of traditional theorem provers. Even if the token limit (2.3.1) is not exceeded, studies have shown that LLMs generally perform worse with overly verbose prompts that include irrelevant information [44].

Discussion. Although scale and accuracy are concerns, LLMs can typically handle more classes of programs than traditional analysis methods, such as symbolic execution. Furthermore, the relative completeness of LLMs, in the absence of precise specifications, means that LLMs can be easily applied to a wide range of applications. The completeness/applicability can often be of greater pragmatic interest than perfect accuracy, and this is one reason behind the explosion of real-world applications.

2.4 Our Approach

Many limitations of traditional symbolic execution stem from those of the underlying theorem prover. Specifically, existing theorem provers and SMT solvers only accept queries in some formal input language, with limited expressiveness compared to the original source language (e.g., C/Java/Python). For example, an SMT solver will only accept queries in the form of a Boolean formula over a given set of theories (T), such as linear inequalities, bit vectors, or arrays. In contrast, the C programming language is significantly more expressive, with complex control-flow (loops), memory, pointers, library calls, data structures, environmental interactions, etc. The discrepancy in the expressiveness complicates the translation from the high-level programming language into the solver input language—such as "unrolling" loops into flat, quantifier-free formulas—and such translation may also be incomplete (e.g., unbounded loops). Furthermore, the SMT solver may not support the necessary theories for reasoning over complex programs, such as heap-manipulating data structures and external environment interactions.

Our underlying approach is to use the path-based decomposition of symbolic execution, but to replace the traditional theorem prover with an LLM. The key advantage of LLMs is that they can *reason over the source code directly*—eliminating the need for translation into a less expressive solver input language and the associated limitations. Instead, our approach represents the path constraint as a generic *strongest post-condition* $sp(S, \mathcal{P})$ predicate transformer [23] over a pre-condition $\mathcal{P}$, and a *derived sub-program* S which represents a path or set of paths. Our key insight is that LLMs can reason over *sp*-constraints directly, since S is just ordinary source code, without the need for "translation". Essentially, we can view the LLM as an effective solver for untranslated *sp*-constraints "as-is". We demonstrate this concept with a simple example.

Example 2.2 (Simple). Consider the simple program ($\textbf{if}\ \mathsf{x} > \mathsf{y}\ \textbf{then}\ \mathsf{z} := \mathsf{x} + 2\ \textbf{else}\ \mathsf{z} := \mathsf{x} * \mathsf{y}\ \textbf{end}$) and post-condition $Q = (\mathsf{z} > \mathsf{y})$ once more (see Example 2.1). Then the path constraint for the *else*-branch can be represented as a formula (φ) or a sub-program (S), as follows:

Formula (φ)	*Strongest Post-Condition over a Sub-program* (S)
$\neg(\mathsf{x} > \mathsf{y}) \land \mathsf{z} = \mathsf{x} \times \mathsf{y}$	$sp(S, true)$ where $S = \{\textbf{assume}(\mathsf{x} \leq \mathsf{y}); \mathsf{z} := \mathsf{x} * \mathsf{y}\}$

The key idea is that both (φ) and $sp(S, true)$ are **equivalent** representations of the same path constraint—i.e., one can be derived from the other under the definition of the language semantics (sp). While the formula is suitable for a theorem prover, the sub-program is suitable for an LLM:

$$\text{“}\textit{Given } \{\textbf{assume}(\mathtt{x} \leq \mathtt{y}); \mathtt{z} := \mathtt{x} * \mathtt{y}\}\textit{, does the post-condition } \mathtt{z} > \mathtt{y} \textit{ always hold?}\text{”} \qquad \text{(VC-Else-2)}$$

The LLM determines the VC does **not** hold, e.g., the following GPT4 [10] output (emphasis original):

"*The post-condition* z > y **does not always hold**. *A simple counterexample is when* x = 1*, where* z = y *instead of* z > y. *Hence, the claim is* **false**." □

Our approach is to enumerate all sub-programs based on a *partitioning* of paths through the original program. Here, each sub-program is algorithmically derived from the original code, and contains all statements that (1) are visited by any path from the partition, and (2) of which post-condition Q is data- or control-flow dependent. Like traditional symbolic execution, our approach is a path-based decomposition of the original program analysis problem into smaller (more tractable) sub-problems. This decomposition helps to address some of the limitations of direct LLM-based reasoning (Section 2.3), such as scale and accuracy. Furthermore, since our approach uses an LLM, it avoids many of the limitations of traditional symbolic execution (Section 2.2) caused by translation. We summarize the benefits as follows:

▷ *Handling Loops (and Recursion).* Our approach avoids translation of (unbounded) into a less expressive language, and loops/recursion can be represented "as-is" in the derived sub-program(s). Similarly, our approach does not need explicit loop invariant recovery or annotation, as LLMs are capable of reasoning over loops without any special intervention.

▷ *Handling External Environment.* Rather than manual modeling or concretization, our approach is to use the LLM to infer the likely behavior of the environment or external function call. Since LLMs are trained on a huge corpus of real-world code, they have significant exposure to common libraries, file formats, protocols, etc. Furthermore, even if the external environment is novel, LLMs can still infer the most likely behavior based on clues from the context (function names, variable names, code comments, placement within the algorithm, etc.), as a form of *abductive reasoning*, or *inference to the best explanation*, without the need for explicit modeling.

▷ *Heap Manipulating Programs.* LLMs can directly interpret heap-manipulating programs without the need for any special logical framework. LLMs can also (abductively) infer the (likely) structural invariants based on direct interpretation of the code, without explicit annotation.

▷ *Scale.* Like traditional symbolic execution, our approach decomposes program analysis problems into smaller (tractable) sub-problems, helping to avoid any hard or soft limit of the LLM. This allows our approach to scale to large/complex programs and analysis problems.

▷ *Approximate Oracles.* Studies [44] show that LLMs perform better with more targeted and concise prompts. By decomposing program analysis problems into sub-problems that capture only the relevant parts of the original (possibly large) code base, we help to focus the LLM and improve the overall accuracy of the analysis.

The decomposition and lack of translation mean that our approach can handle programs that are difficult for traditional program analysis. We illustrate with an example.

Example 2.3 (LLM-based Symbolic Execution)**.** Consider the Figure 1 (a) program that cannot easily be handled by traditional symbolic execution methods (Section 2.2), and a simple program analysis problem that verifies each `lock(&mutx)` operation is paired with an `unlock(&mutx)` operation. Furthermore, assume that (for the sake of example) the (a) program is too complex for an LLM to handle directly (Section 2.3).[1] We can express this as a natural language pre- and post-condition ($\mathcal{P}$

[1]This is not necessarily true, but is an assumption for the sake of an example that can fit within the page limit.

and Q respectively) that "`mutx` is unlocked". Then Figure 1 (b) is an example of a *derived sub-program* that acts as a substitute for the original program with respect to $\mathcal{P}$ and Q. We have that:

$$sp((\mathbf{b}), \mathcal{P}) \not\models Q \qquad \Rightarrow \qquad sp((\mathbf{a}), \mathcal{P}) \not\models Q$$

Thus, to refute Q for (a), it is sufficient to refute Q for (b). Furthermore, program (b) is targeted to the specific program analysis task (that `mutx` is unlocked), and is ~80% smaller in token count (419 vs 81). The LLM determines the post-condition **holds** for (b).

Another example is Figure 1 (c) and the post-condition (`db->key` ≠ `NULL`). Assuming the same condition initially holds, then (c) is ~64% smaller (419 vs 149 tokens). The LLM determines the post-condition does **not hold** for (c) since `strdup()` may return `NULL`. □

Algorithm. An overview of our LLM-based symbolic execution algorithm is summarized in Algorithm 1. Here, the algorithm's *front-tend* is similar to that of a standard compiler, and parses the program into an *Abstract Syntax Tree* (AST, line 2), and then generates a *Control Flow Graph* (CFG, line 3). Next, the algorithm generates a representation of the set of all *paths* through the CFG (line 4). Here, the set of all paths is represented as a set of path *partitions* such that ($paths = \Pi_1 \cup ... \cup \Pi_n$), where each partition Π_i represents some (possibly infinite) subset of paths. One challenge is how to generate a *good* set of partitioning (to be discussed in Section 3). Next, for each partition Π, the algorithm generates a derived sub-program S that generalizes the partition (line 6). For example, Figure 1 (b) and (c) are possible derived sub-programs of Figure 1 (a). Each sub-program S is used to construct a corresponding prompt (line 7), including *rendering* the S back into a text-based source-code representation (line 8). The prompt (line 7) is a natural language representation of the Hoare triple $\{\mathcal{P}\}S\{Q\}$, which holds iff $sp(S, \mathcal{P}) \models Q$. Finally, the prompt is sent to the LLM for inference (line 10). Algorithm 1 can either establish or refute the post-condition, subject to the reasoning capabilities of the underlying LLM. Assuming that the partitions are ordered based on slice, Algorithm 1 will return the *least* sub-program S that is deemed to refute the post-condition. Otherwise, if no such refutation is found, Algorithm 1 deems that the triple holds (`HOLDS`).

```
   Input: A Hoare triple {$\mathcal{P}$}$C${$Q$}
   Output: HOLDS or a counterexample $S$
1  Function LLMSymExe({$\mathcal{P}$}$C${$Q$}):
2      $AST$ ← Parse($C$)
3      $CFG$ ← GenCFG($AST$)
4      $partitions$ ← GenPartitions($CFG$)
5      for $\Pi \in partitions$ do
6          $S$ ← GenSubProg($\mathcal{P}$, $\Pi$, $Q$)
7          $prompt$ ← "assuming $\mathcal{P}$" ++
8                      Render($S$, $AST$) ++
9                      "does $Q$ hold?"
10         $result$ ← LLM($prompt$)
11         if $result$ = FALSE then return $S$
12     return HOLDS
```

Algorithm 1: LLM-based Symbolic Execution

Summary. Like traditional symbolic execution, LLM-based symbolic execution (Algorithm 1) represents a path-based *decomposition* the original program analysis task into smaller subtasks. A summary of the main similarities and differences is shown in Table 1. The substitution of a theorem prover with an LLM changes to various aspects of the design, capabilities, and implementation of the symbolic execution engine. For example, LLMs use *approximate* and *abductive* reasoning, or rely on information learned during the training process, meaning that LLMs do not need precise specifications or environment modeling. Likewise, the lack of translation into some (less expressive) formal language allows the LLM to reason over loops or heap manipulation "as-is", without relying on loop/data-structure invariant discovery.

In this paper, we study the concept of program analysis via LLM-based symbolic execution. First, we study the *principles* of LLM-based symbolic execution in terms of the idealized procedural

Table 1. Summary of main similarities and differences between traditional and LLM-based symbolic execution. Here (*Cap* = *Capabilities*), (*Imp* = *Implementation*), (*sp* = *strongest post-condition*), and (the key differences).

	Property	*Tradition Symbolic Execution*	*LLM-based Symbolic Execution*
Design	*Overall method*	Decomposition & solving path constraints	Decomposition & solving path constraints
	Decomposition method	Decomposition into a formal language	Decomposition into sub-programs
	Reasoning engine	Theorem prover or SMT solver	Large Language Model (LLM)
	Reasoning method	Deductive	Approximate, deductive, abductive
	Path representation	Formal language (unfolded *sp*-path constraints)	Untranslated *sp*-constraints over truncated slices
	Specification language	Formal language	Either formal, code, or natural language
Cap.	*Weaknesses*	Loops, environment, heaps	Complex integer, linear, Boolean reasoning
	Unbounded loops	Infinite unfolding or loop invariants	LLM reasons over loops "as-is"
	Environment	Manual modeling/specifications required	LLM abductive reasoning or training data
	Heap manipulation	Manual annotation + Separation Logic	LLM reasons over heaps "as-is"
Imp.	*Programming languages*	Language specific (C+KLEE [11] and Java+SPF [57])	Programming language agnostic
	Compiler infrastructure	Close integration (LLVM [42]+KLEE, etc.)	Lightweight (AST-level) implementation

programming language used by Hoare logic [35]. We show that program analysis tasks can be decomposed into tasks over derived sub-programs representing paths, or sets of paths (truncated slices), through the original program. Furthermore, we also show that only a *finite* number of *partitions* (Algorithm 1, line 4) needs to be considered, ensuring that LLM-based symbolic execution will always terminate—even for unbounded loops.

In addition, we study the application of LLM-based symbolic execution in *practice*. For this, we design AutoBug—an LLM-based symbolic execution engine for real-world programming languages such as C/Java/Python. Our approach is based on the observation that Algorithm 1 is mostly language agnostic except for specific aspects, such as the parser. This means our approach can be readily ported to other programming languages. Furthermore, we also observe that, since LLMs are fundamentally approximate, we can build a *lightweight* implementation that uses *approximate* parsing and dependency analysis—without relying on any heavyweight and/or language-specific compiler framework.

3 Principles of LLM-based Symbolic Execution

Our goal is to adapt traditional symbolic execution methods to LLMs that reason over code directly, rather than translation into a (less expressive) theorem prover input language.

3.1 Symbolic Execution Foundations

We use the *minimal imperative language* defined by Hoare logic [35] augmented with an explicit **assume**-statement.[2] We define the language syntax as follows:

$$C ::= \textbf{skip} \mid C;C \mid \textbf{assume}(B) \mid x := E \mid \textbf{if } B \textbf{ then } C \textbf{ else } C \textbf{ end} \mid \textbf{while } B \textbf{ do } C \textbf{ done}$$

Where E represents some base language (e.g., arithmetic expressions) and B represents Boolean expressions over E. We also use ϵ to sometimes represent an empty program (equivalent to **skip**). The language semantics are defined inductively (i.e., least relation) in terms of the *strongest post-condition* (*sp*) relation defined as follows:

$$sp(\textbf{skip}, \mathcal{P}) = \mathcal{P} \quad sp(\{C_1; C_2\}, \mathcal{P}) = sp(C_2, sp(C_1, \mathcal{P})) \quad sp(\textbf{assume}(b), \mathcal{P}) = b \wedge \mathcal{P}$$
$$sp(x := e, \mathcal{P}) = \exists y : x = e[x \mapsto y] \wedge \mathcal{P}[x \mapsto y]$$
$$sp(\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ end}, \mathcal{P}) = sp(C_1, b \wedge \mathcal{P}) \vee sp(C_2, \neg b \wedge \mathcal{P})$$
$$sp(\textbf{while } b \textbf{ do } C \textbf{ done}, \mathcal{P}) = sp(C; \textbf{while } b \textbf{ do } C \textbf{ done}, b \wedge \mathcal{P}) \vee (\neg b \wedge \mathcal{P})$$

[2]Note **assume**(B) can be emulated as (**while** $\neg B$ **do skip done**) under partial correctness, but is treated as a special case.

We define a *linear program* to be any program comprising only **skip**-, **assume**-, and *assignment*-statements (without conditionals or loops). We define a *path* to be a linear program that is derived by unfolding the original program C using the following rules:

$$unfold(\textbf{skip}, \pi) = \{\pi; \textbf{skip}\} \qquad unfold(\textbf{assume}(b), \pi) = \{\pi; \textbf{assume}(b)\} \qquad unfold(x := e, \pi) = \{\pi; x := e\}$$

$$unfold(\{C_1; C_2\}, \pi) = \cup\{unfold(C_2, \pi') \mid \pi' \in unfold(C_1, \pi)\}$$

$$unfold(\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ end}, \pi) = \cup \begin{cases} unfold(C_1, \{\pi; \textbf{assume}(b)\}) \\ unfold(C_2, \{\pi; \textbf{assume}(\neg b)\}) \end{cases}$$

$$unfold(\textbf{while } b \textbf{ do } C \textbf{ done}, \pi) = \cup \begin{cases} unfold(\{C; \textbf{while } b \textbf{ do } C \textbf{ done}\}, \{\pi; \textbf{assume}(b)\}) \\ \{\pi; \textbf{assume}(\neg b)\} \end{cases}$$

We abstractly define *symbolic execution* to be any algorithm that combines *path unfolding* with *Verification Condition* (VC) *solving*. Given a program analysis task represented as a Hoare [35] triple $\{\mathcal{P}\}C\{\mathcal{Q}\}$, then a *symbolic execution* algorithm:

(1) exhaustively *generates* the set of all $paths = unfold(C, \epsilon)$ through the program; and
(2) *solves* each corresponding VC ($sp(path, \mathcal{P}) \models \mathcal{Q}$) for each $path \in paths$.

That is, a symbolic execution algorithm computes the following using explicit enumeration:

$$\{\mathcal{P}\}C\{\mathcal{Q}\} \qquad \text{iff} \qquad \bigwedge\{sp(\pi, \mathcal{P}) \models \mathcal{Q} \mid \pi \in unfold(C, \epsilon)\} \tag{SymExe}$$

The triple is deemed to *hold* if each individual VC holds for the corresponding path (π), and to *not hold* otherwise. Here, each $sp(\pi, \mathcal{P})$ is defined to be the *path condition* for the corresponding path π. Essentially, a symbolic execution algorithm decomposes the original program analysis task into simpler subtasks that can be solved separately.

3.1.1 Traditional Symbolic Execution. "Traditional" symbolic execution algorithms solve each VC using a suitable *theorem prover*, such as an SMT solver [21]. To do so, the *sp*-constraints for each path are *translated* into a logical formula φ over the domain of E, by applying the (linear program subset of the) *sp*-rules defined above. The translated VC ($\varphi \models \mathcal{Q}$) is then solved using a theorem prover. Note that the translation is necessary since traditional theorem provers are limited to a specific input language (e.g., SMT-LIB), and cannot make inferences over an abstract *sp*-constraint directly. In addition, most practical symbolic execution tools implement several optimizations, including incremental path unfolding, incremental *sp*-translation, *pruning* infeasible paths, and the *merging* of similar paths. For example, rather than pre-computing the set of paths upfront, most practical implementations maintain a *symbolic state* comprising a current location, a (partially constructed) path constraint/condition, and a set of current variables-of-interest. Such optimizations are consistent with abstract symbolic execution (SymExe) defined above.

We can also understand the limitations of Section 2.2. Firstly, the number of paths in the set $unfold(C, \epsilon)$ may be very large (exponential) or even infinite (unbounded loops, Section 2.2.1). This is known as the *path explosion* problem, and is a well-known limitation of symbolic execution methods. Secondly, the *sp*-translation will be limited for programs that contain operations that are not supported—such as common interactions with the external environment (Section 2.2.2). Finally, the underlying theorem prover itself may be limited. For example, KLEE [11] uses the Z3 [6] SMT solver over the domain of linear arithmetic, arrays, and bit-vectors by default. However, this configuration does not support reasoning over heap-manipulating programs (Section 2.2.3).

3.2 LLM-based Symbolic Execution (Path-based)

The basic idea behind "LLM-based" symbolic execution is to use a *Large Language Model* (LLM) as the underlying reasoning engine instead of a theorem prover. Thus, given a VC of the form

(a) simple program (C_{simple})

```
i := 1;
while i ≤ n do
    read(x);
    if x < 0 then
        xs.delete(−x);
    else
        xs.insert(x);
    z := xs.size();
    write(z);
    i := i + 1;
```

(b) path (π_{simple})

```
i := 1;
assume(i ≤ n);
read(x);
assume(¬(x < 0));
xs.insert(x);
z := xs.size();
write(z);
i := i + 1;
assume(i ≤ n);
read(x);
assume(¬(x < 0));
xs.insert(x);
z := xs.size();
write(z);
i := i + 1;
assume(¬(i ≤ n));
```

(c) truncation (T_{simple})

```
i := 1;
while i ≤ n do
    read(x);
    assume(¬(x < 0));
    xs.insert(x);
    z := xs.size();
    write(z);
    i := i + 1;
```

(d) slice (S_{base})

```
i := 1;
while i ≤ n do
    read(x);
    xs.insert(x);
    i := i + 1;
```

Fig. 2. (a) A simple example program (C_{simple}), (b) one example path (π_{simple}) through (C_{simple}), (c) a truncation (T_{simple}) of (C_{simple}) assuming only the inner *then*-branch is taken, and (d) a slice (S_{simple}) of (T_{simple}) assuming $\{\mathsf{n}, \mathsf{xs}\}$ are the vars-of-interest.

($sp(\pi, \mathcal{P}) \models \mathcal{Q}$), we can use the LLM to reason directly over the path constraint and post-condition $\mathcal{Q}$. This is possible since, through its training process, the LLM can interpret both the syntax of the path π (represented as ordinary code), as well as the language semantics represented by the *sp*-rules. We illustrate with the following example.

Example 3.1 (Path-based LLM-based Symbolic Execution)**.** We consider the simple example shown in Figure 2 (a), which is based on Example Program 3 in [41] . We assume that xs is a data structure implementing a multi-set, and *insert*/*delete*/*size* can be expanded to sub-programs with a suitable multi-set implementation (e.g., a singly linked list), and is initially empty. Furthermore, we assume that the (**read**) operation always returns a positive number, which can be expressed in natural language or as a formal rule ($\forall \mathsf{x}, \mathcal{R} : sp(\mathbf{read}(\mathsf{x}), \mathcal{R}) \longrightarrow \mathsf{x} > 0$). Under these assumptions, the final size of xs should be equal to n. We can express this as the following triple:

$$\{\mathsf{n} \geq 0 \wedge \mathsf{xs}.size() = 0 \wedge \text{“}\mathbf{read}(\mathsf{x}) \text{ always returns a positive number”}\}\ C_{simple}\ \{\mathsf{xs}.size() = \mathsf{n}\} \qquad \text{(TRIPLE)}$$

Although conceptually simple, the Figure 2 (a) program is challenging for several reasons, including an unbounded loop, environmental input (**read**), and data-structure reasoning (xs). We may prove (TRIPLE) by enumerating paths, such as (π_{simple}) from Figure 2 (b), representing two iterations of the loop. We can encode the VC ($sp(\pi_{simple}, \mathcal{P}) \models \mathcal{Q}$) as a prompt, which is confirmed using an LLM:

“*Given the pre-condition $\mathcal{P}$ and the code π_{simple}, does the post-condition $\mathcal{Q}$ hold?*” □

This example shows how LLMs can solve tasks that are challenging for traditional program analysis methods. That said, a simple path-based decomposition still inherits some limitations. Firstly, there can still be an infinite number of paths (i.e., path explosion), leading to non-termination. Secondly, each individual path could still be too long for the LLM to effectively reason over (e.g., path π_{simple} from Figure 2 (b) can be generalized to any length). Finally, paths can accumulate irrelevant statements, such as variable z, that can also exacerbate the path length problem.

3.3 LLM-based Symbolic Execution (Slice-based)

To address the path-explosion problem, our core idea is to merge individual *sp*-constraints (representing individual paths) into *generalized sp-constraints* representing (possibly infinite) *sets of paths*. Given a *partition* $\Pi \subseteq unfold(C, \epsilon)$, our approach constructs a *truncated sub-program* T_Π such that:

$$sp(\pi, \mathcal{P}) \models sp(T_\Pi, \mathcal{P}) \qquad \text{for all } \pi \in \Pi \tag{Generalization}$$

Thus, instead of disposing of a (possibly infinite) number of verification conditions (VCs) of the form ($sp(\pi, \mathcal{P}) \models Q$) for each $\pi \in \Pi$, our approach disposes of a single VC of the form ($sp(T_\Pi, \mathcal{P}) \models Q$) for the entire set ($\Pi$). First, we shall present a method for constructing a truncated sub-program for a given partition. Next, we shall present a partitioning algorithm that needs only consider a finite number of subsets, even for programs with infinite paths (unbounded loops), ensuring the symbolic execution algorithm always terminates.

3.3.1 Construction. We consider the construction of truncated sub-programs.

Truncation. Given a program C and a (possibly infinite) subset of $\Pi \subseteq unfold(C, \epsilon)$, we derive a *truncated sub-program* T_Π equivalent to C for all $\pi \in paths$, and is *unreachable* otherwise:

$$\Pi \quad \subseteq \quad unfold_{reachable}(T_\Pi, \epsilon) \quad \subseteq \quad unfold(C, \epsilon) \tag{Truncation}$$

Here, ($unfold_{reachable}$) excludes all paths that terminate abnormally via a special **assume**(0) (i.e., *assume false*) statement. Thus, **assume**(0) represents a statement that is assumed to be *unreachable*.[3] Consider all statements (s) in C that are *not covered* by any path $\pi \in \Pi$, then we can construct T_Π by replacing all such (s) with **assume**(0). We can represent this idea using the following rewrite rule:[4]

$$s \longrightarrow \textbf{assume}(0) \qquad \text{if } s \notin \pi \text{ for all } \pi \in paths \tag{Unreachable}$$

We may also apply the following rules to further simplify the resulting sub-program:

$$\textbf{assume}(0); C_2 \longrightarrow \textbf{assume}(0) \qquad C_1; \textbf{assume}(0) \longrightarrow \textbf{assume}(0)$$

$$\textbf{if } b \textbf{ then } C_1 \textbf{ else assume}(0) \textbf{ end} \longrightarrow \textbf{assume}(b); C_1 \qquad \textbf{if } b \textbf{ then assume}(0) \textbf{ else } C_2 \textbf{ end} \longrightarrow \textbf{assume}(\neg b); C_2$$

$$\textbf{while } b \textbf{ do assume}(0) \textbf{ done} \longrightarrow \textbf{assume}(\neg b)$$

These rules preserve the (Truncation) property while also reducing the size (token count) of the resulting T_Π, which is beneficial for LLM prompting.

Slicing. We can further reduce the size of the truncated sub-program using *program slicing* [71]. We define a *slice* (S_Π) to be a sub-program derived from T_Π by *deleting* (i.e., replacing by ϵ) any statement (s) in T_Π that does not violate the condition:

$$sp(T_\Pi, \mathcal{P}) \models Q \qquad \text{iff} \qquad sp(S_\Pi, \mathcal{P}) \models Q \tag{Slice}$$

Our main result is as follows. If ($sp(S_\Pi, \mathcal{P}) \models Q$) holds, then:

(1) ($sp(T_\Pi, \mathcal{P}) \models Q$) holds by (Slice); and
(2) ($sp(\pi, \mathcal{P}) \models Q$) for all $\pi \in \Pi$ holds by (1) and (Generalization).

In other words, the single VC ($sp(S_\Pi, \mathcal{P}) \models Q$) is sufficient to dispose of the entire partition (Π). We can apply standard slicing methods, such as Weiser's back slicing algorithm [71] (with $vars(Q)$ as the slice criterion), to construct S_Π from T_Π. The back slicing algorithm traverses the *Control Flow Graph* (CFG), and deletes any statement that is not data- or control-flow-dependent on $vars(Q)$. The slicing algorithm is illustrated in Algorithm 2.

Example 3.2 (Slice-based Symbolic Execution)**.** We consider Example 3.1 once more. Here, the truncated slice (S_{simple}) in Figure 2 (d) is a generalization of the infinite partition (Π) representing *all* feasible paths through the **while**-loop, including the Figure 2 (b) path (π_{simple}). An LLM can be used to verify the generalized *verification condition* (VC) encoded in natural language.

[3]Similar to `__builtin_unreachable()` from `gcc`.
[4]We use the standard notation ($lhs \longrightarrow rhs$) to mean that any term matching the lhs is rewritten to the term matching rhs.

"*Given the pre-condition $\mathcal{P}$ and the code S_{simple}, does the post-condition Q hold?*"

Path-based symbolic execution will infinitely unroll the **while**-loop, generating a new VC for each loop iteration. In contrast, slice-based symbolic execution requires only a single VC to be checked. Furthermore, (S_{simple}, 5 lines, 28 tokens) is simpler and more compact than all of (C_{simple}, 10 lines, 56 tokens), (π_{simple}, 16 lines, 85 tokens), and (T_{simple}, 8 lines, 48 tokens). Truncation and slicing are useful for removing irrelevant statements while preserving paths, meaning that the corresponding prompt is simpler and more concise, thereby improving the accuracy of LLMs. □

Discussion. Our approach is related to *path merging* and *loop invariants* in traditional symbolic execution. Here, given a set of n *path constraints*, represented as translated logical formulae φ_i, $i \in 1..n$, the idea is to find a formula ϕ that is *generalization* ($\varphi_i \models \phi$). Thus, the n verification conditions ($\varphi_i \models Q$) can be combined into a single verification condition ($\phi \models Q$), helping to mitigate the path explosion problem. Similarly, the (possibly infinite) set of path constraints ϱ_i through a loop can be generalized into a *loop invariant* I, such that ($\varrho_i \models I$), allowing the loop to be handled without infinite unrolling. However, loop-invariant discovery over formulae is difficult and undecidable in the general case. In contrast, our approach avoids the problem, since the *sp*-constraints are never translated into a different representation.

```
Input: A CFG sub-graph $G$; reverse topological order
Output: A $slice \subseteq G$
Function GenSlice($G$, $Q$):
    $slice \leftarrow \emptyset$; $vars \leftarrow$ getVars($Q$)
    for $s \in G$ do
        if $s$ modifies any $v \in vars$, or
          conditional $s$ reads any $v \in vars$ then
            $slice \leftarrow slice \cup \{s\}$
            $vars \leftarrow vars \cup$ getDependencies($s$)
    end
    return $slice$
```

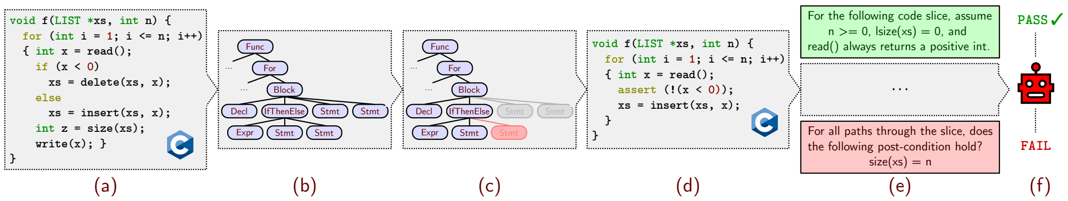
Algorithm 2: Basic back-slicing algorithm.

3.3.2 Partitioning. Section 3.3.1 describes the construction of a *truncated slice* S given a partition (Π) that subset of paths ($\Pi \subseteq unfold(C, \epsilon)$). In principle, any partitioning ($\Pi_1 \cup ... \cup \Pi_n = unfold(C, \epsilon)$) can be used. However, a *good* partitioning aims to minimize the slice (S_i) *size* for each Π_i, $i \in 1..n$, in order to simplify the prompts ultimately sent to the LLM. Our approach is to construct a partitioning based on *path coverage*.

```
Input: The Control-Flow Graph (CFG)
Output: A coverage-based partitioning
Globals: Coverage map ($covMap$), end node ($end$)
Function GenPartitions($node, pathCov, path$):
    if $pathCov \in covMap$ then return $\emptyset$
    $pathCov \leftarrow pathCov \cup \{node\}$
    $path \leftarrow path +\!\!+ node$
    $covMap \leftarrow covMap \cup \{path\}$
    if $node = end$ then return $\{path\}$
    $partitions \leftarrow \emptyset$
    for $succ \in successors(node)$ do
        $partitions \leftarrow partitions \cup$
            GenPartitions($succ, pathCov, path$)
    return $partitions$
```

Algorithm 3: Path partitioning algorithm

Path Coverage. Under Section 3.1, we define a *path* (π) to be a *linear program* over statements or branches (represented as assertions) from C. Here, we define a *Control Flow Graph* (CFG) *path* to be a sequence of nodes (a.k.a. locations) $\langle l_1, ..., l_m \rangle$ through the CFG representation of C. Given a CFG path (π), we define *path coverage* to be the set ($cov(\pi) = \{l_1, ..., l_m\}$), i.e., the reinterpretation of (π) from a sequence to a set. It is common for distinct paths to have the same coverage; for example, different iterations of the same loop have distinct sequences, but produce equivalent sets (i.e., covering the same nodes). We make two key insights. First, paths with distinct coverage will have distinct truncations, since each path must



```
void f(LIST *xs, int n) {
  for (int i = 1; i <= n; i++)
  { int x = read();
    if (x < 0)
      xs = delete(xs, x);
    else
      xs = insert(xs, x);
    int z = size(xs);
    write(x); }
}
```

```
void f(LIST *xs, int n) {
  for (int i = 1; i <= n; i++)
  { int x = read();
    assert (!(x < 0));
    xs = insert(xs, x);
  }
}
```

For the following code slice, assume n >= 0, lsize(xs) = 0, and read() always returns a positive int.

...

For all paths through the slice, does the following post-condition hold? size(xs) = n

Fig. 3. Example workflow. Here, (a) is the original source code, (b) is the *Abstract Syntax Tree* (AST) parsed from the code, (c) is a generated path slice (Algorithm 3 and Algorithm 2), (d) is the slice *rendered* back into the original source code, and (e) is the generated *prompt* comprising a pre-condition, the slice, and the post-condition, and (f) is the LLM inference.

differ by at least one location. This will be used as the basis for partitioning. Secondly, the number of distinct paths w.r.t. coverage is *finite*, meaning that LLM-based symbolic execution (slice-based) necessarily terminates—even for unbounded loops.

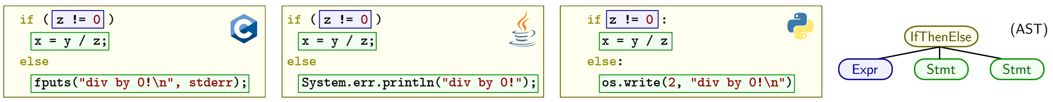
Partitioning Algorithm. The core idea is to enumerate CFG paths (π_{cov}) with distinct coverage (cov). Each partition ($\Pi_{cov} \subseteq unfold(C, \epsilon)$) is implicitly defined as all paths with the same path coverage as (cov). The algorithm uses a *Depth First Search* (DFS) exploration over the CFG, as illustrated in Algorithm 3. Here, the algorithm takes a CFG representation of C, and generates a *partitioning* as a set of *representative paths* (π_{cov}) for each distinct coverage class (cov). The algorithm works as a standard DFS path construct algorithm, but also maintains a *coverage map* ($covMap$) that tracks previously-seen coverage prefixes. When a ($node \in CFG$) is visited, the coverage map is consulted, and the path is *pruned* if the prefix has been observed before. The coverage map ensures that Algorithm 3 both (1) terminates, and (2) only returns paths (partitions) that differ by at least one location/node. Each output path (π_{cov}) corresponds to a partition (Π_{cov}), which is then used to construct a corresponding truncated sub-programs (T_{cov}) and slice (S_{cov}) that is used for LLM prompting.

Given two distinct coverages (cov, cov'), the resulting slices (i.e., S_{cov}, $S_{cov'}$) can sometimes be equivalent ($S_{cov} = S_{cov'}$) if all distinguishing nodes are removed by the slicer. Furthermore, some truncated slices may generalize others, i.e., ($sp(S_{cov}, \mathcal{P}) \models sp(S_{cov'}, \mathcal{P})$). For example, under Algorithm 3, then $S = \langle \mathtt{i} := 1; \mathbf{assume}(\neg(\mathtt{i} \le \mathtt{n}))\rangle$ is also a valid slice of (C_{simple}, Figure 2) representing zero iterations of the **while**-loop. Nevertheless, our LLM-based symbolic execution algorithm orders VCs by size, in order to find a *least* refutation of the post-condition ($\mathcal{Q}$).

Discussion. Algorithm 3 is guaranteed to *terminate*, since the number of possible distinct coverage sets is finite. Thus, unlike path-based symbolic execution, slice-based symbolic execution will always terminate, even for programs with unbounded loops. For example, the truncated slice (S_{simple}) from Figure 2 (b) generalizes infinitely many paths through the loop (for any number of iterations). Furthermore, the size of each slice S is always bounded by the size of the original program C, whereas unrolled paths can exceed any length.

4 LLM-based Symbolic Execution in Practice

The truncated slice-based symbolic execution algorithm of Section 3 is defined for an idealized programming language (the language of Hoare triples). In this section, our aim is to port our abstract approach to real-world programming languages (e.g., C/Java/Python). Our first main insight is that the main symbolic execution and slicing algorithms are *agnostic* to the target programming language—provided there is a method for parsing the source code into some suitable representation for path generation and slicing. Our second main insight is that, since LLMs are *approximate*



```
if (z != 0)
  x = y / z;
else
  fputs("div by 0!\n", stderr);
```

```
if (z != 0)
  x = y / z;
else
  System.err.println("div by 0!");
```

```
if z != 0:
  x = y / z
else:
  os.write(2, "div by 0!\n")
```

Fig. 4. Example AST representation of an *if-then-else* statement across the C, Java, and Python programming languages. Each AST node is represented *generically* by the *file range* from which the element was parsed, as illustrated by the nested colored boxes.

oracles, the greater workflow can also be approximate—meaning that each individual step need not be perfectly precise, provided the overall accuracy is not significantly degraded. These insights significantly simplify our overall design.

An example of the workflow is illustrated in Figure 3. First, our workflow uses a lightweight parsing framework (specifically, `tree-sitter` [9]) to parse the source code (a) into an *Abstract Syntax Tree* (AST) representation (b). From the AST, a *Control-Flow Graph* (CFG) is constructed, allowing for the finite enumeration of all path partitions using Algorithm 3. For each enumerated partition, a *truncation* and *slice* is constructed (c), by removing statements that are not control- or data-flow dependent on the post-condition Q, using Algorithm 2. Next, the slice is *rendered* back into the original source language (d). Essentially, this means emitting sliced path elements, deleting (i.e., not emitting) non-sliced path elements, and replacing (i.e., replace by `assume(0)`) all other non-path elements. The resulting slice generalizes a (possibly infinite) set of paths through the program. Finally, the slice is used to construct a text-based prompt (e) used to query an LLM (f). steps (c)-(f) are repeated for each partition using Algorithm 1, until exhaustion or the discovery of a counterexample.

4.1 Partitioning and Truncated Slice Generation

Parsing. We use the `tree-sitter` framework [9] to parse the source code into an AST. Each AST node comprises a node *type* representing some syntactic element, the *file range* (*source file* and *offset* range) from which the element was parsed, and zero or more children of sub-elements. The AST is *unified* so that the shared language features are mapped to the same representation, as illustrated in Figure 4. Although `tree-sitter` does not guarantee perfect parsing, this is tolerable under our approximate design. Our approach also avoids heavyweight and specialized compiler frameworks necessary for precise parsing (e.g., `clang` [42] for C, `javac` [4] for Java, and CPython [68] for Python), significantly simplifying the implementation.

CFG Construction. The next step is to *lower* the AST into a *Control Flow Graph* (CFG), where each node represents a statement or a condition, followed by one (or more) control-flow edges to successor nodes. The CFG is language agnostic, and common language features (e.g., *if-then-else*, *while-loops*, etc.) are lowered into common CFG patterns.

Slice generation. The path partitioning is generated using Algorithm 3. For each partition Π, a corresponding truncated sub-program T_Π is generated by substituting non-covered nodes/sub-programs with an `assume(0)` statement, followed by simplification. From this, a truncated slice S_Π is generated by applying Weiser's back-slicing algorithm (as illustrated in Algorithm 2). Here, the variables from the post-condition (Q) are used as the slicing criterion, and Algorithm 2 removes all nodes from T_Π of which Q is not control- or data-flow dependent.

4.2 Truncated Slice Rendering

So far, each slice (S_Π) is represented as a *sub-graph* of the (truncated) CFG. Our ultimate target is an off-the-shelf LLM, meaning that the slice must be *rendered* back into a generic text form. For this, we leverage the underlying AST representation based on *file ranges*. For the rendering algorithm, we use an *interval tree* (I) mapping ranges to strings. For each CFG node (and the corresponding AST statement s), we insert all $s \in S_\Pi$ into I.

Context. We also include the AST path from the root down to each sliced (s), such as the enclosing *function*, control-flow context (*if-then-else*, *while-loops*, etc.), or *class definitions* (for Java). This ensures that the rendered slice is coherent (reparsable) code, rather than a collection of isolated statements. Furthermore, it is often useful to include all relevant *non-executable* AST nodes from which each s depends, including *variable declarations*, *type declarations*, *member declarations*, *function declarations*, *global declarations*, *macro definitions*, etc. Such dependencies are matched based on the AST name. For example, given the executable statement (`xs = insert(xs, x)`), then any function declaration or macro definition matching the name "`insert`" can be included in the context. Since we rely on lightweight parsing and not a compiler front-end with semantic information, name-based matching may over-approximate dependencies. However, this is allowable under our approximate workflow, and the LLM will often ignore irrelevant context.

The slice renderer also preserves the original formatting, including the preceding and succeeding whitespace and comments. Preserving such information is not strictly necessary for `C` and Java, but is essential for whitespace-sensitive languages such as Python. Furthermore, source-code comments can provide additional context, such as the intent of programmers, which can assist LLM inference.

Example. An example rendered slice is illustrated in Figure 3 (d). Here, all sliced executable statements are preserved, as well as the enclosing context (e.g., function declaration for `f()`). Furthermore, the inner *if-then-else* has been simplified to an assertion, and the formatting (whitespace) has been preserved. The resulting slice is coherent `C` code in its own right. Crucially, each path through the slice corresponds to an equivalent path in the original programming, meaning that any inference on the slice is also a valid inference for the original code.

LLM Inference. Once the rendered slice has been generated, the final step is prompt construction and LLM inference. This step can be highly customized, but for our basic design, we use a prompt structure that mirrors a Hoare triple $\{\mathcal{P}\}S\{Q\}$ where $\mathcal{P}$ is a pre-condition, Q is a post-condition, and S is the generated slice (see Example 2.3, Example 3.2, and Figure 3 (e)). In addition, some basic instructions for the LLM are provided, such as the output format. Here, we assume that the LLM will generate one of two possible responses to the prompt, namely `PASS` (the post-condition holds) or `FAIL` (the post-condition does not hold).

4.3 Implementation

We have implemented our workflow in the form of the AutoBug tool. AutoBug takes as input a program C in a supported programming language (currently `C`/Java/Python), pre- and post-conditions ($\mathcal{P}$ and Q) expressed as code, constraints, or natural language. AutoBug automatically decomposes the program into a sequence of truncated slices, and then constructs a sequence of prompts (a.k.a. verification conditions) to be sent to the LLM for inference. The slices are ordered by size to find the *least* counterexample to the post-condition where applicable. AutoBug is also language-agnostic, except for the parser and some elements of the renderer. Furthermore, AutoBug is lightweight and approximate by design—significantly simplifying the implementation (i.e., it does not rely on language-specific compiler infrastructure). This also reflects the nature of LLM inference, which is heuristic by nature rather than relying on precise parsing and semantic analysis, but can still make useful inferences for many real-world applications. The implementation is also designed to be LLM-agnostic: since most modern LLMs expose a common API interface, AutoBug can be configured to query different models without changes to the core workflow.

5 Evaluation

To evaluate the effectiveness of LLM-based symbolic execution implemented by AutoBug, we answer the following research questions:

RQ.1 (Accuracy) What is the accuracy of AutoBug compared to baselines?
RQ.2 (Scalability) Can AutoBug scale to real-world large-scale programs?
RQ.3 (Language Agnosticism) Can AutoBug support multiple programming languages?

5.1 Experimental Setup

5.1.1 Dataset. We evaluated AutoBug on two widely-used datasets: REval and CodeForces. REval [13] is designed to evaluate the code-reasoning abilities of LLMs. It consists of 85 Python solutions to LeetCode problems, some of which include accompanying test suites. The subjects in REval usually employ multiple control flow constructs, such as nested conditions and loops. This dataset was released in June 2024. In addition, we included the CodeForces dataset, which presents more challenging programming tasks. CodeForces is a programming contest platform that regularly publishes new problem sets. From this platform, we collected 662 subjects written in both Python and `C`, published in June 2025. To the best of our knowledge, this is the most recent and largest open dataset currently available from the platform. The subjects in both datasets typically span around 100 lines of code, with token counts reaching up to 900.

To conduct the evaluation, we also needed to construct non-trivial Hoare triples for each subject. This presented a challenge, as both REval and CodeForces provide only problem descriptions in natural language, source-code solutions, and test suites in some cases. To generate Hoare triples, we employed three different strategies:

- REval-Desc: We treated the program descriptions in natural language as the post-conditions, and used any restrictions on test input when available as the pre-conditions. This strategy was applied to the REval dataset to construct REval-Desc with the corresponding pre- and post-conditions.
- CodeForces-Automatic: In this strategy, we used an LLM to automatically generate pre- and post-conditions in natural language based on the program descriptions from CodeForces. This resulted in the dataset CodeForces-Automatic for evaluation.
- Mixed-Manual: Here, we manually annotated formal pre- and post-conditions in executable code based on implicit properties of the original problems, e.g., by adding assertions to verify the correctness of a sorting algorithm. Given the substantial manual effort required, we randomly sampled 30 subjects in both REval and CodeForces, covering both Python and `C` solutions, and applied this strategy to them. These formed another dataset Mixed-Manual.

5.1.2 Baselines. For comparison, we selected two main types of baselines. For the first baseline, we used an "ad-hoc" LLM-based program analysis over the entire subject without decomposition. This baseline queries the LLM for a counterexample (if one exists) or whether the post-condition always holds for inputs that satisfy the pre-condition. This baseline aims to show the effectiveness of the systematic partitioning provided by our tool AutoBug.

The second baseline includes traditional symbolic execution tools. For Python subjects, we selected CrossHair [61] as a baseline. Although there are other traditional symbolic execution tools for Python, such as PyExZ3 [7] and pySym [8], they are no longer maintained (unmaintained for six years or more at the time of writing). For the `C` subjects, we compared our tool with KLEE [11], a well-established and mature traditional symbolic execution tool. A practical issue of these traditional tools is the lack of expressivity in the pre- and post-conditions, which must be specified in code or other formal languages. Therefore, we only compare AutoBug with traditional tools on the Mixed-Manual dataset, which contains conditions in the formal form of executable code.

5.1.3 Test Models. We selected five of the most popular openly available language models (at the time of writing)[5]. Where possible, we also included different numbers of parameters to demonstrate the potential impact of the parameter number.

- Meta's LLAMA3 series [33]: Instruction-tuned model optimized for general use cases such as dialogue and chatting, which was released in December 2024.
- Microsoft's PHI-4 [1]: Model trained on high-quality data from filtered public domain websites and acquired academic books, designed to aid research on language models, which was released in December 2024.
- DeepSeek's DEEPSEEK-R1 [22]: The first-generation reasoning model from DeepSeek, achieving performance comparable to OpenAI-o1 across math, code, and reasoning tasks with much smaller parameters, which was released in January 2025.
- Google's GEMMA3 [66]: A lightweight family of multimodal models that features a larger context window of 128K, and released in March 2025.
- Alibaba's QWEN3 [75]: The latest version of the QWEN model family, designed to advance performance, efficiency, and multilingual capabilities, and it was released on April 2025.

All experiments were carried out using `ollama` version v0.9.6. Each experiment was repeated three times, and we report the average results rounded to the nearest integer.

5.2 RQ.1: Accuracy

Method. To evaluate the accuracy of AUTOBUG in generating counterexamples and verifying post-conditions, we compared AUTOBUG with the "ad-hoc" LLM-based program analysis on REVAL-DESC, CODEFORCES-AUTOMATIC, and MIXED-MANUAL. In addition, we compared AUTOBUG with traditional symbolic execution tools CrossHair and KLEE on MIXED-MANUAL with formal pre- and post-conditions expressed in executable Python or `C` code. We did not conduct the comparison on other datasets, since traditional tools cannot handle post-conditions in natural language. In each subject, we executed AUTOBUG and baselines, and reported their results. The accuracy in each subject was measured by comparing the result with the ground truth derived from the provided test suites, if available, or manual verification.

Main result. Table 2 shows the accuracy of AUTOBUG compared to the "ad-hoc" LLM-based program analysis across different datasets under different models. The experiments were conducted under the list of models (*Model*). For each dataset, we report the total number of subjects analyzed (*Total*), how many subjects were correctly analyzed by the tool (*Correct*), and the corresponding accuracy rate (*Accuracy*). Overall, AUTOBUG consistently achieves a higher accuracy than the baseline in all datasets using different forms of Hoare triples. On average, AUTOBUG improves accuracy from 84.7% to 90.6% on REVAL-DESC, from 79.2% to 86.4% on CODEFORCES-AUTOMATIC, and from 66.7% to 73.3% on MIXED-MANUAL. Notably, on these three (3) datasets under eight (8) language models, AUTOBUG outperforms the baseline in 20 out of 24 experiments (i.e., 83.3% scenarios), with the only exceptions on REVAL-DESC and MIXED-MANUAL when using the model GEMMA3-27B. These results demonstrate the effectiveness of our approach in improving the accuracy of program analysis.

[5]https://ollama.com/search

Table 2. Accuracy of AutoBug and the "ad-hoc" LLM-based program analysis under different datasets.

Model	Method	REval-Desc			CodeForces-Automatic			Mixed-Manual		
		Total	Correct	Accuracy	Total	Correct	Accuracy	Total	Correct	Accuracy
Llama3-8B	AutoBug	85	82	**96.5%**	662	654	**98.8%**	30	23	**76.7%**
	Baseline	85	79	92.9%	662	653	98.6%	30	22	73.3%
Llama3.1-8B	AutoBug	85	84	**98.8%**	662	616	**93.1%**	30	23	**76.7%**
	Baseline	85	71	83.5%	662	547	82.6%	30	18	60.0%
Llama3.3-70B	AutoBug	85	72	**84.7%**	662	617	**93.2%**	30	26	**86.7%**
	Baseline	85	70	82.4%	662	589	89.0%	30	21	70.0%
Phi-4-14B	AutoBug	85	74	**87.1%**	662	626	**94.6%**	30	24	80.0%
	Baseline	85	72	84.7%	662	584	88.2%	30	24	80.0%
DeepSeek-R1-70B	AutoBug	85	85	**100.0%**	662	561	**84.7%**	30	19	**63.3%**
	Baseline	85	82	96.5%	662	508	76.7%	30	17	56.7%
Gemma3-4B	AutoBug	85	73	**85.9%**	662	305	**46.1%**	30	20	**66.7%**
	Baseline	85	64	75.3%	662	265	40.0%	30	16	53.3%
Gemma3-27B	AutoBug	85	72	84.7%	662	598	**90.3%**	30	20	66.7%
	Baseline	85	76	**89.4%**	662	547	82.6%	30	23	**76.7%**
Qwen3-32B	AutoBug	85	71	**83.5%**	662	601	**90.8%**	30	21	70.0%
	Baseline	85	61	71.8%	662	499	75.4%	30	21	70.0%
Average	AutoBug	85	77	**90.6%**	662	572	**86.4%**	30	22	**73.3%**
	Baseline	85	72	84.7%	662	524	79.2%	30	20	66.7%

The comparison results with traditional symbolic execution tools are presented in Table 3, which shows the accuracy of AutoBug, and CrossHair and KLEE on Mixed-Manual with formal post-conditions in the form of executable code. We can see that AutoBug significantly outperforms these traditional tools. AutoBug successfully analyzes 73.3% of subjects, compared to just 46.7% for traditional tools. This performance gap is due to the inherent limitations of traditional symbolic execution (as illustrated in Section 2.2), such as the difficulties in reasoning about nested loops and complex constructs. Moreover, compared to traditional tools, AutoBug also demonstrates greater applicability by supporting expressive post-conditions, including those written in natural language.

Table 3. Accuracy of AutoBug and traditional symbolic execution tools.

Method	Mixed-Manual		
	Total	Correct	Accuracy
AutoBug	30	22	**73.3%**
Baseline	30	14	46.7%

AutoBug improves accuracy over the baselines across different datasets. Moreover, AutoBug is more applicable than traditional symbolic execution tools, which can handle pre- and post-conditions expressed in either code or natural language.

Impact of different LLMs. We evaluated AutoBug on multiple different LLMs, even with different numbers of parameters as shown in Table 2: Meta's Llama3, Llama3.1 and Llama3.3 models with 8B and 70B parameters, Microsoft's 14B Phi-4 model, the 70B distilled version of DeepSeek-R1 model, Google's Gemma3 model with 4B and 27B parameters, and Alibaba's 32B Qwen3.

It is interesting to note that the accuracy varies significantly under different models, even with different model sizes. Among the models used, the most significant accuracy improvement stems from Llama3.1-8B, where AutoBug achieves 98.8% accuracy compared to the baseline's 83.5%. This highlights the benefits of partitioning in improving the accuracy of LLM reasoning for comparatively small models. In contrast, on larger models such as Llama3.3-70B and DeepSeek-R1-70B, such improvements in accuracy are more subtle, mainly due to the fact that these larger models inherently have a stronger reasoning ability. However, the result is consistently positive for larger models, with the exception of Gemma3-27B. This is a positive result—smaller LLMs can be run locally on consumer hardware, and AutoBug exhibits a clear benefit for these use cases.

Under Gemma3-27B, AutoBug outperforms the baseline on CodeForces-Automatic, but shows weaknesses on REval-Desc and Mixed-Manual. Upon closer investigation, we attribute this to Gemma3-27B's tendency to overinterpret problem specifications and penalize even minor mismatches. For example, during slicing, AutoBug may omit certain input-handling statements if they are irrelevant to the target context, resulting in incomplete input-processing logic. Gemma3-32B penalizes such incompleteness and reports unsatisfactory when inputs are not fully processed, while Gemma3-4B tends to overlook these issues and focuses on reasoning about the main task.

AutoBug improves accuracy over the evaluated LLMs, where smaller LLMs benefit the most.

Prompt size. Table 4 shows the average token counts used in the LLM prompts for both the baseline and AutoBug across different datasets. To calculate the token count in each LLM prompt, we use the tool `tiktoken` [37] with the corresponding Byte-Pair Encoding (BPE) token encoding, which is also used by GPT-4o mini [55]. To clearly demonstrate the reduction ability of AutoBug, we also reported the reduction ratio achieved by AutoBug, compared with the baseline.

As shown in Table 4, AutoBug effectively reduces token counts—reducing the length and complexity of the prompt—with average reductions of approximately 9.5%, 25.3%, and 26.3% compared to the baseline for REval-Desc, CodeForces-Automatic, and Mixed-Manual, respectively.

Table 4. Average LLM token counts for the baseline and AutoBug under different datasets, along with the token reduction ratio achieved by AutoBug. Each value is reported in the form of Mean ± Standard Deviation.

Method	REval-Desc	CodeForces-Automatic	Mixed-Manual
Baseline	224.8 ± 85.9	346.4 ± 116.4	146.9 ± 48.3
AutoBug	203.4 ± 75.5	258.6 ± 102.1	108.3 ± 40.2
Reduction Ratio	9.5%	25.3%	26.3%

AutoBug significantly reduces the average token count required for LLM queries to find the minimum counterexample across all tested datasets.

Case study. To illustrate the advantages of AutoBug over the baselines, we use a concrete example in Figure 5. This example consists of a buggy Python function shown in Figure 5 (a), which is extracted from the REval dataset [13]. Here, the pre- and post-conditions are formally annotated with the corresponding `PRE` and `POST` comments. The pre-condition specifies that `value` is not empty, which is indicated by the (`assume len(value) > 0`) statement located at the beginning of the function. The post-condition of the function states that $|\texttt{res}| \leq |\mathit{float}(\texttt{value})|$. This condition does not always hold, since the given function rounds up any decimal input ending in "`.5`". For example, $\mathit{closest\_integer}(\texttt{'1.5'}) = 2$, which violates the post-condition.

We find that most LLMs cannot correctly reason over the whole program. A typical response from Llama3.1-8B [33] when fed with the entire original program is as follows:

(a) original Python function

```
def closest_integer(value):
    assume len(value) > 0  # PRE

    if value.count('.') == 1:
        # remove trailing zeros
        while value[-1] == '0':
            value = value[:-1]

    num = float(value)
    if value[-2:] == '.5':
        if num > 0:
            res = ceil(num)
        else:
            res = floor(num)
    elif len(value) > 1 or value[0] != '0':
        res = int(round(num))
    else:
        res = 0

    assert abs(res) <= abs(float(value))  # POST
```

(b) truncated slice #1

```
def closest_integer(value):
    assume len(value) > 0  # PRE
    if value.count('.') == 1:
        # remove trailing zeros
        while value[-1] == '0':
            value = value[:-1]
    num = float(value)
    assume value[-2:] != '.5'
    assume len(value) > 1 or value[0] != '0'
    res = int(round(num))
    assert abs(res) <= abs(float(value))  # POST
```

(c) truncated slice #2

```
def closest_integer(value):
    assume len(value) > 0  # PRE
    assume value.count('.') != 1
    num = float(value)
    assume value[-2:] == '.5'
    assume num > 0
    res = ceil(num)
    assert abs(res) <= abs(float(value))  # POST
```

Fig. 5. (a) is a buggy Python program that implements a simple function to take a value (string) representing a number and return the closest integer to it. In cases where the number is equidistant from two integers, it rounds away from zero. This example includes an unbounded loop (`while value[-1] == '0': ...`), invokes third-party library APIs (`ceil`, `floor`, and `abs`), and introduces complex language constructs like list slicing. (b) and (c) show the corresponding program slides for inputs `"0.0"` and `"2.5"`, respectively.

> "*For inputs with a single decimal point, removing trailing zeros ensures that the absolute value of* `res` *(the rounded integer) is less than or equal to the absolute value of the original float (float*`(value)`*).*"

The LLM then proceeds to incorrectly declare that the post-condition of the program will always be satisfied. In this case, the LLM is distracted over the trailing zero removal branch so much, thereby it does not actually try to analyze what the program is supposed to do with the converted value—i.e., an example of hallucination.

In contrast, AutoBug decomposes the input program into truncated slices. Two of the generated slices from the original Python function are shown in Figure 5 (b) and (c). As illustrated in this example, we can see that AutoBug significantly reduces the complexity and size of the resulting truncated slices. In Figure 5 (b), only the `elif` branch is considered, with the others truncated, forming implicit pre-conditions in the form of `assume`-statements. In Figure 5 (c), only the second `if` branch is assumed to be taken. In addition, the original Python code (a) contains 430 tokens, which is reduced to 103 tokens (~76% reduction) for (b) and 69 tokens (~84% reduction) for (c). Together with the reduced CFG complexity, the slices form a smaller and more targeted prompt, increasing the chance of a correct validation result. Using these truncated slices, the LLM correctly infers that the post-condition holds for slice (b), but does not hold for slice (c). For example, Llama3.1-8B outputs the following for (c) (emphasis original):

> "In other words, given an input that satisfies the precondition (but has a decimal point followed by `'.5'`), the postcondition will not be satisfied. The result of `ceil` will exceed the absolute value of the original float. Therefore, we conclude that the postcondition indicated by the `assert` statement with a `POST` comment is **not always satisfiable**."

Since there exists a counterexample for (c), AutoBug has shown that the post-condition does not hold for the original program (a).

Table 5. Statistics of large-scale programs used, including lines of code (#LoC) and total token counts (#Token).

Subject	X11	ncurses	nano	cURL	SQLite3	OpenSSH	FFmpeg	Zstd
#LoC	244K	54K	24K	172K	225K	144K	255K	35K
#Token	4960K	438K	197K	1379K	2341K	1427K	2761K	370K

Table 6. The token counts for file-based (File), function-based (Func), and slice-based (Slice) decompositions of real-world bugs from X11 client applications and other examples. Here (✓) means the LLM correct detects the bug, (✗) means the bug was not detected, and (–) means the token limit for the model was exceeded.

Subject	Bug	Token Count			LLAMA3.1-8B			GEMMA3-4B		
		File	Func	Slice	File	Func	Slice	File	Func	Slice
`xinput`	`NULL`-*pointer*	4281	1197	178	✗	✗	✓	✗	✗	✓
`xlsclients`	`NULL`-*pointer*	140120	1090	229	–	✗	✗	–	✗	✗
`xmodmap`	`NULL`-*pointer*	18683	2189	279	✗	✗	✓	✗	✗	✓
`xset`	*Divide-by-zero*	17078	3164	506	✗	✓	✓	✗	✓	✓
`xwininfo`	*Buffer-overflow*	152020	2560	597	–	✗	✗	–	✗	✗
`ncurses`	`NULL`-*pointer*	1446	271	201	✓	✓	✓	✗	✓	✓
`nano`	*Environment race*	6716	698	125	✗	✓	✓	✗	✗	✓
`cURL`	`assert`-*trigger*	21285	5581	202	✗	✓	✓	✗	✗	✓
`SQLite3`	`NULL`-*pointer*	2341085	213088	4691	–	–	✗	–	–	✗
`OpenSSH`	`assert`-*trigger*	10164	446	208	✗	✓	✓	✗	✓	✓
`FFmpeg`	*Buffer-overflow*	118456	19242	4370	✗	✗	✓	✗	✗	✓
`Zstd`	`NULL`-*pointer*	60813	936	328	✗	✗	✓	✗	✗	✓
Accuracy	—	—	—	—	8.3%	41.7%	**75.0%**	0%	25.0%	**75.0%**

We also attempted to analyze Figure 5 (a) using CrossHair, a traditional symbolic execution engine for Python. However, CrossHair terminates after failing to find any violation of the post-condition. We believe that this is due to an incomplete search or incomplete solving of the path constraint by the underlying SMT solver (z3). In contrast, AUTOBUG is capable of exhaustively enumerating all path partitions, and the underlying LLM correctly infers the corresponding slice. This case study, although it involves a relatively simple program, highlights the limitations of traditional methods when compared to AUTOBUG.

5.3 RQ.2: Scalability

Method. To evaluate the scalability of AUTOBUG, we selected eight (8) real-world large-scale programs: X11, ncurses, nano, cURL, SQLite3, OpenSSH, FFmpeg, and Zstd, which are widely-used benchmarks for evaluating the performance of bug-finding techniques. Table 5 summarizes the statistics of these subject programs, including the lines of code (*#LoC*) and the token counts (*#Token*). These subjects have lines of code up to 250K and have 197K–4960K token counts. In this experiment, we examined whether AUTOBUG could successfully analyze the bugs recently found by existing bug detectors [49]. These bugs were found in several X11 client applications (xinput, xlsclients, xmodmap, xset, xwininfo), as well as in ncurses, SQLite3, and other common UNIX libraries, respectively. To construct post-conditions, we negated the crash condition, such as (*ptr* `!=` `NULL`) for `NULL`-pointer dereference. To demonstrate the effectiveness of AUTOBUG's decompositions, we evaluated three forms: only relevant source files included (*File*), only relevant functions included (*Func*), and relevant code sliced by AUTOBUG (*Slice*) along a path that exercises the bug.

Results. The results are shown in Table 6. AutoBug demonstrates strong scalability to real-world, large-scale programs, with token counts for all cases remaining within the context window of LLMs, making LLM-based symbolic execution feasible. Moreover, AutoBug achieves high accuracy, successfully analyzing 75.0% of the bugs. In contrast, traditional symbolic execution tools suffer from the well-known path-exploration problem and fail to scale to these programs. Additionally, the token counts for these subjects are too large for the ad-hoc LLM-based analysis within a single prompt, making decomposition necessary.

Compared to naïve decomposition methods based on files and functions, AutoBug's slice-based decomposition method is significantly more effective. Using slicing, AutoBug successfully analyzes 75.0% of the bugs, while the file-based and function-based methods achieve only 8.3% and 41.7% success rates, respectively. To further show the effectiveness of AutoBug's slicing method, we examine the bug in xinput. The bug resides in a program that contains 244K sLOC (~4.96M tokens), but the sliced code is reduced to just 178 tokens, as shown in Figure 6. With the code size and complexity dramatically reduced, the bug becomes apparent: the `XIQueryDevice()` function returns `NULL` (derived from an error-handling path), and this value is immediately passed to `XIFreeDeviceInfo()` (the intermediate code was removed by the slicer), thus violating the post-condition. After truncation and slicing, the violation is readily apparent even to small LLMs. In contrast, the other decompositions based on files and functions result in 4,281 and 1,197 tokens, respectively. They include large amounts of irrelevant information that obscure the bug and cause LLMs to miss it.

```
XIDeviceInfo* XIQueryDevice(Display *dpy,
              int deviceid, int *ndevices_return)
{
  xXIQueryDeviceReply reply;
  XExtDisplayInfo *extinfo = XInput_find_display(dpy)
      ;
  ...
  *ndevices_return = -1;
  return NULL;
}
void XIFreeDeviceInfo(XIDeviceInfo* info)
{
  // POST: info != NULL
}
static int list_xi2(Display *display,
         enum print_format format)
{
  // PRE: true
  int ndev;
  XIDeviceInfo *info, *dev;
  info = XIQueryDevice(display, XIAllDevices, &ndev);
  XIFreeDeviceInfo(info);
}
```

Fig. 6. Relevant code sliced for the bug in xinput.

AutoBug scales to real-world large-scale programs with 75% accuracy. Truncated slicing significantly reduces prompt size and complexity—improving the accuracy of LLM inference.

5.4 RQ.3: Language Agnosticism

Method. AutoBug implements a lightweight workflow that is language-agnostic. To evaluate AutoBug's ability to analyze programs written in different programming languages, we conducted this experiment using the same set of subjects implemented in Python, C, and Java, respectively. Specifically, we use GPT-4o-mini to automatically translate the REval-Desc dataset (originally written in Python) into equivalent C and Java versions. These translated programs form the C-REval-Desc and Java-REval-Desc datasets, each consisting of 85 programs. We evaluated the three datasets using the same LLMs and framework as in RQ.1, and then reported the average results.

Results. The results are illustrated in Table 7. Across these three datasets implemented in different programming languages, AutoBug maintains similar performance and consistently improves the accuracy of the analysis. Specifically, compared to the baseline—the ad-hoc LLM-based analysis, AutoBug increases the accuracy of the analysis from 84.7% to 90.6% in Python, 78.8% to 87.1% in C, and 83.5% to 89.4% in Java, respectively. These results demonstrate the versatility of AutoBug in

Table 7. Accuracy of AutoBug and the baseline on the Python, C, and Java versions of Reval-Desc.

Model	Method	Python-REval-Desc			C-REval-Desc			Java-REval-Desc		
		Total	Correct	Accuracy	Total	Correct	Accuracy	Total	Correct	Accuracy
Average	AutoBug	85	77	**90.6%**	85	74	**87.1%**	85	76	**89.4%**
	Baseline	85	72	84.7%	85	67	78.8%	85	71	83.5%

handling multi-language programs, which is the direct result of its language-agnostic workflow and the inherent ability of LLM to understand multiple languages. Notably, these improvements are achieved without reliance on precise parsers and/or language-specific compiler front-ends.

To evaluate traditional symbolic execution tools such as CrossHair and KLEE, annotating formal post-conditions requires significant manual effort. In RQ.1 (Section 5.2), we manually annotated formal post-conditions and have shown the effectiveness of AutoBug in the programs with formal post-conditions compared to traditional symbolic execution. In this experiment, we explored the potential of using LLMs to automatically generate post-conditions based on specifications and translate them into code suitable for symbolic execution. Specifically, for CrossHair, we instructed LLM to generate the PEP 315-suitable contract specification, while we prompted LLM to generate suitable calls to `klee_make_symbolic` for KLEE. However, the results were far from satisfactory. The LLMs consistently failed to translate natural language post-condition specifications into concrete, executable code. The generated code frequently contains compiler or runtime errors, including incomplete implementations, missing function bodies, or absent header and import statements. For example, CrossHair was only able to analyze fewer than 10% of the translated dataset (8 out of 85 subjects). These findings indicate that substantial manual effort is still necessary when using traditional symbolic execution tools.

Across the same set of subjects implemented in different programming languages (C, Python, and Java), AutoBug maintains similar performance and achieves consistent improvements in the accuracy of the analysis.

5.5 Discussion

Our results show that path-based decomposition of program analysis tasks is effective at improving LLM-based program analysis, especially for small LLMs and real-world problems. The significance is that it allows for higher accuracy to be achieved with smaller models that can be run on consumer-grade hardware. In addition, we prove that a lightweight and language agnostic workflow is feasible and can still achieve good results.

Limitations. LLM-based program analysis is applicable to program analysis tasks that cannot be handled by traditional means. That said, the approximate reasoning of LLMs is not suitable for all applications, even with improved accuracy. LLMs are also unlikely to ever achieve the same accuracy as traditional solvers for some tasks, such as solving systems of linear equations. As such, we propose LLM-based symbolic execution as a complementary method that does not necessarily replace traditional approaches for all use cases. Another limitation is that path-based decomposition may still explode, even when our approach is guaranteed to terminate. This is an inherent limitation of path-based reasoning. However, we believe that the decomposition based on the truncated slice is a significant mitigation.

Threat to validity. A primary concern is the potential for data contamination in LLMs [60], where our evaluation datasets may have been included in the models' training data. To mitigate this issue, we selected the subjects from CodeForces that were released in June 2025, after the models' pre-training cutoff date. As shown in Table 2, AutoBug consistently outperforms the baseline on these subjects. This suggests that the performance of AutoBug is not a result of data contamination.

6 Related Work

Static program analysis via symbolic execution. Symbolic execution [40] is an established method for static programs whose origins also relate to early work formalizing programs as mathematical logic. The idea is to execute symbolic states, representing sets of concrete inputs, allowing for the exhaustive exploration of program behavior. Over the decades, many different symbolic execution engines and frameworks have been developed, such as: KLEE [11], Owi [3], *Symbolic PathFinder* (SPF) [57], *Java PathFinder* (JPF) [69], CrossHair [61], Angr [64], S2E [16], PyExZ3 [7], etc. Unlike our approach, such traditional tools translate paths into some underlying formal language for theorem proving, thus inheriting many of the limitations discussed in this paper. Furthermore, most existing tools are specialized to a specific language (C, Java, binary, etc.) and are closely integrated into specific compiler frameworks (e.g., LLVM [42] for KLEE). That said, LLMs use a fundamentally different type of reasoning compared to the deductive reasoning of theorem provers. As such, traditional approaches are suitable for problems that can be handled by traditional methods and for applications where perfect accuracy is required.

LLM-based program analysis. One recent alternative to traditional program analysis methods is *Large Language Models* (LLMs). LLMs are very general tools and can be applied to a wide variety of tasks, including *fuzzing* [5, 50], *vulnerability detection* [79], and *program repair* [26]. Another recent innovation is LLM-based *agents* [73], which are algorithms where decisions are made by the LLM. Our core Algorithm 1 is traditional and not agent-based. However, an agentized version could be made, but decisions (e.g., which branch to explore first) could be deferred to the LLM.

Intersection between symbolic execution and LLMs. There are some other nascent works that combine LLMs and symbolic execution. HyLLfuzz [48] focuses on improving concolic execution integrated in hybrid fuzzing. LLM-Sym [70] is an agent-based symbolic execution framework for Python code that uses an LLM to translate paths into traditional *path constraints* suitable for solving via z3 [21]. LLM-Sym is fundamentally different in that our approach avoids translation altogether, instead directly using the LLM itself as a solver. Since LLM-Sym still uses translation to z3, it inherits many of the limitations of traditional symbolic execution engines discussed in this paper. Similarly, Loopy [38] aims to discover *loop invariants* using LLMs, which can then be applied to symbolic analysis. Our approach avoids the need for invariant discovery, since it is not based on translation.

7 Conclusion

In this paper, we introduce a variant of symbolic execution that uses an LLM as the underlying reasoning engine instead of a traditional theorem prover or SMT solver. Our approach introduces a generic path constraint representation in terms of the original code—allowing the LLM to reason directly over the path constraint and avoiding translation into a (less expressive) formal language. Our approach allows for a path-based decomposition of the analysis task into smaller (more tractable) subtasks, which use fewer tokens (helping *scale*), and are more targeted (helping *accuracy*). We implemented our approach in the form of AutoBug—a practical LLM-based symbolic execution engine that supports multiple programming languages (i.e., language agnostic, supporting C/Python/Java) without depending on heavyweight compiler infrastructure. Our experimental results demonstrate measurable improvements in terms of both accuracy and scale, especially in smaller models that can run on consumer-grade GPUs.

References

[1] M. Abdin and et al. 2024. Phi-4 Technical Report. arXiv:2412.08905 [cs.CL] https://arxiv.org/abs/2412.08905

[2] H. Agrawal, R. Demillo, and E. Spafford. 1993. Debugging with dynamic slicing and backtracking. *Softw. Pract. Exper.* 23, 6 (June 1993), 589–616. doi:10.1002/spe.4380230603

[3] L. Andrès, F. Marque, A. Carcano, P. Chambart, J. Santos, and J. Filliâtre. 2024. Owi: Performant Parallel Symbolic Execution Made Easy, an Application to WebAssembly. *The Art, Science, and Engineering of Programming* 9, 1 (Oct. 2024). doi:10.22152/programming-journal.org/2025/9/3

[4] K. Arnold, J. Gosling, and D. Holmes. 2005. *The Java programming language.* Addison Wesley Professional.

[5] Asmita, Y. Oliinyk, M. Scott, R. Tsang, C. Fang, and H. Homayoun. 2024. Fuzzing BusyBox: Leveraging LLM and Crash Reuse for Embedded Bug Unearthing. In *33rd USENIX Security Symposium (USENIX Security 24).* USENIX Association, Philadelphia, PA, 883–900. https://www.usenix.org/conference/usenixsecurity24/presentation/asmita

[6] R. Baldoni, E. Coppa, D. D'elia, C. Demetrescu, and I. Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (May 2018), 39 pages. doi:10.1145/3182657

[7] T. Ball and J. Daniel. 2015. Deconstructing Dynamic Symbolic Execution. *Dependable Software Systems Engineering* 40 (January 2015).

[8] M. Bann. 2016. pySym: Python Symbolic Execution. https://github.com/bannsec/pySym.

[9] M. Brunsfeld and et al. 2025. *tree-sitter/tree-sitter: v0.25.1.* doi:10.5281/zenodo.14788680

[10] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. Lee, Y. Li, S. Lundberg, H. Nori, H. Palangi, M. Ribeiro, and Y. Zhang. 2023. Sparks of Artificial General Intelligence: Early experiments with GPT-4. arXiv:2303.12712 [cs.CL] https://arxiv.org/abs/2303.12712

[11] C. Cadar, D. Dunbar, and D. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08).* USENIX Association, San Diego, CA. https://www.usenix.org/conference/osdi-08/klee-unassisted-and-automatic-generation-high-coverage-tests-complex-systems

[12] C. Cadar and K. Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90. doi:10.1145/2408776.2408795

[13] J. Chen, Z. Pan, X. Hu, Z. Li, G. Li, and X. Xia. 2024. Reasoning Runtime Behavior of a Program with LLM: How Far Are We? arXiv:2403.16437 [cs.SE] https://arxiv.org/abs/2403.16437

[14] M. Chen and et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[15] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin. 2024. ChatUniTest: A Framework for LLM-Based Test Generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024).* Association for Computing Machinery, New York, NY, USA, 572–576. doi:10.1145/3663529.3663801

[16] V. Chipounov, V. Kuznetsov, and G. Candea. 2012. The S2E Platform: Design, Implementation, and Applications. *ACM Trans. Comput. Syst.* 30, 1, Article 2 (Feb. 2012), 49 pages. doi:10.1145/2110356.2110358

[17] E. Clarke and E. Emerson. 1982. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs*, Dexter Kozen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 52–71.

[18] L. Clarke. 1976. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering* SE-2, 3 (1976), 215–222. doi:10.1109/TSE.1976.233817

[19] P. Cousot and R. Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77).* Association for Computing Machinery, New York, NY, USA, 238–252. doi:10.1145/512950.512973

[20] P. Cousot and R. Cousot. 1979. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (San Antonio, Texas) *(POPL '79).* Association for Computing Machinery, New York, NY, USA, 269–282. doi:10.1145/567752.567778

[21] L. de Moura and N. Bjørner. 2008. Z3: an efficient SMT solver. In *2008 Tools and Algorithms for Construction and Analysis of Systems.* Springer, Berlin, Heidelberg, 337–340. https://www.microsoft.com/en-us/research/publication/z3-an-efficient-smt-solver/

[22] DeepSeek-AI. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948 [cs.CL] https://arxiv.org/abs/2501.12948

[23] E. Dijkstra. 1975. Guarded commands, nondeterminancy and formal derivation of programs. *Commun. ACM* 18, 8 (1975), 453–457. doi:10.1145/360933.360975

[24] D. Distefano, M. Fähndrich, F. Logozzo, and P. O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (July 2019), 62–70. doi:10.1145/3338112

[25] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. 2001. Bugs as deviant behavior: a general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev.* 35, 5 (Oct. 2001), 57–72. doi:10.1145/502059.502041

[26] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. Tan. 2023. Automated Repair of Programs from Large Language Models. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1469–1481. doi:10.1109/ICSE48619.2023.00128

[27] C. Fang, N. Miao, S. Srivastav, J. Liu, R. Zhang, R. Fang, Asmita, R. Tsang, N. Nazari, H. Wang, and H. Homayoun. 2024. Large language models for code analysis: do LLMs really do their job?. In *Proceedings of the 33rd USENIX Conference on Security Symposium* (Philadelphia, PA, USA) *(SEC '24)*. USENIX Association, USA, Article 47, 18 pages.

[28] R. Floyd. 1967. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics* 19 (1967), 19–32. http://laser.cs.umass.edu/courses/cs521-621.Spr06/papers/Floyd.pdf

[29] M. Gadelha, H. Ismail, and L. Cordeiro. 2017. Handling loops in bounded model checking of C programs via k-induction. *Int. J. Softw. Tools Technol. Transf.* 19, 1 (Feb. 2017), 97–114. doi:10.1007/s10009-015-0407-9

[30] P. Godefroid, N. Klarlund, and K. Sen. 2005. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 213–223. doi:10.1145/1065010.1065036

[31] J. Goodenough and S. Gerhart. 1975. Toward a theory of test data selection. In *Proceedings of the International Conference on Reliable Software* (Los Angeles, California). Association for Computing Machinery, New York, NY, USA, 493–510. doi:10.1145/800027.808473

[32] S. Graham, P. Kessler, and M. Mckusick. 1982. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction* (Boston, Massachusetts, USA) *(SIGPLAN '82)*. Association for Computing Machinery, New York, NY, USA, 120–126. doi:10.1145/800230.806987

[33] A. Grattafiori and et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[34] R. Hamlet. 1977. Testing Programs with the Aid of a Compiler. *IEEE Trans. Softw. Eng.* 3, 4 (July 1977), 279–290. doi:10.1109/TSE.1977.231145

[35] C. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. doi:10.1145/363235.363259

[36] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods*, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 41–55.

[37] S. Jain. 2025. tiktoken: A fast BPE tokeniser for use with OpenAI's models. https://github.com/openai/tiktoken.

[38] A. Kamath, A. Senthilnathan, S. Chakraborty, P. Deligiannis, S. Lahiri, A. Lal, A. Rastogi, S. Roy, and R. Sharma. 2023. Finding Inductive Loop Invariants using Large Language Models. arXiv:2311.07948 [cs.PL] https://arxiv.org/abs/2311.07948

[39] A. Kaur and R. Nayyar. 2020. A Comparative Study of Static Code Analysis tools for Vulnerability Detection in C/C++ and JAVA Source Code. *Procedia Computer Science* 171 (01 2020), 2023–2029. doi:10.1016/j.procs.2020.04.217

[40] J. King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (July 1976), 385–394. doi:10.1145/360248.360252

[41] B. Korel and J. Laski. 1988. Dynamic program slicing. *Inform. Process. Lett.* 29, 3 (1988), 155–163. doi:10.1016/0020-0190(88)90054-3

[42] C. Lattner and V. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. San Jose, CA, USA, 75–88.

[43] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. doi:10.1109/TSE.2011.104

[44] M. Levy, A. Jacoby, and Y. Goldberg. 2024. Same Task, More Tokens: the Impact of Input Length on the Reasoning Performance of Large Language Models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 15339–15353. doi:10.18653/v1/2024.acl-long.818

[45] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, T. Hubert, P. Choy, C. Autume, I. Babuschkin, X. Chen, P. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. Mankowitz, E. Robson, P. Kohli, N. Freitas, K. Kavukcuoglu, and O. Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (Dec. 2022), 1092–1097. doi:10.1126/science.abq1158

[46] F. Long and M. Rinard. 2016. Automatic patch generation by learning correct code. *SIGPLAN Not.* 51, 1 (Jan. 2016), 298–312. doi:10.1145/2914770.2837617

[47] Y. Majdoub and E. Charrada. 2024. Debugging with Open-Source Large Language Models: An Evaluation. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Barcelona, Spain) *(ESEM '24)*. Association for Computing Machinery, New York, NY, USA, 510–516. doi:10.1145/3674805.3690758

[48] R. Meng, G. Duck, and A. Roychoudhury. 2024. Large language model assisted hybrid fuzzing. *arXiv preprint arXiv:2412.15931* (2024).

[49] R. Meng, G. Duck, and A. Roychoudhury. 2024. Program Environment Fuzzing. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 720–734. doi:10.1145/3658644.3690229

[50] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury. 2024. Large Language Model guided Protocol Fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*.

[51] B. Miller, L. Fredriksen, and B. So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44. doi:10.1145/96267.96279

[52] M. Nelson. 2005. A Survey of Reverse Engineering and Program Comprehension. arXiv:cs/0503068 [cs.SE] https://arxiv.org/abs/cs/0503068

[53] N. Nethercote and J. Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.* 42, 6 (June 2007), 89–100. doi:10.1145/1273442.1250746

[54] OpenAI. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[55] OpenAI. 2024. GPT-4o mini: advancing cost-efficient intelligence. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/

[56] C. Pacheco and M. Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion* (Montreal, Quebec, Canada) *(OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 815–816. doi:10.1145/1297846.1297902

[57] C. Păsăreanu and N. Rungta. 2010. Symbolic PathFinder: symbolic execution of Java bytecode. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering* (Antwerp, Belgium) *(ASE '10)*. Association for Computing Machinery, New York, NY, USA, 179–180. doi:10.1145/1858996.1859035

[58] S. Rapps and E. Weyuker. 1985. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering* SE-11, 4 (1985), 367–375. doi:10.1109/TSE.1985.232226

[59] J. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 55–74. doi:10.1109/LICS.2002.1029817

[60] J. Sallou, T. Durieux, and A. Panichella. 2024. Breaking the Silence: the Threats of Using LLMs in Software Engineering. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results* (Lisbon, Portugal) *(ICSE-NIER'24)*. Association for Computing Machinery, New York, NY, USA, 102–106. doi:10.1145/3639476.3639764

[61] P. Schanely. 2017. CrossHair: Symbolic Execution for Python. https://github.com/pschanely/CrossHair.

[62] K. Sen, D. Marinov, and G. Agha. 2005. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Lisbon, Portugal) *(ESEC/FSE-13)*. Association for Computing Machinery, New York, NY, USA, 263–272. doi:10.1145/1081706.1081750

[63] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, USA, 28.

[64] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. 138–157. doi:10.1109/SP.2016.17

[65] X. Si, H. Dai, M. Raghothaman, M. Naik, and L. Song. 2018. Learning loop invariants for program verification. *Advances in Neural Information Processing Systems* 31 (2018).

[66] Gemma Team. 2024. Gemma: Open Models Based on Gemini Research and Technology. arXiv:2403.08295 [cs.CL] https://arxiv.org/abs/2403.08295

[67] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL] https://arxiv.org/abs/2302.13971

[68] G. van Rossum. 1995. *Python tutorial*. Technical Report CS-R9526. Centrum voor Wiskunde en Informatica (CWI), Amsterdam.

[69] W. Visser, C. Păsăreanu, and S. Khurshid. 2004. Test input generation with java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis* (Boston, Massachusetts, USA) *(ISSTA '04)*. Association for Computing Machinery, New York, NY, USA, 97–107. doi:10.1145/1007512.1007526

[70] W. Wang, K. Liu, A. Chen, G. Li, Z. Jin, G. Huang, and L. Ma. 2024. Python Symbolic Execution with LLM-powered Code Generation. arXiv:2409.09271 [cs.SE] https://arxiv.org/abs/2409.09271

[71] M. Weiser. 1981. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering* (San Diego, California, USA) *(ICSE '81)*. IEEE Press, 439–449.

[72] E. Weyuker. 1983. Assessing Test Data Adequacy through Program Inference. *ACM Trans. Program. Lang. Syst.* 5, 4 (Oct. 1983), 641–655. doi:10.1145/69575.357231

[73] Z. Xi, W. Chen, X. Guo, W. He, Y. Ding, B. Hong, M. Zhang, J. Wang, S. Jin, E. Zhou, R. Zheng, X. Fan, X. Wang, L. Xiong, Y. Zhou, W. Wang, C. Jiang, Y. Zou, X. Liu, Z. Yin, S. Dou, R. Weng, W. Cheng, Q. Zhang, W. Qin, Y. Zheng, X. Qiu, X. Huang, and T. Gui. 2025. The rise and potential of large language model based agents: a survey. *Science China Information Sciences* 68, 2 (17 Jan 2025), 121101. doi:10.1007/s11432-024-4222-0

[74] C. Steven Xia, Y. Wei, and L. Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE).* 1482–1494. doi:10.1109/ICSE48619.2023.00129

[75] A. Yang and et al. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] https://arxiv.org/abs/2505.09388

[76] A. Zeller. 2005. *Why Programs Fail: A Guide to Systematic Debugging.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[77] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. 2024. *The Fuzzing Book.* CISPA Helmholtz Center for Information Security. https://www.fuzzingbook.org/ Retrieved 2024-07-01 16:50:18+02:00.

[78] X. Zhou, S. Cao, X. Sun, and D. Lo. 2024. Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead. *ACM Trans. Softw. Eng. Methodol.* (Dec. 2024). doi:10.1145/3708522 Just Accepted.

[79] X. Zhou, T. Zhang, and D. Lo. 2024. Large Language Model for Vulnerability Detection: Emerging Results and Future Directions. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results* (Lisbon, Portugal) *(ICSE-NIER'24).* Association for Computing Machinery, New York, NY, USA, 47–51. doi:10.1145/3639476.3639762

[80] H. Zhu, P. Hall, and J. May. 1997. Software unit test coverage and adequacy. *ACM Comput. Surv.* 29, 4 (Dec. 1997), 366–427. doi:10.1145/267580.267590

Received 2025-03-26; accepted 2025-08-12