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 probabilistic by nature, and therefore face significant challenges in relation to the *accuracy* and *scale* of the analysis in real-world application. 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) sub-tasks. 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 AutoExe, 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 AutoExe can improve both the accuracy and scale of LLM-based program analysis, especially for smaller LLMs that can run on consumer grade hardware.

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* [39], *abstract interpretation* [19, 20], and *model checking* [17]—that analyze the program without executing it, as well as *dynamic* methods—such as *fuzzing* [51, 76], *concolic execution* [29, 61], *instrumentation* [53, 62], and *profiling* [31]—that analyze program behavior based on observations of actual executions. In general, program analysis has many applications, such as program testing [11, 18, 29, 30, 33, 56, 58, 61, 71, 79], debugging [2, 25, 75], verification [28, 34], repair [43, 46], reverse engineering [52], and vulnerability detection [38]. One recent alternative to traditional program analysis methods has emerged in the form of *Large Language Models* (LLMs) [10, 14, 45, 54, 66]. Here, LLMs can make inferences over code directly 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 [73, 77, 78] 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 uses

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

probabilistic 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, 48], 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 probabilistic 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 probabilistic 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 sub-tasks 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 both in 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 AutoExe—an automated LLM-based symbolic execution engine. AutoExe uses a path-based decomposition of a program analysis task into smaller (more-tractable) sub-tasks that are suitable for LLM inference. Based on the observation that LLMs are inherently approximate oracles, we propose a lightweight design of AutoExe that is language agnostic and does not rely on any heavyweight compiler infrastructure. AutoExe 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/etc.). 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 AutoExe—a lightweight LLM-based symbolic execution engine that supports multiple programming languages without relying on heavyweight compiler infrastructure.

- We evaluate AutoExe against various program analysis tasks for C/Python/Java code. We show that AutoExe 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, 39] 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*, a.k.a. *path constraints*, that are traditionally represented as a set of *variables* (e.g., x, y, etc.) subject to *constraints* (e.g., $\mathsf{x} < \mathsf{y}$, etc.). The symbolic state represents the set of *concrete* states (e.g., $\{(\mathsf{x},\mathsf{y}) \mid \mathsf{x} < \mathsf{y}\}$) that is reachable from the 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 [34]:

$$\{\mathcal{S}\}\ \ \mathsf{x{+}{+}}\ \ \{\exists z : \mathsf{x} = z + 1 \wedge \mathcal{S}[\mathsf{x} \mapsto z]\}$$

Given a symbolic state ($\mathcal{S}$), the triple describes the resulting symbolic state ($\mathcal{S}'$) after the execution of the increment operator. For example, given $\mathcal{S} = (\mathsf{x} < \mathsf{y})$, then $\mathcal{S}' = (\exists z : \mathsf{x} = z + 1 \wedge z < \mathsf{y})$, or equivalently, $\mathcal{S}' = (\mathsf{x} \leq \mathsf{y})$. Conditional statements (e.g., **if** c **then** T **else** E **end**) are typically handled by *forking* the symbolic state ($\mathcal{S}$) into separate $\mathcal{S}_1 = (\mathcal{S} \wedge c)$ and $\mathcal{S}_2 = (\mathcal{S} \wedge \neg c)$, and then continuing the execution along the individual *then*- and *else*-branches. As similar strategy is used for loops (e.g., **while** c **do** B **done**). Finally, for each path (π) through the program, a final path constraint $\mathcal{S}_\pi$ 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 ($\mathcal{S}_\pi \models Q$) holds or not. The VCs can be (dis)proved (i.e., *disposed* of) with the help of a suitable *theorem prover*, such as an SMT solver [21]. By executing sets of concrete inputs at once, symbolic execution will exhaustively explore the (infinite) space of program behaviors—something cannot be explored by concrete execution alone.

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

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

A theorem prover disproves (VC-Else), meaning (Q) does **not** hold for the whole program. □

2.1.2 Large Language Models. The emergence of *Large Language Models* (LLMs) [10, 14, 45, 54, 66] 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], repair [73, 77, 78], etc. 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

(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.

answer. This is because LLMs fundamentally rely on learned patterns and probabilistic 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, etc.), it also has well-known limitations regarding loops, environment, and heap manipulating programs. We elaborate on the ideas from [48], as summarized 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 lead to an infinite unrolling of the loop, where each possible loop iteration $\{0, 1, 2, ...\}$ is treated as a separate path.

Popular symbolic execution tools, such as KLEE [11], handle this problem using a *concrete iteration bound*—exchanging potential infinite exploration with an incomplete exploration, as necessary.

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 solving* (CBMC [41]), or machine learning (Code2Inv [64]). 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 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.

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, DAGs, etc. 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 [35] and Infer [24]. However, VeriFast is manual and annotation-heavy, and Infer is limited to heuristic-based inference based on common structural invariants, such as linked-lists.

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 of 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 are *approximate oracles* in that they use probabilistic reasoning, and may not necessarily generate accurate answers. 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]. This concept could be described as a *soft* token limit, i.e., that prompts should be concise and targeted in order to maximize LLM accuracy.

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 of the limitations of traditional symbolic execution are limitations 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 formula—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, external environment interactions, etc.

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)
$\mathsf{x} > \mathsf{y} \land \mathsf{z} = \mathsf{x} \times \mathsf{y}$	$sp(S, \mathit{true})$ where $S = \{\textbf{assume}(\mathsf{x} \leq \mathsf{y}); \mathsf{z} := \mathsf{x} * \mathsf{y}\}$

The key idea is that both (φ) and $sp(S, \mathit{true})$ 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:

"*Given* $\{\textbf{assume}(\mathsf{x} \leq \mathsf{y}); \mathsf{z} := \mathsf{x} * \mathsf{y}\}$*, does the post-condition* $\mathsf{z} > \mathsf{y}$ *always hold?*" (VC-Else-2)

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

> "*The post-condition* $\mathsf{z} > \mathsf{y}$ **does not always hold**. *A simple counterexample is when* $\mathsf{x} = 1$*, where* $\mathsf{z} = \mathsf{y}$ *instead of* $\mathsf{z} > \mathsf{y}$. *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 ($\mathcal{P}$ 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:

$$sp(\textbf{(b)}, \mathcal{P}) \models Q \qquad \Rightarrow \qquad sp(\textbf{(a)}, \mathcal{P}) \models Q \qquad\qquad (\textsc{VeriCond})$$

Thus, to prove Q for holds (a), it is sufficient to prove that Q holds for (b). Furthermore, program (b) is targeted to the specific program analysis task (that `mutx` is unlocked), and is ~80% smaller than in terms of token count (419 vs 81).[2] The LLM can determine that VeriCond **holds** using the program (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). In this case, the LLM determines the post-condition does **not hold** using (c), since `strdup()` may return `NULL`. □

[1]This is not necessarily true, but is an assumption for the sake of an example that can fit within the page limit.
[2]As counted by https://platform.openai.com/tokenizer at the time of writing

	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	Probabilistic, 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], Java+SPF [57])	Programming language agnostic
	Compiler infrastructure	Close integration (LLVM [42]+KLEE, etc.)	Lightweight (AST-level) implementation

Fig. 2. Summary of the main similarities and differences between traditional and LLM-based symbolic execution. Here (*Cap* = *Capabilities*), (*Imp* = *Implementation*), (*sp* = *strongest post-condition*), and (the key differences).

Algorithm. An overview of our LLM-based symbolic execution algorithm is summarized in Algorithm 1. Here, the algorithm's *frontend* 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 be used to prove or disprove the post-condition modulo the LLM's reasoning capabilities. 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 {P}C{Q}
Output: HOLDS or a counter-example S
1  Function LLMSymExe({P}C{Q}):
2      AST ← Parse(C)
3      CFG ← GenCFG(AST)
4      partitions ← GenPartitions(CFG)
5      for Π ∈ partitions do
6          S ← GenSubProg(P, Π, Q)
7          prompt ← "assuming 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 sub-tasks. A summary of the main similarities and differences are shown in Figure 2. 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 *probabilistic* 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 programming language used by Hoare logic [34]. 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 AutoExe—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 [34] augmented with an explicit **assume**-statement.[3] 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:

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

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:

$$\begin{gathered} unfold(\textbf{skip}, \pi) = \{\pi; \textbf{skip}\} \qquad unfold(\textbf{assume}(b), \pi) = \{\pi; \textbf{assume}(b)\} \qquad unfold(x := e, \pi) = \{\pi; x := e\} \\ unfold(\{C_1;C_2\}, \pi) = \cup\{unfold(C_2, \pi') \mid \pi' \in unfold(C_1, S)\} \\ unfold(\textbf{if}\ b\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{end}, \pi) = \cup \begin{cases} unfold(C_1, \{\pi; \textbf{assume}(b)\}) \\ unfold(C_2, \{\pi; \textbf{assume}(\neg b)\}) \end{cases} \\ unfold(\textbf{while}\ b\ \textbf{do}\ C\ \textbf{done}, \pi) = \cup \begin{cases} unfold(\{C; \textbf{while}\ b\ \textbf{do}\ C\ \textbf{done}\}, \{\pi; \textbf{assume}(b)\}) \\ \{\pi; \textbf{assume}(\neg b)\} \end{cases} \end{gathered}$$

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

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 [34] triple $\{\mathcal{P}\}C\{Q\}$, 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 ($sp(path, \mathcal{P}) \models Q$) for each $path \in paths$.

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

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

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 sub-tasks 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 ($\varphi \models Q$) 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 ($sp(\pi, \mathcal{P}) \models Q$), we can use the LLM to reason directly over the path constraint and post-condition 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 3 (a) which is based on [40] Figure 7 (a.k.a. Example Program 3). 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 positive number, which can be expressed in natural language or as a formal rule ($\forall \mathsf{x}, \mathcal{R} : sp(\mathbf{read}(\mathsf{x}), \mathcal{R}) \rightarrow \mathsf{x} > 0$). Under these assumptions, the final size of `xs` should equal `n`. We can express this as the following triple:

(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. 3. (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.

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

Although conceptually simple, the Figure 3 (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 3 (b) representing two iterations of the loop. We can encode the VC ($sp(\pi_{simple}, \mathcal{P}) \models Q$) as a prompt, which is confirmed using an LLM:

“*Given the pre-condition $\mathcal{P}$ and the code π_{simple}, does the post-condition 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 3 (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* $\Pi \subseteq \mathit{unfold}(C, \epsilon)$, our approach constructs a *truncated sub-program* T_Π such that:

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

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 need 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 $\Pi \subseteq \mathit{unfold}(C, \epsilon)$, we derive a *truncated sub-program* T_Π equivalent to C for all $\pi \in \mathit{paths}$, and is *unreachable* otherwise:

$$\Pi \quad \subseteq \quad \mathit{unfold}_{\mathit{reachable}}(T_\Pi, \epsilon) \quad \subseteq \quad \mathit{unfold}(C, \epsilon) \qquad \text{(TRUNCATION)}$$

Here, ($\mathit{unfold}_{\mathit{reachable}}$) excludes all paths that terminate abnormally via a special $\mathbf{assume}(0)$ (i.e., *assume false*) statement. Thus, $\mathbf{assume}(0)$ represents a statement that is assumed to be *unreachable*.[4] 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 $\mathbf{assume}(0)$. We can represent this idea using the following rewrite rule:[5]

$$s \longrightarrow \mathbf{assume}(0) \qquad \text{if } s \notin \pi \text{ for all } \pi \in \mathit{paths} \qquad \text{(UNREACHABLE)}$$

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

$$\mathbf{assume}(0); C_2 \longrightarrow \mathbf{assume}(0) \qquad C_1; \mathbf{assume}(0) \longrightarrow \mathbf{assume}(0)$$
$$\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ \mathbf{assume}(0)\ \mathbf{end} \longrightarrow \mathbf{assume}(b); C_1 \qquad \mathbf{if}\ b\ \mathbf{then}\ \mathbf{assume}(0)\ \mathbf{else}\ C_2\ \mathbf{end} \longrightarrow \mathbf{assume}(\neg b); C_2$$
$$\mathbf{while}\ b\ \mathbf{do}\ \mathbf{assume}(0)\ \mathbf{done} \longrightarrow \mathbf{assume}(\neg b)$$

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* [70]. 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:

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

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 [70] (with $\mathit{vars}(Q)$ 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 $\mathit{vars}(Q)$. 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 3 (d) is a generalization of the infinite partition (Π) representing *all* feasible paths through the **while**-loop, including the Figure 3 (b) path (π_{simple}). An LLM can be used to verify the generalized *verification condition* (VC) encoded in natural language.

"*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), (T_{simple}, 8 lines, 48 tokens), and (π_{simple}, 16 lines, 85 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 LLM accuracy. □

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

[4] Similar to `__builtin_unreachable()` from gcc.

[5] We use the standard notation ($\mathit{lhs} \longrightarrow \mathit{rhs}$) to mean that any term matching the lhs is rewritten to the term matching rhs.

($\varphi_i \models Q$) 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 invariants 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.

3.3.2 Partitioning. Section 3.3.1 describes the construction of a *truncated slice* S given a partition (Π) that subset of paths ($\Pi \subseteq \mathit{unfold}(C, \epsilon)$). In principle, any partitioning ($\Pi_1 \cup ... \cup \Pi_n = \mathit{unfold}(C, \epsilon)$) 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: A CFG sub-graph G; reverse topological order
Output: A slice ⊆ G
Function GenSlice(G, Q):
    slice ← ∅; vars ← getVars(Q)
    for s ∈ G do
        if s modifies any v ∈ vars, or
          conditional s reads any v ∈ vars then
            slice ← slice ∪ {s}
            vars ← vars ∪ getDependencies(s)
    end
    return slice
```

Algorithm 2: Basic back-slicing 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 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, e.g., different iterations of the same loop with be distinct sequences, but will be equivalent sets (i.e., cover the same nodes). We make two key insights. First, paths with distinct coverage will have distinct truncations, since each path must 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 ($\Pi_{cov} \subseteq \mathit{unfold}(C, \epsilon)$) 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, and is illustrated in Algorithm 3. Here, the algorithm takes a CFG representation fo C, and generates a *partitioning* as a set of *representative paths* (π_{cov}) for each distinct coverage class (cov). The algorithm works are 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 (line 2), 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

```
Input: The Control-Flow Graph (CFG)
Output: A coverage-based partitioning
Globals: Coverage map (covMap), end node (end)
1  Function GenPartitions(node, pathCov, path):
2      if pathCov ∈ covMap then return ∅
3      pathCov ← pathCov ∪ {node}
4      path ← path ++ node
5      covMap ← covMap ∪ {path}
6      if node = end then return {path}
7      partitions ← ∅
8      for succ ∈ successors(node) do
9          partitions ← partitions ∪
10             GenPartitions(succ, pathCov, path)
11     return partitions
```

Algorithm 3: Path partitioning algorithm

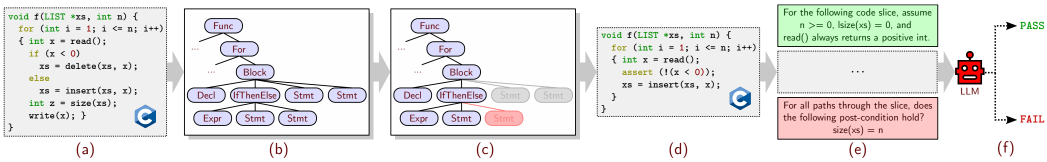

Fig. 4. Example workflow. Here, (a) is the original source code, (b) is the *Abstract Syntax Tree* (AST) parsed from the code, (c) is a generated path slice (Algorithm 3 and Algorithm 2), (d) is the slice *rendered* back into the original source code, and (e) is the generated *prompt* comprising a pre-condition, the slice, and the post-condition, and (f) is the LLM inference.

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}) which 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 $S = \langle \texttt{i} := 1; \textbf{assume}(\neg(\texttt{i} \leq \texttt{n}))\rangle$ is also a valid slice of (C_{simple}, Figure 3) 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 3 (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 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 4. 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 $\mathcal{Q}$, 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).

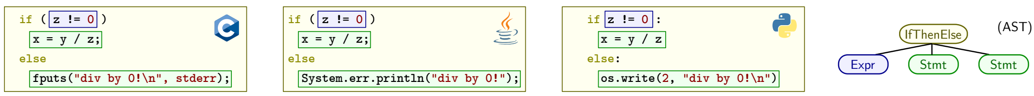

Fig. 5. Example AST representation of an *if-then-else* statement across the C, Java, and Python programming languages. Each AST node is represented *generically* by the *file range* from which the element was parsed, as illustrated by the nested colored boxes.

steps (c)-(f) are repeated for each partition using Algorithm 1, until exhaustion or the discovery of a counter-example.

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 sub-element children. The AST is *unified* in that shared language features are mapped to the same representation, as illustrated by Figure 5. 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, CPython [67] for Python, etc.), significantly simplifying the implementation.

CFG Construction. The next step *lowers* the AST into a *Control Flow Graph* (CFG), where each node represents a statement or conditional, 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 substitution 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 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 the rendered slice is coherent (re-parsable) 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 programmer intent, etc., which can assist LLM inference.

Example. An example rendered slice is illustrated in Figure 4 (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 $\{\mathcal{P}\}S\{\mathcal{Q}\}$ where $\mathcal{P}$ is a pre-condition, $\mathcal{Q}$ is a post-condition, and S is the generated slice (see Example 2.3, Example 3.2, and Figure 4 (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 AutoExe tool. AutoExe takes as input a program C in a supported programming language (currently `C`/Java/Python), pre- and post-conditions ($\mathcal{P}$ and $\mathcal{Q}$) expressed as code, constraints, or natural language. AutoExe 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, as to find the *least* counter-example to the post-condition where applicable. AutoExe is also language agnostic modulo the parser and some elements of the renderer. Furthermore, AutoExe is lightweight and approximate by design—significantly simplifying the implementation (i.e., does not rely on language-specific compiler infrastructure). This also reflects the nature of LLM inference, which is heuristical by nature, rather than relying on precise parsing and semantic analysis, and yet can still make useful inference for many real-world applications.

5 Evaluation

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

RQ1 Accuracy: What is the accuracy of AutoExe compared to existing methods?
RQ2 Scale: Can AutoExe scale to large code bases?
RQ3 Language Agnostic: Can the AutoExe workflow support multiple programming languages?

5.1 Experiments Dataset and Baseline

The dataset used in the experiments is gathered from two primary sources. The first source is the REval database [13], which consists of curated Python solutions to LeetCode problems with accompanying test suites. These subjects were selected based on the presence of multiple control flow constructs, including nested conditionals and loops, ensuring a diverse set of program behaviors that cannot be solved easily by traditional symbolic execution methods nor whole-program LLM prompting. Additionally, we curated a dataset of `C`-language solutions to LeetCode problems to complement the REval dataset and assess the tool's ability to generalize across programming languages. These C solutions were selected following similar criteria, emphasizing functions with multiple control flow branches to ensure the dataset's representativeness.

One challenge was the construction of non-trivial Hoare triples for evaluation since REval only provides the code and a test suite, while the LeetCode database we used only consists of code and problem descriptions. For this, we use two distinct methods:

- Python-Desc: Utilizing the availability of *program descriptions*, this case used the description of the problem as the natural-language post-conditions of the Hoare triple and also used the restrictions on input (if any) as the pre-condition.
- Mixed-Curated: A set of 31 subjects that consists of both Python and C programs curated from both the problem solutions in the original REval dataset and the LeetCode dataset, that comes with handwritten and nontrivial pre- and post-conditions that reflect hidden properties of the original problem and implementation.

We consider two main baselines for comparison. The first is traditional symbolic execution tools. For Python, we considered CrossHair [60], PyExZ3 [7], and pySym [8]. However, with the exception of CrossHair, most traditional symbolic execution tools are unmaintained (6 years or more, as of writing). For C, we consider KLEE [11] as a mature traditional symbolic execution tool. One practical issue with traditional tools is the lack of expressivity in the pre- and post-conditions, which must be specified in either code or a formal language. For some subjects (Python-Desc), the pre-/post-conditions are expressed in natural language, which can be handled by LLMs but not traditional tools.

For the second baseline, we consider an "ad hoc" LLM-based program analysis over the entire subject. For this baseline, we query the LLM for a counter-example (if one exists) or whether the post-condition always holds for inputs that satisfy the pre-condition. While such an approach may yield reasonable results for more straightforward programs, it lacks the systematic partitioning capabilities that are provided by our framework.

Test Models. We tested AutoExe on several open-source LLMs:

- Meta's Llama3 series [32]: Instruction-tuned models that are fine-tuned and optimized for general use cases like dialogue and chatting.
- Microsoft's Phi-4 [1]: A 14B model that is built on high-quality data from filtered public domain websites and acquired academic books, designed to aid research on language models.
- DeepSeek's DeepSeek-R1: A first-generation reasoning model coming from DeepSeek, achieving performance comparable to OpenAI-o1 [22] across math, code and reasoning tasks with much smaller parameters.
- Google's Gemma3 [65]: A lightweight family of multimodal models that features a larger context window of 128K.

Note that AutoExe is optimized for local models, since it potentially generates a large amount of prompt traffic. All experiments are run using `ollama` version v0.5.7.

5.2 RQ1: Accuracy

To evaluate the accuracy of AutoExe in generating counter-examples and verifying post-conditions, we compare the results of the baseline approach and the output of AutoExe for the above dataset. Specifically, for each program, we executed AutoExe and the baseline approach, recording their respective outputs (verified, unknown, or unverified with a counter-example) under the same prompt and LLM model. Each test is run 5 times, and the most common output is selected as the final result to mitigate random variations in the results. The accuracy of each test is measured by comparing the result against ground truth derived from the provided test suites and manual verification.

Model	Method	Python-Desc			Mixed-Curated		
		Total	Correct	Accuracy	Total	Correct	Accuracy
Llama3-8B	AutoExe	85	82	96.5%	31	23	74.2%
	Baseline	85	79	92.9%	31	22	71.0%
Llama3.1-8B	AutoExe	85	84	98.8%	31	24	77.4%
	Baseline	85	71	83.5%	31	19	61.3%
Llama3.3-70B	AutoExe	85	72	84.7%	31	27	87.1%
	Baseline	85	70	82.4%	31	22	71.0%
Phi-4-14B	AutoExe	85	74	87.1%	31	24	77.4%
	Baseline	85	72	84.7%	31	24	77.4%
DeepSeek-R1-70B	AutoExe	85	85	100.0%	31	19	61.3%
	Baseline	85	82	96.5%	31	17	54.8%
Gemma3-4B	AutoExe	85	73	85.9%	31	20	64.5%
	Baseline	85	64	75.3%	31	16	51.6%
Gemma3-27B	AutoExe	85	72	84.7%	31	20	64.5%
	Baseline	85	76	89.4%	31	23	74.2%
Average	AutoExe	85	**77.4**	**91.1%**	31	**22.4**	**72.4%**
	Baseline	85	73.4	86.4%	31	20.4	65.9%

Fig. 6. Accuracy of the baseline method and AutoExe under different datasets.

Main result. Figure 6 shows the results of evaluating AutoExe on different datasets presented above. Compared against the baseline available, the experimental results demonstrate that AutoExe generally improves accuracy compared to the baseline across the models and datasets being tested, though the extent of improvement varies. For the Python-Desc dataset, AutoExe consistently outperforms the baseline for most subjects and models. Meanwhile, on the Mixed-Curated dataset, improvement in accuracy is less obvious due to the dataset consisting of more complex programs with a lower-level language, but AutoExe still presents similar performance compared to baseline methods, even in such settings. We also note that the baseline accuracy for LeetCode-derived datasets is already high due to the relatively small size of the subjects. We examine some more realistic case studies in RQ2, where the impact of partitioning is stronger.

We also attempted to run CrossHair [60] on the Python portion of the Mixed-Curated dataset since the post-conditions are expressed in executable Python code. However, CrossHair was unable to identify any counterexample for all Python subjects—likely due to the limitations of traditional symbolic execution as well as the specific implementation. The other subjects, including Python-Desc, express the post-condition as natural language descriptions and thus are unsuitable for CrossHair (or other surveyed traditional symbolic execution tools).

> AutoExe enhances accuracy compared to the baseline methods for most subjects. AutoExe is more applicable than traditional symbolic execution tools such as CrossHair [60], and can handle pre- and post-conditions expressed in code or natural language.

Impact of different LLMs. Figure 6 above also evaluates AutoExe on multiple different LLMs, including Meta's Llama3, Llama3.1, and Llama3.3 models [32] of various parameter sizes, Microsoft's Phi-4 model [1], 4B and 27B versions of Google's Gemma3 model [65], and the 70B distilled version of DeepSeek-R1 model [22]. Interestingly, the results reveal some notable differences in performance across various LLMs, highlighting the impact of model size and specialization on the results. Among the tested models, the most significant improvement in accuracy is observed with Llama3.1-8B, where AutoExe achieves 98.8% accuracy compared to the baseline's 83.5%, highlighting the benefits of partitioning in improving the accuracy of LLM reasoning for comparatively small models. In contrast, on larger models like Llama3.3-70B and DeepSeek-R1-70B, such improvements in accuracy are more subtle, mainly due to the fact that larger models inherently have a stronger reasoning ability. This is a positive result—smaller LLMs can be run locally on consumer hardware, and AutoExe exhibits a clear benefit for these use cases. For larger models, the result is still positive, with the exception of Gemma3-27B. We expect that the relatively small size of the test subjects benefits ad hoc analysis. We shall test larger subjects in RQ2.

> AutoExe improves accuracy over the tested LLMs, where smaller LLMs benefit the most.

Method	Python-Desc	Mixed-Curated
AutoExe Min-Counter TCs	203.4 ± 75.5	113.3 ± 39.0
Baseline TCs	224.8 ± 85.9	146.9 ± 48.7

Fig. 7. The average LLM token counts (TCs) for AutoExe and baseline methods under different datasets. Each entry is reported in the form of Mean±Standard Deviation.

Prompt size. The statistics for the average *Token Counts* (TCs) across different datasets are shown in Figure 7. The TCs for each query to the LLM are calculated using `tiktoken` [36], with the corresponding Byte-Pair Encoding (BDE) token encoding (also used by GPT-4o mini [55]). In the table, we compare the average TCs for the baseline method against the average TCs for the *minimum counterexample* for each test case in AutoExe's execution using Llama3.1-8B as the LLM.

From the table, AutoExe demonstrates practical token count reduction ability compared to the baseline method—reducing the length and complexity of the prompt—with reductions of approximately 9.5% and 22.9% on average, respectively, for Python-Desc and Mixed-Curated datasets. We shall also test larger subjects in RQ2.

> AutoExe reduces the average token count needed for LLM queries to find the minimum counter-example for all tested datasets, where larger datasets benefit the most.

Case study. We present Figure 8 as a concrete example. Here, the pre- and post-conditions of the original Python function (shown in Figure 8 (a)) are indicated with the corresponding `PRE` and `POST` comments. In this example, the original function has a pre-condition 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 is listed as |`res`| ≤ |*float*(`value`)|, which does not always hold, as the given function will round up any decimal input ending in "`.5`". For example, *closest_integer*(`'1.5'`) = 2, thus violating the post-condition. We find that most small LLMs cannot correctly analyze the whole program. A typical response of GPT-4o mini [55] when fed with the entirety of the original program is:

> "*Rounding using ceil or floor will always produce a result that is at least less than or equal to the absolute value of* `value`."

The LLM then proceeds to incorrectly declare that the post-condition of the program will always be satisfied. In this instance, we speculate that the LLM is reasoning over what the program is *supposed* to do rather than what the program actually does—i.e., a form of hallucination.

In contrast, AutoExe decomposes the input program into truncated slices. Two of the generated slices of the original Python function are shown in Figure 8 (b) and (c). A key observation of

(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. 8. An example buggy Python program (a) that implements a simple function that takes a value (string) representing a number and returns the closest integer to it. If the number is equidistant from two integers, round it away from zero. The example program includes an unbounded loop (`while value[-1] == '0': ...`), interaction with standard library APIs (`ceil`, `floor`, and `abs`), and complex language constructs like list slicing. In addition, slices (b) and (c) correspond to input `"0.0"` and `"2.5"`, respectively.

those generated slices are their reduced complexity. For example, in (b), only the `elif` branch is considered, with the others truncated, forming implicit pre-conditions in the form of `assume`-statements. In (c), only the second `if` branch is assumed to be taken. In this example, we can see that AutoExe significantly reduces the complexity and size of the resulting truncated slices. Specifically, 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. With the truncated slices, the LLM correctly infers the post-condition **holds** for (b) but does **not hold** for (c). For example, GPT-4o mini outputs the following for (c) (emphasis original):

> "*There are cases (in fact, all cases where* `num > 0` *and ends in* `'.5'`*) that violate the* `POST` *condition since* `abs(res)` *will always be greater than* `abs(float(value))`*. Thus, the answer is* **No, the post-condition (**`POST`**) does not always hold**."

Since there exists a counter-example (c), we have shown that the post-condition does **not hold** for the original program (a).

We also attempted to analyze Figure 8 (a) using CrossHair [60], a traditional symbolic execution engine for Python. However, CrossHair terminates after failing to find any violation of the post-condition. We believe this is due to an incomplete search or incomplete solving of the path constraint by the underlying SMT solver (z3). In contrast, AutoExe can exhaustively enumerate all path partitions, and the underlying LLM makes the correct inferences on the corresponding slice. Even this simple case study highlights the challenges with traditional methods.

5.3 RQ2: Scale

To evaluate the ability of AutoExe to scale, we consider a large read-world project, specifically the X11 [74] client libraries and applications, of which several recent crashes were found via fuzzing [49]. For these tests, we construct post-conditions by negating the crash condition, e.g., (*ptr* `!=` `NULL`) for

Subject	Bug	Token Count			Llama3.1			Gemma3		
		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	✗	✗	✗	✗	✗	✗
Average		66436	3.07%	**0.54%**	0%	20%	**60%**	0%	20%	**60%**

Fig. 9. The token counts for file-based (File), function-based (Func), and slice-based (Slice) decompositions of real-world bugs from X11 client applications. Here (✓) means the LLM correct detects the bug, and (✗) means the token limit is exceeded, or the bug was not detected.

NULL-pointer dereference. The test suite consists of ~244K sLOC of C code (~4.96M tokens)—far too large for analysis within a single prompt—meaning that some kind of decomposition is necessary. We consider the following decompositions: (File) only relevant source files included, (Func) only relevant functions included, (Slice) the AutoExe minimum counter-example assessed against the ground truth. The results are shown in Figure 9. We consider five X11 client applications (`xinput`, `xlsclients`, `xmodmap`, `xset`, `xwininfo`).

The results are shown in Figure 9. Unsurprisingly, the slice-based decomposition (Slice) is the most effective at reducing the total token count over the naïve (File) and (Func) based methods. The reduction in the prompt size and complexity also has a noticeable impact on LLM inference. For this test, we consider two small models (Llama3.1-8B and Gemma3-4B) and see that (Slice) achieves 60% accuracy (3/5), compared to 20% for (Func) and 0% for (File). We also note that, for larger subjects, the reduction in token counts and accuracy are far more impactful than the relatively small subjects of RQ1. An example slice for `xinput` is shown in Figure 10. Once the program has been 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 the truncation and slicing, the violation is readily apparent to even small LLMs. In the non-sliced code, the bug is significantly more obfuscated by additional (and irrelevant) code. For the original code, see here and here.

```
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. 10. Example slice for `xinput`.

> The truncated slicing of AutoExe can significantly reduce the size and complexity of prompts for a real-world case study. The distilled prompts correlate to improved accuracy of LLM inference.

5.4 RQ3: Language Agnosticism

AutoExe implements a lightweight workflow that is *language agnostic*. To evaluate AutoExe's ability to analyze different programming languages, we analyze the same subjects implemented

Model	Method	Translated-C-Desc			Translated-Java-Desc		
		Total	Correct	Accuracy	Total	Correct	Accuracy
Average	AutoExe	85	**73.3**	**86.2%**	85	**76.0**	**89.4%**
	Baseline	85	66.5	78.2%	85	71.3	83.8%

Fig. 11. Accuracy of the baseline method and AutoExe under original and translated datasets.

in Python, C, and Java. For this, we use GPT-4o-mini to automatically translate the Python-Desc dataset into equivalent C and Java programs. The translated program is then gathered to form the Translated-C-Desc and Translated-Java-Desc datasets, also consisting of 85 programs. Note that the translation is not necessarily faithful, but can still be processed by AutoExe using the same evaluation framework detailed in RQ1. The results are summarized in Figure 11.

Compared against the original Python dataset, we can see there is a general decline in accuracy for both baseline methods and AutoExe. Nevertheless, AutoExe is still able to improve accuracy compared to the baseline method for the translated datasets, demonstrating its versatility in handling multi-language programs that is the direct result of its language-agnostic workflow and the inherent ability for LLM to understand multiple languages. These results are achieved with reliance on precise parsers and/or language-specific compiler front-ends.

AutoExe's language-agnostic design maintains similar performance and accuracy results for the same subjects implemented in different programming languages (C, Python and Java).

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 probabilistic reasoning on 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 truncated-slice based decomposition is a significant mitigation.

6 Related Work

Static program analysis via symbolic execution. Symbolic execution [39] 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) [68], CrossHair [60], Angr [63], S2E [16], PyExZ3 [7], etc. Unlike our approach, such traditional tools translate paths into some underlying formal language for theorem proving and thus inherit 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* [78], and *program repair* [26]. Another recent innovation is LLM-based *agents* [72], 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 combining LLMs and symbolic execution. LLM-Sym [69] 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 still inherits many of the limitations of traditional symbolic execution engines discussed in this paper. Similarly, Loopy [37] 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

Large Language Model (LLM)-based program analysis has enjoyed an explosion of application over the past few years. The ability of LLMs to reason over code *directly* has advantages over traditional static program analysis methods, such as symbolic execution, that have significant limitations regarding difficult-to-analyze programs (e.g., loops, environment, heaps, etc.). That said, LLMs also have inherent limitations. For example, since LLMs essentially use a form of *approximate* or *probabilistic* reasoning, they are not always *accurate*, which may limit their application to certain tasks. Another limitation is *scale*, where the LLM token limit restricts the size and scope of the program to be analyzed. Such problems can be mitigated using a larger LLM, but this has additional costs, such as the requirement of enterprise grade GPUs in order to run the model.

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) sub-tasks, which use less tokens (helping *scale*), and are more targeted (helping *accuracy*). We implemented our approach in the form of AutoExe—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] Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J. Hewett, Mojan Javaheripi, Piero Kauffmann, James R. Lee, Yin Tat Lee, Yuanzhi Li, Weishung Liu, Caio C. T. Mendes, Anh Nguyen, Eric Price, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Xin Wang, Rachel

Ward, Yue Wu, Dingli Yu, Cyril Zhang, and Yi Zhang. 2024. Phi-4 Technical Report. arXiv:2412.08905 [cs.CL] https://arxiv.org/abs/2412.08905

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

[3] Léo Andrès, Filipe Marques, Arthur Carcano, Pierre Chambart, José Fragoso Santos, and Jean-Christophe 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] Ken Arnold, James Gosling, and David Holmes. 2005. *The Java programming language*. Addison Wesley Professional.

[5] Asmita, Yaroslav Oliinyk, Michael Scott, Ryan Tsang, Chongzhou Fang, and Houman 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] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (May 2018), 39 pages. doi:10.1145/3182657

[7] Thomas BALL, Jakub DANIEL, and Thomas Ball. 2015. *Deconstructing Dynamic Symbolic Execution* (proceedings of the 2014 marktoberdorf summer school on dependable software systems engineering, the 2014 marktober summer school on deop ed.). Technical Report MSR-TR-2015-95. https://www.microsoft.com/en-us/research/publication/deconstructing-dynamic-symbolic-execution/ Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence, Boston, MA.

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

[9] Max Brunsfeld, Amaan Qureshi, Andrew Hlynskyi, Patrick Thomson, ObserverOfTime, Josh Vera, dundargoc, Phil Turnbull, Timothy Clem, Douglas Creager, Andrew Helwer, Rob Rix, Daumantas Kavolis, Hendrik van Antwerpen, Michael Davis, Will Lillis, Christian Clason, Ika, Tuan-Anh Nguyen, Riley Bruins, Amin Ya, Stafford Brunk, Matt Massicotte, bfredl, Niranjan Hasabnis, Mingkai Dong, Samuel Moelius, Steven Kalt, and Segev Finer. 2025. *tree-sitter/tree-sitter: v0.25.1*. doi:10.5281/zenodo.14788680

[10] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. 2023. Sparks of Artificial General Intelligence: Early experiments with GPT-4. arXiv:2303.12712 [cs.CL] https://arxiv.org/abs/2303.12712

[11] Cristian Cadar, Daniel Dunbar, and Dawson 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] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90. doi:10.1145/2408776.2408795

[13] Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin 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] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[15] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei 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] Vitaly Chipounov, Volodymyr Kuznetsov, and George 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] Edmund M. Clarke and E. Allen 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.A. 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] Patrick Cousot and Radhia 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] Patrick Cousot and Radhia 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] Leonardo de Moura and Nikolaj 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, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948 [cs.CL] https://arxiv.org/abs/2501.12948

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

[24] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (July 2019), 62–70. doi:10.1145/3338112

[25] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin 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] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei 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] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Asmita, Ryan Tsang, Najmeh Nazari, Han Wang, and Houman 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] Robert W. 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] Patrice Godefroid, Nils Klarlund, and Koushik 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

[30] John B. Goodenough and Susan L. 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

[31] Susan L. Graham, Peter B. Kessler, and Marshall K. 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
[32] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao

Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[33] R. G. 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

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

[35] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank 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.

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

[37] Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K. Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. 2023. Finding Inductive Loop Invariants using Large Language Models. arXiv:2311.07948 [cs.PL] https://arxiv.org/abs/2311.07948

[38] Arvinder Kaur and Ruchikaa 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

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

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

[41] Daniel Kroening and Michael Tautschnig. 2014. CBMC – C Bounded Model Checker. In *Tools and Algorithms for the Construction and Analysis of Systems*, Erika Ábrahám and Klaus Havelund (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 389–391.

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

[43] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley 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] Mosh Levy, Alon Jacoby, and Yoav 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] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (Dec. 2022), 1092–1097. doi:10.1126/science.abq1158

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

[47] Yacine Majdoub and Eya Ben 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] Ruijie Meng, Gregory J. Duck, and Abhik Roychoudhury. 2024. Large Language Model assisted Hybrid Fuzzing. arXiv:2412.15931 [cs.SE] https://arxiv.org/abs/2412.15931

[49] Ruijie Meng, Gregory J. Duck, and Abhik 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] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large Language Model guided Protocol Fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*.

[51] Barton P. Miller, Lars Fredriksen, and Bryan 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] Michael L. Nelson. 2005. A Survey of Reverse Engineering and Program Comprehension. arXiv:cs/0503068 [cs.SE] https://arxiv.org/abs/cs/0503068

[53] Nicholas Nethercote and Julian 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] Carlos Pacheco and Michael D. 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] Corina S. Păsăreanu and Neha 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.J. 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.C. 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] Phillip Schanely. 2017. CrossHair: Symbolic Execution for Python. https://github.com/pschanely/CrossHair.

[61] Koushik Sen, Darko Marinov, and Gul 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

[62] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry 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.

[63] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni 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

[64] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning loop invariants for program verification. *Advances in Neural Information Processing Systems* 31 (2018).

[65] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, Pouya Tafti, Léonard Hussenot, Pier Giuseppe Sessa, Aakanksha Chowdhery, Adam Roberts, Aditya Barua, Alex Botev, Alex Castro-Ros, Ambrose Slone, Amélie Héliou, Andrea Tacchetti, Anna Bulanova, Antonia Paterson, Beth Tsai, Bobak Shahriari, Charline Le Lan, Christopher A. Choquette-Choo, Clément Crepy, Daniel Cer, Daphne Ippolito, David Reid, Elena Buchatskaya, Eric Ni, Eric Noland, Geng Yan, George Tucker, George-Christian Muraru, Grigory Rozhdestvenskiy, Henryk Michalewski, Ian Tenney, Ivan Grishchenko, Jacob Austin, James Keeling, Jane Labanowski, Jean-Baptiste Lespiau, Jeff Stanway, Jenny Brennan, Jeremy Chen, Johan Ferret, Justin Chiu, Justin Mao-Jones, Katherine Lee, Kathy Yu, Katie Millican, Lars Lowe Sjoesund, Lisa Lee, Lucas Dixon, Machel Reid, Maciej Mikuła, Mateo Wirth, Michael Sharman, Nikolai Chinaev, Nithum Thain, Olivier Bachem, Oscar Chang, Oscar Wahltinez, Paige Bailey, Paul Michel, Petko Yotov, Rahma Chaabouni, Ramona Comanescu, Reena Jana, Rohan Anil, Ross McIlroy, Ruibo Liu, Ryan Mullins, Samuel L Smith, Sebastian Borgeaud, Sertan Girgin, Sholto Douglas, Shree Pandya, Siamak Shakeri, Soham De, Ted Klimenko, Tom Hennigan, Vlad Feinberg, Wojciech Stokowiec, Yu hui Chen, Zafarali Ahmed, Zhitao Gong, Tris Warkentin, Ludovic Peran,

Minh Giang, Clément Farabet, Oriol Vinyals, Jeff Dean, Koray Kavukcuoglu, Demis Hassabis, Zoubin Ghahramani, Douglas Eck, Joelle Barral, Fernando Pereira, Eli Collins, Armand Joulin, Noah Fiedel, Evan Senter, Alek Andreev, and Kathleen Kenealy. 2024. Gemma: Open Models Based on Gemini Research and Technology. arXiv:2403.08295 [cs.CL] https://arxiv.org/abs/2403.08295

[66] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL] https://arxiv.org/abs/2302.13971

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

[68] Willem Visser, Corina S. Pundefinedsundefinedreanu, and Sarfraz 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

[69] Wenhan Wang, Kaibo Liu, An Ran Chen, Ge Li, Zhi Jin, Gang Huang, and Lei Ma. 2024. Python Symbolic Execution with LLM-powered Code Generation. arXiv:2409.09271 [cs.SE] https://arxiv.org/abs/2409.09271

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

[71] Elaine J. 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

[72] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huang, Qi Zhang, and Tao 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

[73] Chunqiu Steven Xia, Yuxiang Wei, and Lingming 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

[74] X.Org Foundation. 2024. X Window System (X11). https://www.x.org

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

[76] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2024. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security. https://www.fuzzingbook.org/ Retrieved 2024-07-01 16:50:18+02:00.

[77] Xin Zhou, Sicong Cao, Xiaobing Sun, and David 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.

[78] Xin Zhou, Ting Zhang, and David 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

[79] Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software unit test coverage and adequacy. *ACM Comput. Surv.* 29, 4 (Dec. 1997), 366–427. doi:10.1145/267580.267590