PALM: Path-aware LLM-based Test Generation with Comprehension

Yaoxuan Wu
UCLA
Los Angeles, USA
thaddywu@cs.ucla.edu

Xiaojie Zhou
UCLA
Los Angeles, USA
xiaojiez6@g.ucla.edu

Ahmad Humayun
Virginia Tech
Blacksburg, USA
ahmad35@vt.edu

Muhammad Ali Gulzar
Virginia Tech
Blacksburg, USA
gulzar@cs.vt.edu

Miryung Kim
UCLA
Los Angeles, USA
miryung@cs.ucla.edu

Abstract

Symbolic execution is a widely used technique for test generation, offering systematic exploration of program paths through constraint solving. However, it is fundamentally constrained by the capability to model the target code including library functions in terms of symbolic constraint and the capability of underlying constraint solvers. As a result, many paths involving complex features remain unanalyzed or insufficiently modeled. Recent advances in large language models (LLMs) have shown promise in generating diverse and valid test inputs. Yet, LLMs lack mechanisms for systematically enumerating program paths and often fail to cover subtle corner cases. We observe that directly prompting an LLM with the full program leads to missed coverage of interesting paths.

In this paper, we present PALM, a test generation system that combines symbolic path enumeration with LLM-assisted test generation. PALM statically enumerates possible paths through AST-level analysis and transforms each into an executable variant with embedded assertions that specify the target path. This avoids the need to translate path constraints into SMT formulas, by instead constructing program variants that LLM can interpret. Importantly, PALM provides an interactive frontend that visualizes path coverage alongside generated tests, assembling tests based on the specific paths they exercise. A user study with 12 participants demonstrates that PALM's frontend helps users better understand path coverage and identify which paths are actually exercised by PALM-generated tests, through verification and visualization of their path profiles.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

LLM-assisted testing, symbolic execution

ICPC '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2482-4/26/04
https://doi.org/10.1145/3794763.3794797

ACM Reference Format:
Yaoxuan Wu, Xiaojie Zhou, Ahmad Humayun, Muhammad Ali Gulzar, and Miryung Kim. 2026. PALM: Path-aware LLM-based Test Generation with Comprehension. In *34th IEEE/ACM International Conference on Program Comprehension (ICPC '26), April 12–13, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3794763.3794797

1 Introduction

Symbolic execution is a widely adopted software testing and verification technique [4, 12, 15, 16]. It systematically explores different program paths and collects the branch predicates along each path as logical constraints, a.k.a. *path constraints*. These constraints are typically encoded as Satisfiability Modulo Theories (SMT) formulas and passed to constraint solvers for concrete test input generation.

However, symbolic execution is fundamentally limited by the symbolic encoding required for individual functions and subsequent constraint solving. As shown in Fig. 1, the method `parseFilePath` processes a list of arguments and attempts to extract the file path following the `"-f"` flag. A key branch condition in this method involves the string comparison `args[i].equalsIgnoreCase("-f")`. Constraint solvers such as Z3 [7] or CVC5 [3] lack built-in support for such operations, unless symbolic execution tool developers manually supply symbolic models for the corresponding library function, such as `equalsIgnoreCase`.

Recent studies [17, 29] have demonstrated that LLMs are very effective in automated test generation. LLMs are capable of understanding program semantics and producing test inputs.

However, LLMs lack the ability to systematically enumerate program paths. As shown in Fig. 1, the method `parseFilePath` parses command-line arguments and extracts the file path following the `"-f"` flag. GPT-4o generates several test cases that cover common scenarios, such as `{"-f","input.txt"}` and `{"-v","-f", "data.csv"}`. However, it misses the subtle edge case `{"-f", "-v"}`, where the program erroneously interprets `"-v"` as the file path due to a lack of input validation. The LLM-generated tests fail to expose this bug. We observed that *directly prompting the LLM with the entire program, without indicating which execution paths exist and how to trigger them, often leads to missed edge cases.*

Second, existing LLM-based test generation tools primarily focus on maximizing overall code coverage through automated test generation, but offer limited support for understanding which specific execution paths are exercised by each generated test. To the best of

```java
public class ArgParser {
  public static String parseFilePath(String[] args){
    boolean verbose = false;
    String path = null;
    for (int i = 0; i < args.length; i++) {
        if (args[i].equalsIgnoreCase("-v")) {
            verbose = true;
        }
        else if (args[i].equalsIgnoreCase("-f")
                    && i + 1 < args.length) {
            path = args[i + 1];
        }
    }
    return path;
  }
}
// GPT-4o generated tests
parseFilePath(new String[]{});
parseFilePath(new String[]{"-v"});
parseFilePath(new String[]{"-f", "input.txt"});
parseFilePath(new String[]{"-v", "-f", "data.csv"});
parseFilePath(new String[]{"-f", "log.txt", "-v"});
parseFilePath(new String[]{"-f"});
```

Figure 1: This code snippet parses the argument following `"-f"` as a file path. While GPT-4o generates tests covering typical cases, it misses the edge case `{"-f","-v"}`, where `"-v"` is mistakenly interpreted as the file path due to the lack of validation. This path is also challenging for symbolic execution-based testing as SMT solvers like Z3 and CVC5 do not support string operations such as `equalsIgnoreCase`.

our knowledge, *prior tools lack explicit test–path alignment, limiting the user's ability to identify missing coverage and guide LLM-driven test generation toward uncovered paths.*

In this paper, we present PALM, a test generation system with comprehension support that combines the path enumeration capability of symbolic execution with the test generation strength of LLMs, while sidestepping the symbolic modeling and constraint solving bottlenecks of traditional symbolic execution. PALM also features an interactive frontend that visualizes the symbolic execution tree, highlights path coverage, and allows users to inspect or refine test generation for specific paths. PALM's test generation consists of two phases:

Path extraction. PALM performs AST-level analysis on the subject program to systematically enumerate program paths. For each path, it automatically constructs a corresponding program variant using program transformation, which is a modified version of the original program. The variant is augmented with `assertTrue` and `assertFalse` statements that encode the branch decisions along the path. This variant provides a path-specific prompt that guides the LLM to generate the required test input. In other words, this variant directly serves as the hint for path-specific test generation.

Test generation. PALM traverses the symbolic execution tree and generates a test input for each enumerated path. Each generated test is executed on its corresponding variant to verify whether it indeed exercises the target path. The final test suite is organized within the symbolic tree, allowing users to inspect the alignment between test inputs and program paths.

In our experiments, we evaluated PALM on 124 Java programs from HumanEval-Java [14], which include complex control flows such as nested loops and which use external library functions. By systematically enumerating and extracting program paths, PALM achieves 35.0% and 24.2% higher path coverage than GPT-4o-mini and GPT-o3-mini, respectively, when using the same LLM backend. Its iterative test validation and refinement further improves path coverage by 14.2% over the non-iterative setting with GPT-4o-mini. In contrast, Symbolic PathFinder (a representative traditional symbolic executor) fails to derive a single accurate path constraint in 34.3% of the programs due to insufficient symbolic modeling of external API calls. We also conducted a within-subject user study with 12 participants to understand how much PALM can assist users in generating and comprehending path-aware test inputs. Participants reported higher confidence in their ability to generate sufficient tests, disambiguate redundant or missing tests, and check whether generated tests indeed exercise specific paths, compared to using a pure LLM as is.

The contributions of PALM are summarized below:

- We combine symbolic execution and LLM-assisted test generation by leveraging symbolic execution for systematic path enumeration through AST-level analysis, and utilizing LLMs to generate test inputs without relying on constraint solving.
- We design an interactive frontend that visualizes the symbolic execution tree, enabling users to inspect program paths and verify whether LLM-generated tests exercise the intended ones.
- We conducted a user study with 12 participants, and the results show that PALM's interactive frontend helps users better understand path coverage and identify tests that exercise specific paths.

2 Tool Overview

PALM is a test generation system for Java programs that combines symbolic execution with LLMs. For each enumerated path, PALM uses an LLM to generate a path-exercising test input. Its interface centers around a symbolic execution tree, a tree-based representation where each node corresponds to a program statement or a branch condition, and each path from the root to a leaf represents a possible execution path. The symbolic tree visualizes explored paths and highlights covered ones in green. Users can visualize the execution trace of a given test and align test inputs with path-level program execution behavior. In the following three subsections, we explain PALM's key features using the example shown in Fig. 2.

2.1 Test generation

At its core, PALM systematically enumerates program paths and invokes an LLM to generate test inputs for each enumerated path. Users begin by providing the subject program, such as a program named *tutorial* in the example figure (❶), along with symbolic execution configurations. In this example, the method *tutorial* is marked as the entry point and symbolic, meaning its internal paths will be enumerated. Although the program contains no loops, users can still configure a loop bound, such as 2, to control the loop unrolling depth in programs. After clicking the Test Generation Button (❸), PALM begins enumerating paths, constructing program variants, and requesting LLMs for path-specific test generation.

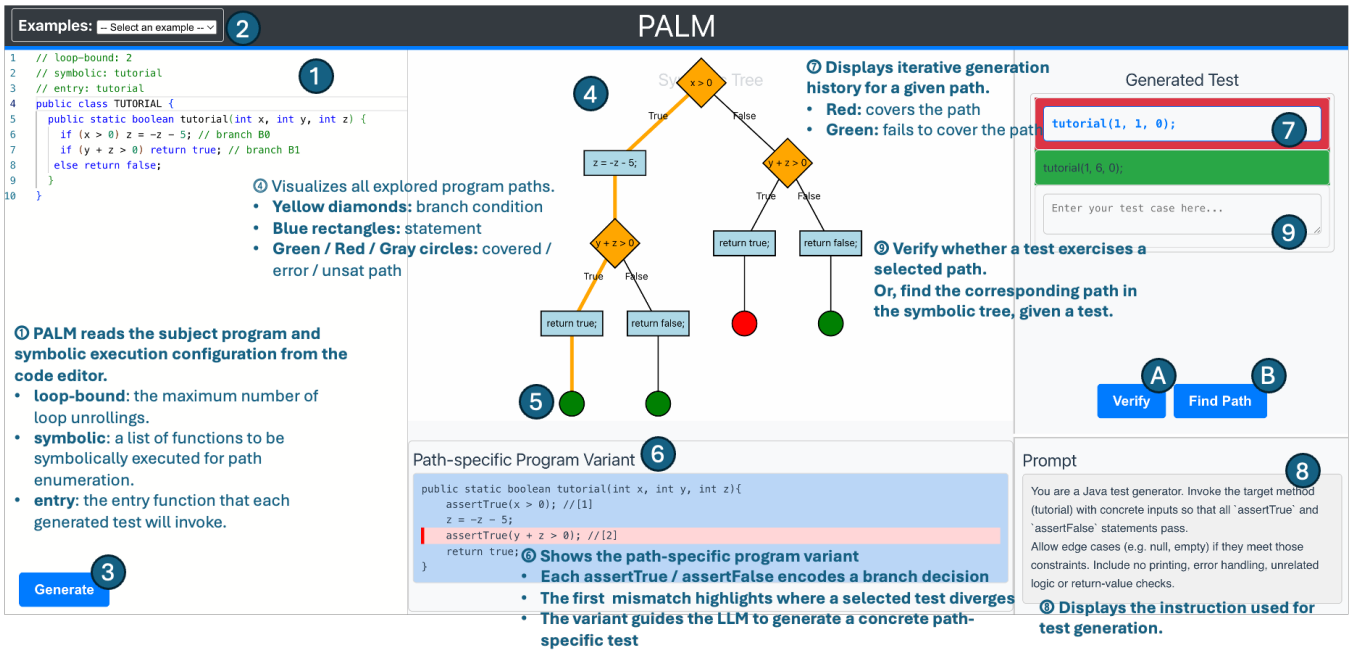


Figure 2: PALM user interface. ❶ Code editor and symbolic-execution settings. ❷ Built-in example selector. ❸ Start symbolic execution and test generation. ❹ Symbolic execution tree (leaf nodes show coverage). ❺ Select a path (click a leaf). ❻ Path-specific program variant (assertions encode branch decisions). ❼ Iterative test-generation history. ❽ Prompt for the selected test. ❾ Test editor. Ⓐ Verify whether a test exercises the selected path. Ⓑ Locate the corresponding path for a given test.

2.2 Path understanding with symbolic tree and path variant

An interactive symbolic execution tree serves as a central component in PALM. This tree visualizes the explored execution paths of the program and distinguishes branch conditions (yellow diamond nodes) from regular statements (blue rectangular nodes). It also encodes path coverage status as green (covered), red (uncovered), and gray (unreachable). For example, the highlighted orange path (❺) contains two conditional branches, `x>0` and `y+z>0`, both of which must evaluate to `true`. The non-conditional assignment `z=-z-5` appears between them and affects the outcome of the second condition, illustrating a data dependency along the path.

Once a path is selected, PALM presents its corresponding *path-specific program variant* as an executable program (❻). This variant is constructed from the symbolic execution trace, augmented with `assertTrue` or `assertFalse` statements that reflect the outcome of each branch condition along the path. In the above example, since both conditions must evaluate to true, the variant contains `assertTrue(x>0)` and `assertTrue(y+z>0)`.

The variant not only helps users interpret the logic of an execution path, but also serves as a prompt that guides the LLM to generate test inputs satisfying the conditions along that path. Unlike traditional symbolic execution tools that translate path constraints into SMT formulae, PALM leverages LLMs to handle complex language features, such as string operations and library calls, that are often inexpressible in SMT logic. This approach eliminates the need for manually modeling library functions or treating them as uninterpreted, a key advantage over conventional symbolic executors.

2.3 Validating and refining generated tests

LLMs may generate a concrete test input that diverges from the intended path. After receiving the LLM generated test, PALM verifies whether it exercises the selected path. When a path is selected (❺), PALM displays the generation history for that path. Each test is color-coded: green boxes indicate successful coverage (e.g., `tutorial(1,6,0)` covers the highlighted path), while red boxes denote failures (e.g., ❼ `tutorial(1,1,0)` diverges at the second branch `y+z>0`). When a test diverges from the selected path, PALM highlights the first failing assertion in the path variant (❻); in this example, it is `assertTrue(y+z>0)`.

Users can inspect the divergence and hypothesize the cause. For example, the LLM might ignore the update `z=-z-5` before evaluating the `y+z>0` branch. To guide the LLM, users can manually edit the prompt in the Prompt Box (❽) and inject guidance such as: *"The y+z>0 branch should consider that z is updated as -z-5."* Users can also update the test to `tutorial(1,6,0)` and verify whether it covers the intended path (Ⓐ), or identify the path actually taken by a test (Ⓑ).

In PALM's feedback-based refinement loop, the system provides the previous failing test and the first diverging assertion to the LLM to guide subsequent attempts. This helps the model pinpoint where its prior generation deviated from the target path. By default, PALM performs up to five iterations per path to produce a valid concrete path-covering test.

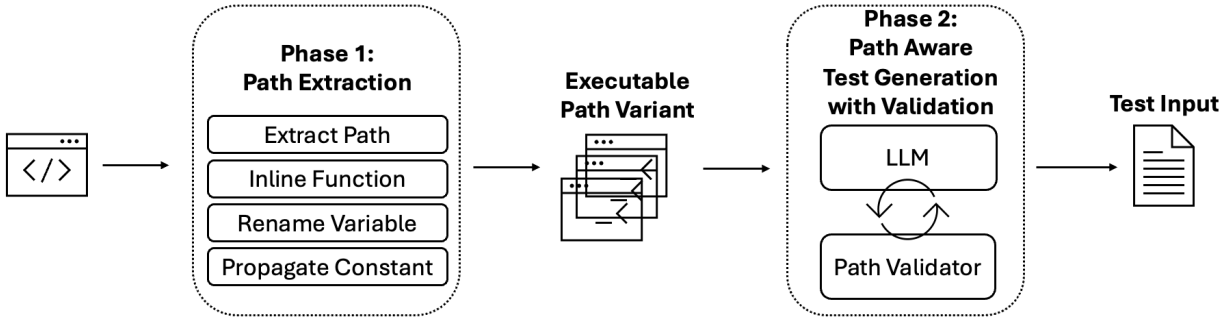


Figure 3: PALM has two phases: (1) enumerate paths and synthesize executable path-specific variants via loop unrolling, inlining, renaming, and constant propagation/folding; (2) traverse the path tree, call an LLM to generate inputs, and iteratively validate them by execution, regenerating with feedback when a test misses the intended path.

3 Methodology

PALM's backend test generation consists of two phases: (1) path enumeration and extraction and (2) path-specific concrete test generation with validation. Fig. 3 presents an overview of this workflow.

PALM takes a Java program as input and systematically enumerates its execution paths using AST-level analysis. For each enumerated path, PALM generates a corresponding path-specific program variant, which is an executable program that represents the statements and branch decisions along the path. These variants encode each path's semantics in a form of normal Java statements that is interpretable by LLMs, without translating to SMT constraints. To facilitate precise control over test generation, PALM organizes and visualizes paths into a symbolic execution tree.

In the second phase, PALM uses the constructed path-specific program variants as prompts to guide LLMs in generating concrete test inputs. Each generated test is dynamically validated against its corresponding program variant to ensure that it indeed exercises the intended execution path. When validation fails, PALM provides the LLM with feedback about the failed assertion to iteratively refine the test.

3.1 Path Extraction

PALM conducts path enumeration at the AST level and transforms each path into a program variant. Each variant encodes the evaluation outcomes of all selected branches using `assertTrue` and `assertFalse` statements, thereby preserving the semantics of the execution path.

Fig. 4 illustrates an example of path extraction. The subject program performs a palindrome check using a loop with an inner conditional. The extracted path (c) corresponds to an execution trace where the loop condition `i<len` must hold initially (❷), and the inner branch condition `text.charAt(i) != text.charAt(len-i-1)` must evaluate to `true` (❸). In contrast, path (d) corresponds to a different trace where the loop conditions must be true in two consecutive iterations (❷, ❺); the inner branch must evaluate to `false` in the first iteration (❹), and `true` in the second iteration (❻).

Algorithm 1 formalizes our path-enumeration process. It recursively traverses the program's AST and compositionally constructs the set of enumerated paths. Lines 2–5 handle `If-Then-Else` constructs by splitting into the two branches and inserting `assertTrue` or `assertFalse` based on the branch outcome. For example, ❹ and ❻ correspond to the two branches of the inner conditional check `text.charAt(i) != text.charAt(len-i-1)` in Fig. 4.

Lines 6–8 handle `While` loops by unrolling the loop body up to a user-defined loop-unrolling bound K. Other loop constructs such as `Do-While` and `For` loops are handled analogously. Line 10 processes compound `BlockStatement` structures consisting of sequential statements by enumerating the combinations of subpaths induced by each statement.

Algorithm 1 Path Extraction for each Symbolic Function

Require: Program *program*, loop bound K (default = 2)
Ensure: $Paths_m$: a set of paths for symbolic function m
1: **Procedure** `ExtractPath`(*node*) // *node* is an AST node
2: **if** *node* is `IfStatement` **then**
3: $P_{\text{then}} \leftarrow$ `PrependAll(assertTrue(`*node.branch*`),ExtractPath(`*node.thenBlock*`))`
4: $P_{\text{else}} \leftarrow$ `PrependAll(assertFalse(`*node.branch*`),ExtractPath(`*node.elseBlock*`))`
5: **return** $P_{\text{then}} \cup P_{\text{else}}$
6: **else if** *node* is `WhileStatement` **then**
7: Unroll the loop for 0 to K iterations, then apply `ExtractPath` to each unrolled version
8: **return** the union of all extracted paths
9: **else if** *node* is `BlockStatement` **then**
10: **return** combinations of `ExtractPath`(s), for all $s \in$ *node*
11: **end if**
12: ... // Handle other code structures
13: **End Procedure**

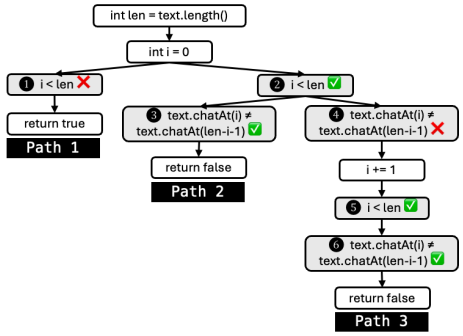
A common practice in symbolic execution is to allow annotations specifying which functions should undergo path enumeration (referred to as symbolic functions), as well as which function serves as the entry point. This allows users to focus on path exploration on functions of interest. PALM supports this flexibility as well.

Since path extraction flattens control structures and isolates a single execution path per symbolic function, key semantic information may be lost if not preserved explicitly. To address this, we apply function inlining and variable renaming to preserve the semantics of nested calls and variable scopes. Additionally, we perform constant propagation and folding to simplify each path-specific program variant, reducing syntactic complexity and easing LLM reasoning.

Function inlining. A symbolic function may invoke other symbolic functions multiple times, each leading to a different execution path. If we retain only a single path-specific variant per function, this can lead to incomplete path semantics. For example, suppose symbolic function `A` calls another symbolic function `B` twice, and `B`

```java
// symbolic: is_palindrome
// entry: is_palindrome
public static boolean is_palindrome(String text) {
  int len = text.length();
  for (int i = 0; i < len; i += 1) {
    if (text.charAt(i) != text.charAt(len-i-1))
      return false;
  }
  return true;
}
```

(a) Program under test: palindrome test



(b) Symbolic tree

```java
public static boolean is_palindrome(String text) {
  int len = text.length();
  int i = 0;
  assertTrue(i < len); ❷ // [1]
  assertTrue(text.charAt(i) != // [2]
             text.charAt(len-i-1)); ❸
  return false;
}
```

(c) Path 2's program variant

```java
public static boolean is_palindrome(String text) {
  int len = text.length();
  int i = 0;
  assertTrue(i < len); ❷ // [1]
  assertFalse(text.charAt(i) != // [2]
              text.charAt(len-i-1)); ❹
  i += 1;
  assertTrue(i < len); ❺ // [3]
  assertTrue(text.charAt(i) != // [4]
             text.charAt(len-i-1)); ❻
  return false;
}
```

(d) Path 3's program variant

Figure 4: Example palindrome checker with a `for` loop and two branches (green, yellow). (b) shows the symbolic execution tree, where each edge corresponds to a source predicate. After loop unrolling, PALM derives concrete path conditions as assertion-based encodings in (c) and (d), translating each branch to an `assertTrue`/`assertFalse` over the predicate. For example, ❸ indicates the yellow predicate evaluates to `true` in the first iteration.

Algorithm 2 Function Inlining

Require: P_m: set of symbolic paths for each symbolic function m
Require: R: recursion bound (default = 2)
Ensure: T_m: complete set of inlined paths for function m

$T \leftarrow$ paths without symbolic calls
$N \leftarrow$ paths with symbolic calls
for recursion level $r = 1$ to R **do**
  $N' \leftarrow \emptyset$
  for each non-terminating path $n \in N$ **do**
    Identify call-sites $c_1, \ldots, c_k$ in n calling functions $m_1, \ldots, m_k$
    Enumerate all path tuples $(p_1, \ldots, p_k)$ where $p_i \in T_{m_i}$
    $N' \leftarrow N' \cup$ the set of paths obtained by replacing each c_i in n with p_i
  end for
  $T \leftarrow T \cup N'$ // Prepare for next inlining level
  $N \leftarrow N'$ // Continue inlining remaining calls
end for
return T

takes different paths in each call. Without inlining, we would need to represent B with multiple variants, complicating test generation.

To address this, PALM inlines all symbolic callees within each caller's path variant. This ensures that the semantics of nested function calls are fully embedded in the caller's path, allowing PALM to construct a flattened, self-contained path variant for the entry function. Consequently, the final prompt to the LLM contains a single expanded version of the entry function with all relevant behaviors explicitly included.

Algorithm 2 formalizes this process. PALM first extracts intra-procedural path variants for all annotated symbolic functions. For each callsite in the entry function, PALM enumerates all corresponding callee paths and replaces the call with each of those inlined bodies. This ensures that all possible call behaviors are embedded and the full execution semantics are preserved, without requiring multiple variant copies of the same symbolic function.

Variable renaming. In the original subject program, variable scopes are managed implicitly by the nesting structure. However, after path extraction with control flow expansion, such as loop unrolling or function inlining, variable scopes are flattened. As a result, variable reuse can lead to name conflicts or ambiguity. To preserve semantic clarity and avoid confusion during test generation, PALM renames local variables by appending a numeric suffix.

For example, if a variable `i` is declared inside a loop and the loop is unrolled twice, the two versions will be renamed as `i_0` and `i_1`, explicitly reflecting their different iterations.

Constant propagation and folding. PALM applies intra-procedural constant propagation and constant folding. This step is performed after function inlining and variable renaming, and it only propagates numerical and boolean constants. For example, in Fig. 4, ❷ contains `assertTrue(i<len);` where `i` is initialized to `0`; this can be folded into `assertTrue(0<len)`. Constant folding also enables pruning of trivially infeasible paths, such as `assertTrue(x < y)` where `x=1` and `y=0`. These simplifications reduce the syntactic complexity of path variants, easing both the LLM's symbolic reasoning and human understanding of path semantics.

3.2 Path Aware Test Generation with Validation

Inspired by traditional symbolic execution, PALM explores paths in a depth-first order, conducting feasibility checks during exploration.

Table 1: PALM prompt template. Path constraints are encoded as `assertTrue`/`assertFalse` statements, and failures provide assertion-level feedback for iterative refinement.

Task Description
You are a Java test generator. Invoke the target method **[focal method]** with concrete inputs so that all `assertTrue` and `assertFalse` statements pass. Allow edge cases (e.g., `null`, empty) if they meet those constraints. Include no printing, error handling, unrelated logic, or return-value checks.
Input program
[imports] **[other functions]** **[path condition]**
Generation History
Round **[i]** generation: **[test code]** .
Failed assertion: **[failed assertion]** .
Output Format
[output instruction]

Test generation with a symbolic execution tree. After extracting path variants, PALM organizes them into a symbolic execution tree, where each leaf node corresponds to a complete path variant, and internal nodes represent shared prefixes between paths. PALM performs test generation by traversing this tree in a top-down fashion. Algorithm 3 formalizes this process. Line 7-12 send the corresponding path variant to the LLM for test generation. For each path, PALM allows up to five trials to generate an assertion-satisfying test and backtracks if the path is deemed infeasible. This symbolic execution tree not only facilitates test generation, but also provides the foundation for checking whether a specific path is covered and for visualizing test coverage at the path level.

When constructing the prompt for test generation, PALM retains the full implementation of class fields and non-symbolic methods from the input program. This provides essential context for the LLM to reason about (1) input-output behavior and (2) internal state transitions within non-symbolic functions. PALM excludes the implementation of external or third-party library functions not present in the input code, as modern LLMs are already well-trained on common APIs. Including such definitions would be redundant and could dilute prompt clarity by increasing its length without contributing additional semantic value. Table 1 shows PALM's prompt template during test generation.

By leveraging LLMs for test generation, PALM eliminates the need to symbolically model used library functions, a major pain point of traditional symbolic execution engines such as Symbolic Pathfinder [22]. In such engines, developers must either manually encode each function's semantics as symbolic constraints or treat them as uninterpreted.

Path exercisability validation via runtime execution. When the LLM produces a test, PALM runs it on the corresponding program variant to check whether it exercises the target path. If it fails, PALM reports the first violated assertion to the LLM for refinement. For example, for the path in Fig. 4(d), the LLM may generate `is_palindrome("ab")`. The run violates the assertion at ❻ (the second iteration should evaluate the condition to `true`). PALM feeds back `assertTrue(text.charAt(i) != text.charAt(len-i-1))`, and the LLM corrects the test to `is_palindrome("abca")`, which exercises the intended path.

Algorithm 3 Test Generation on Symbolic Tree

```
Require: root: the root node of the program's symbolic execution tree
Ensure: T: a test suite aligned with program paths
 1: Procedure GenerateTests(node)
 2: p ← the path condition from root to node
 3: if node is a leaf then
 4:     t ← attempt to generate a test for path p (up to 5 trials)
 5:     if t is not found then
 6:         return ∅
 7:     end if
 8:     return {t}                                  // Return a valid test if exists
 9: end if
10: if node is divergent then
11:     Run test generation for path p
12:     if unsatisfiable or generation fails after 5 trials then
13:         return ∅
14:     end if
15: end if
16: return ⋃ GenerateTests(c), ∀c ∈ child(node)
17: End Procedure
```

4 Evaluation

We answered the following research questions:

- **RQ1:** Does PALM's path-aware program variant generation improve path coverage?
- **RQ2:** Does PALM's path exercisability validation and error-feedback guided test generation improve path coverage?
- **RQ3:** Does PALM overcome the limitation of traditional symbolic executors (Symbolic Pathfinder) on handling external functions without symbolic modeling?
- **RQ4:** Does PALM help better understand and identify redundant or missing test cases in terms of different path profiles and produce concrete tests aligned with specific paths?

Dataset. We used 124 benchmark programs from HumanEval-Java, a Java benchmark for evaluating automated program repair tools [14]. Its programs contain various external method calls, data structure manipulations, and complex control flow.

Implementation. We implemented the path extraction component based on JavaParser [13], and developed the frontend using Vue.

Configuration. In our evaluation, we set the loop-unrolling and recursion-depth bounds to 2. We treat the method whose name matches the target class as the symbolic entry, and treat all other methods (including program-defined helpers and external API calls) as non-symbolic and excluded from path enumeration. During test generation, we enumerate up to 50 distinct paths per program (including infeasible paths), capping exploration to mitigate path explosion and bound analysis cost. We chose 50 as a practical budget that keeps the total cost bounded while covering the majority of feasible paths in HumanEval-Java. Increasing this limit leads to diminishing returns in path coverage relative to the additional LLM cost. Each experiment is run 5 times.

Table 2: Comparison of average test coverage between PALM and LLM under different LLM backends. Each configuration was executed five times to reduce variance. Reported cost represents the total for all 124 benchmarks.

Tool	Path Cov	Branch Cov	Line Cov	Cost ($)
w/ GPT-4o-mini				
PALM	663.0	198.5	948.5	0.58
LLM	491.5	187.4	912.1	0.15
w/ GPT-o3-mini				
PALM	775.8	199.4	950.6	8.52
LLM	624.6	202.8	954.8	0.63
w/ GPT-o4-mini				
PALM	746.2	201.0	955.2	6.14
LLM	811.6	203.2	959.0	0.74

```java
public static boolean any_int(double x, double y, double z) {
  if ((int) x == x && (int) y == y && (int) z == z) {
    if (x + y == z || x + z == y || y + z == x)
      return true;
  } // else-branch
  return false;
}
```

Figure 5: Code snippet from `any_int`. The condition `(int)x == x` checks whether `x` is an integer (i.e., has no fractional part). The highlighted else-branch corresponds to inputs where at least one of `x`, `y`, or `z` is not an integer. LLM-generated tests (GPT-4o-mini) fail to cover this branch, whereas PALM covers with `any_int(3.0,1.1,2.0)`.

4.1 Path-aware test generation

In this section, we evaluate the effectiveness of PALM's path-aware test generation strategy and compare to LLM's capability to generate tests. Fig. 5 shows the code snippet of `any_int`. LLM-generated tests fail to cover the highlighted else-branch, corresponding to the edge case where some inputs are non-integers. In contrast, PALM systematically enumerates program paths and generates an input `any_int(3.0,1.1,2.0)` to cover this branch.

Table 2 illustrates PALM's code coverage against LLM. PALM with GPT-4o-mini achieves a 35.0% higher path coverage, 5.9% branch coverage, 4.0 % line coverage than direct LLM generation with GPT-4o-mini. It is worth noting that PALM with GPT-4o-mini achieves higher path coverage than direct LLM generation with a stronger model GPT-o3-mini. PALM with GPT-o3-mini achieves a 24.2% higher path coverage, 1.7% lower branch coverage, 0.4% lower line coverage than direct LLM generation with GPT-o3-mini.

We also observe that, as the LLM backend becomes stronger, direct generation improves and can even outperform PALM under our current configuration: with GPT-o4-mini, direct generation achieves 8.8%, 1.1%, and 0.4% higher path, branch, and line coverage than PALM, respectively. This is partly because PALM caps exploration at 50 enumerated paths per program to mitigate path explosion. Nevertheless, PALM remains complementary via path-level guidance and test-path visualization for diagnosing missing or redundant coverage.

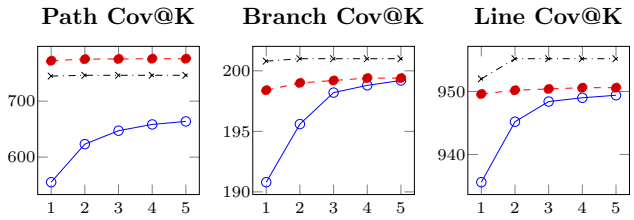


Figure 6: Test coverage progress of PALM with k rounds of trial for each program path using three LLM backends

```java
public static boolean check_if_last_char_is_a_letter(String txt) {
    String[] parts = txt.split(" ");
    String last = " ";
    if(parts.length != 0)
        last = parts[parts.length - 1];
    if(txt.length() != 0 &&
     txt.charAt(txt.length()-1) == ' ') last = " ";
    if(txt.length() == 0) last = " ";
    int last_char_pos =
      Character.toLowerCase(last.charAt(0)) - 'a';
    return (last.length() == 1) &&
        (0 <= last_char_pos && last_char_pos <= 25);
}
```

Figure 7: SPF lacks sufficient symbolic modeling for the two highlighted API calls: `split` and `toLowerCase`. The converted SMT path constraints are thus imprecise. In contrast, PALM utilizes LLM's capability to interpret the semantics of commonly used APIs.

4.2 Test validation and refinement

Fig. 6 shows the path, branch, and line coverage achieved by iteratively generated tests over K rounds. With the weaker model GPT-4o-mini, four additional rounds of test generation lead to substantial improvements: 14.2% in path coverage, 4.2% in branch coverage, and 1.4% in line coverage. The coverage gains plateau after the fifth round, indicating convergence. In contrast, the stronger reasoning model GPT-o3-mini typically achieves high coverage for each path on the first attempt, with only marginal improvements across rounds: 0.5% in path coverage, 0.5% in branch coverage, and 0.1% in line coverage.

4.3 Comparison with Symbolic PathFinder

We compared PALM against Symbolic PathFinder (SPF). As is typical for many symbolic execution engines, SPF requires symbolic modeling of used API functions or considers them as uninterpreted.

Fig. 7 shows an example, where SPF fails to model the path constraints due to two unsupported API calls and to model a symbolic heap object. Symbolic modeling is particularly challenging here because SPF would need to represent the heap object returned by `split()`, namely a `String[]` array, a type that SPF does not

Table 3: Java code patterns that lack symbolic support in Symbolic PathFinder (SPF). For these cases, SPF either terminates prematurely or falls back to concrete execution without complete path modeling.

Category	Example Code Structure	# Affected Programs	SPF Failure Reason
String Operations	`split()`, `toCharArray()`, `toLowerCase()`	39 (27.3%)	Lack of symbolic modeling for 22 commonly used String APIs, including unsupported symbolic heap representation for string arrays returned by `split()`
Type Conversion	`parseDouble()`	3 (2.1%)	No SMT-level support for string-to-double conversion predicates
Generic Types	`HashSet<?>.add()`, `HashMap<?,?>.put()`	32 (22.4%)	Hard to model method calls for generic objects, e.g. `Object.equals` and `Object.compareTo`
Other Libcalls	`ScriptEngine.eval()`, `BigDecimal.setScale()`	3 (2.1%)	Falls back to concrete execution due to missing symbolic model

currently support symbolically. Alternatively, falling back to concrete execution on `split()` loses the dependency between `txt` and `parts`, resulting in an incomplete path modeling.

Table 3 summarizes the list of library functions that have not been modeled by SPF. As a result, SPF fails to model even a single complete path in 49 programs (34.3%). In contrast, PALM does not lower path constraints into SMT. PALM adopts symbolic execution-style path enumeration and utilizes LLM to infer constraints such as `parts=txt.split(" ")`, bypassing the need for explicit symbolic modeling.

4.4 User study

To understand how PALM assists users in generating path-aware tests, we conducted a controlled user study comparing PALM with a baseline LLM-based approach. We evaluate whether PALM helps users better understand path coverage, identify redundant or missing tests, and produce concrete tests aligned with specific execution paths. We report task accuracy, completion time, and self-reported confidence for each tool. Our user study follows a counterbalanced, within-subject design. Each participant uses both tools in different orders, with tool usage and task assignment randomized and counterbalanced to mitigate learning and order effects.

Participants. We recruited 12 participants: 6 CS PhD students, 5 CS master's students, and 1 CS undergraduate student. All participants were familiar with Java and had prior experience using LLMs.

Task. To evaluate how each tool supports path-aware test generation, participants were asked to generate tests for two subject programs using either PALM or a pure LLM chatbox. As a part of test generation task, they completed three targeted comprehension questions designed to assess their understanding of (1) the total number of feasible paths, (2) the equivalence between tests in terms of path exercising behavior, and (3) the ability to match a test to a natural-language path description. Table 4 and Table 5 summarize the subject programs and associated questions.

The two subject programs, *CuteArray* and *Triangle*, exhibit key elements of path complexity. *CuteArray* contains a loop with inner loop conditions, where earlier iterations modify the array and affect later branching decisions, introducing data dependencies. *Triangle* contains nested branches that sort the inputs before performing a triangle test, requiring different input orderings to fully exercise all program paths.

Experimental Design. We adopted a 2×2 crossover design [8], where each participant used both tools (PALM and baseline) on two different programs in varied orders, evenly distributed to mitigate order bias and learning effects.

Experimental Procedure. Before the experiment, we provided participants with a brief overview of unit test generation and the concept of path coverage to ensure a shared baseline understanding. We then distributed a tutorial for PALM, introducing its features and explaining the user interface in detail. To familiarize participants with all key functionalities of PALM, we asked them to complete a warm-up task that involved selecting a path on the symbolic tree, observing the generated test, and composing their own test to verify its path exercisability on a simple tutorial program with four feasible paths, as shown in Fig. 2.

Table 4: Programs used in user study.

Subject	LOC	Paths	Description
CuteArray	12	7	Counts divisible-by-7 triplets in a loop, while modifying array elements.
Triangle	19	12	Sorts three side lengths and checks if they form a valid triangle.

Table 5: Questions used in user study.

ID	Question	Question Description
Q1	Path Enumeration	Identify the number of distinct feasible paths within a specified loop bound of 2.
Q2	Same Path Test	From a list of candidate tests, identify the one that exercises the same path as a given reference test.
Q3	Targeted Test	From a list of candidate tests, select a test that would cover a particular path described in natural language.

Table 6: Post-study questionnaire on tool confidence

ID	Question	Post-Study Question Description
P1	Path Coverage Confidence	How confident are you in the studied tool's generated tests in terms of achieving comprehensive path coverage?
P2	Redundancy Avoidance Confidence	How confident are you in the tool's capability to avoid generating redundant tests that cover the same path?
P3	Targeted Test Generation Confidence	How confident are you in the tool's capability to generate a test that indeed exercises a given specific path?

Table 7: Comparison of participant task accuracy and task completion time using different tools

Tool	Accuracy			Task Time (min)
	Q1	Q2	Q3	Avg
PALM	0.92	1	1	5.43
LLM	0.58	1	0.92	5.17

Post-study Questionnaire. We used a post-task questionnaire to obtain feedback. Table 6 lists the three questions to assess participants' perceived confidence in completing test generation tasks using PALM or LLMs. All three questions adopt a 5-point Likert scale, where 1 indicates "Not confident" and 5 indicates "Very confident."

Result. **Accuracy.** We define *accuracy* as the percentage of correct responses to three task questions, each with known ground truth. Q1 asks for the total number of feasible paths under a loop bound of 2. Q2 asks which candidate test exercises the same path as a given reference test. Q3 asks which test exercises a path described in natural language.

As shown in Table 7, participants using PALM achieved higher accuracy across all three questions: 0.92 on Q1 and 1.0 on both Q2 and Q3. The only incorrect response occurred on Q1, which assessed path enumeration, where one participant answered 6 instead of the correct 7. This participant was observed to correctly navigate the symbolic tree, which visualizes all feasible paths and allows users to inspect which ones are covered or missing, but still miscounted the total number.

In contrast, participants using the LLM chatbox struggled to enumerate feasible paths and match tests to paths. On Q1, four participants underestimated the number of feasible paths in *CuteArray* (4 vs. 7), often missing cases where the loop executes 0 or 1 iteration. On Q3, one participant selected an incorrect candidate after directly prompting the description and accepting the first response, leading to a mismatch at an inner condition due to overlooked data dependency. In comparison, PALM users could verify test-path alignment via the visualized execution trace.

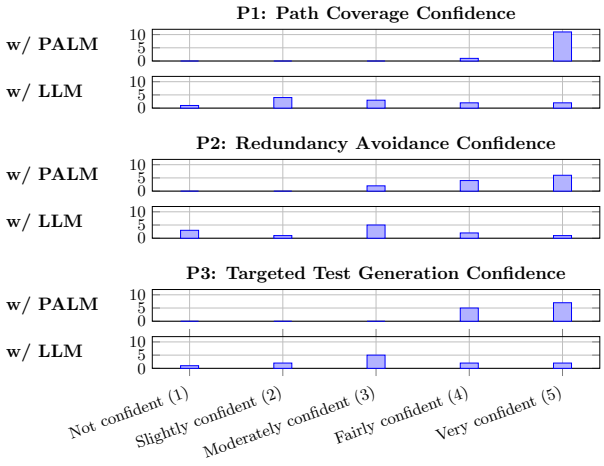


Figure 8: Post-study questionnaire results: distribution of confidence scores (x-axis) and corresponding participant counts (y-axis) for using PALM or LLM.

Completion time. Participants using PALM and participants using the LLM chatbox completed tasks in a similar time range (5.43 vs. 5.17 minutes on average).

Post-survey confidence. Figure 8 shows self-reported confidence across three aspects when using the two tools: full path enumeration capability (P1), avoidance of redundant tests (P2), and alignment between tests and target paths (P3).

Participants using PALM gave consistently high confidence ratings, with an average of 4.92 on P1 (close to very confident), 4.33 on P2, and 4.58 on P3 (both between fairly and very confident). In contrast, participants using LLMs reported noticeably lower confidence: 3.00 on P1 (moderately confident), 2.75 on P2 (between slightly and moderately confident), and 3.17 on P3 (close to moderately confident).

We observed that PALM helped participants better understand which paths were covered or missing (P1), enabling them to augment their test suites to cover missing cases. It also improved their understanding of how each test input exercises a different execution path (P2, P3), allowing them to identify which paths are covered. This was particularly useful in reducing redundant testing effort.

5 Threats to Validity

Internal validity. Our comparison with direct LLM prompting is not budget-equivalent. PALM enumerates paths and performs path-specific generation and validation, which typically requires longer prompts and higher LLM usage than one-shot prompting. Thus, PALM's coverage gains may partly reflect more generation and validation opportunities. We do not include an ablation that equalizes LLM budgets, so results reflect PALM under its intended workflow rather than a cost-controlled comparison.

For Symbolic PathFinder (SPF), we align key search settings such as the same loop-unrolling bounds. However, SPF outputs satisfying assignments rather than executable unit tests and does not synthesize JUnit-style tests from these assignments, especially for arrays or heap-allocated objects. Therefore, we do not report

SPF path coverage on HumanEval-Java; instead, we report how many programs yield at least one SMT-formatted path constraint as a coarse applicability indicator.

External validity. We evaluate PALM on HumanEval-Java, whose programs are relatively small and primarily use Java standard-library APIs. This setting may not fully represent real-world projects with larger codebases, richer third-party dependencies, and more complex behaviors arising from I/O, concurrency, reflection, and framework-driven callbacks. Moreover, our implementation targets Java, and the results may not generalize to other languages. Finally, HumanEval benchmarks may be subject to data contamination in LLM training corpora, which could inflate the absolute performance of LLM-based approaches.

Construct validity. We primarily use branch-based path coverage as the quantitative metric. This metric does not capture data-flow behaviors that do not manifest as distinct branch outcomes and may under-represent the strengths of traditional symbolic execution in generating boundary values within the same branch structure. Our user study complements coverage by evaluating users' ability to reason about path-level behaviors using PALM's interface. However, the study tasks focus on matching tests to paths and identifying redundant or missing tests, which may differ from end-to-end usage of test-generation tools in real projects. We also do not comprehensively measure other test-quality dimensions such as readability, naturalness, and bug-finding capability.

Conclusion validity. Our user study includes 12 participants, which limits statistical power and the strength of quantitative claims. Since LLM-based generation is stochastic, single-run results can be sensitive to sampling variance; we reduce this risk by repeating runs and summarizing aggregate results, but residual variance may still exist.

6 Related Work

Symbolic execution is a widely used program analysis technique for systematically reasoning about program behaviors and generating concrete inputs that witness property violations [4, 12, 15, 16]. A key challenge is modeling system and environment interactions under the constraints of solver-based reasoning [2]. DART [10] and CUTE [24] provide concolic execution to execute external functions concretely, which can under-approximate path constraints and limit exploration when important conditions reside inside those calls. To mitigate this, several systems introduce symbolic abstractions for common external APIs, e.g., KLEE's symbolic file system [6], AEG's models for file systems, network sockets, and environment variables [1], and Cloud9's POSIX abstractions [5].

In comparison, PALM avoids translating path constraints into SMT formulas and therefore does not require SMT-level symbolic abstractions for external functions. However, the current implementation enumerates paths primarily by explicit branch outcomes. In contrast, mature symbolic execution engines [22] can additionally explore paths induced by runtime exceptions and implicit checks (e.g., nullness, bounds, divide-by-zero), as well as thread-scheduling nondeterminism in concurrent programs.

Large Language Models (LLMs) have recently gained popularity for unit test generation. Prior work mainly improves generation quality by enriching prompts or adding analysis-driven feedback. For example, ChatUniTest [30] augments prompts with focal-method context and dependencies, while Vikram et al. [26] leverage API documentation to synthesize property-based tests. Other studies examine how prompt design affects coverage and bug finding [11, 18]. More analysis-guided approaches include HITS [28], which uses slicing to incrementally generate tests, and TELPA [31], which refines tests using program analysis and feedback from ineffective attempts. IntUT [21] introduces explicit test intentions to guide generation, but it does not aim to exercise specific control-flow paths.

A growing line of work combines LLMs with symbolic execution or constraint reasoning. LLMSym [27] uses LLMs to assist in translating code or constraints into SMT for test generation, while Cottontail [25] uses LLMs to help satisfy SMT-formatted constraints and structured-input validity. DeFiAligner [9] applies symbolic execution on smart-contract bytecode and uses LLMs to check consistency between symbolic summaries and documentation. SymPrompt [23] embeds branch conditions into prompts to steer LLM reasoning, and HyLLFuzz [20] invokes LLMs to propose new inputs when fuzzing stalls. AutoEXE [19] replaces SMT solving with LLM-based reasoning over code-level path constraints, but it provides limited transparency into which paths are actually exercised, especially in the presence of loops or recursion.

In comparison, PALM enumerates paths and constructs executable assertion-instrumented variants, enabling validated, iterative test generation with a visual interface to inspect path-level behavior and identify redundant or missing tests.

7 Conclusion

We introduced PALM, a test generation system that combines symbolic path enumeration with LLM-guided test generation. By constructing executable path-specific variants with embedded assertions, PALM provides hints to LLM on which input to generate to target each path, bypasses the need for symbolic modeling of all invoked API functions, and enables precise test-path alignment. Its interactive front-end provides visibility into path coverage and test behavior, helping users detect missing or redundant tests. User study participants reported higher confidence in reasoning about comprehensiveness of generated tests, disambiguating missing or redundant tests, and comprehending which test inputs exercise a specific path.

Data-availability Statement

Our implementation and evaluation artifacts are available at https://github.com/UCLA-SEAL/PALM.

Acknowledgments

This work is supported by the National Science Foundation under grant numbers 2426162, 2106838, 2106404 and 2106420. It is also supported in part by funding from Amazon and Samsung. We want to thank the anonymous reviewers for their constructive feedback that helped improve the work.

References

[1] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. 2014. Automatic exploit generation. *Commun. ACM* 57, 2 (2014), 74–84.

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.

[3] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. 2022. cvc5: A versatile and industrial-strength SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 415–442. doi:10.1007/978-3-030-99524-9_24

[4] Robert S Boyer, Bernard Elspas, and Karl N Levitt. 1975. SELECT—a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices* 10, 6 (1975), 234–245.

[5] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the sixth conference on Computer systems*. 183–198.

[6] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, Vol. 8. 209–224. https://dl.acm.org/doi/10.5555/1855741.1855756

[7] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340. doi:10.1007/978-3-540-78800-3_24

[8] Amirhossein Deljouyi, Roham Koohestani, Maliheh Izadi, and Andy Zaidman. 2024. Leveraging large language models for enhancing the understandability of generated unit tests. *arXiv preprint arXiv:2408.11710* (2024).

[9] Liangjun Deng, Qi Zhong, Jingcheng Song, Hang Lei, and Wenjuan Li. 2025. LLM-Based Unknown Function Automated Modeling in Sensor-Driven Systems for Multi-Language Software Security Verification. *Sensors* 25, 9 (2025), 2683.

[10] 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*. 213–223.

[11] Vitor Guilherme and Auri Vincenzi. 2023. An initial investigation of ChatGPT unit test generation capability. In *Proceedings of the 8th Brazilian Symposium on Systematic and Automated Software Testing*. 15–24.

[12] William E. Howden. 1977. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering* 4 (1977), 266–278.

[13] JavaParser Team. 2025. JavaParser. https://javaparser.org/.

[14] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1430–1442.

[15] James C King. 1975. A new approach to program testing. *ACM Sigplan Notices* 10, 6 (1975), 228–233.

[16] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[17] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 919–931.

[18] Vincent Li and Nick Doiron. 2023. Prompting code interpreter to write better unit tests on quixbugs functions. *arXiv preprint arXiv:2310.00483* (2023).

[19] Yihe Li, Ruijie Meng, and Gregory J Duck. 2025. Large Language Model powered Symbolic Execution. *arXiv preprint arXiv:2505.13452* (2025).

[20] Ruijie Meng, Gregory J Duck, and Abhik Roychoudhury. 2024. Large Language Model assisted Hybrid Fuzzing. *arXiv preprint arXiv:2412.15931* (2024).

[21] Zifan Nan, Zhaoqiang Guo, Kui Liu, and Xin Xia. 2025. Test Intention Guided LLM-based Unit Test Generation. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 779–779.

[22] 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*. 179–180. doi:10.1145/1858996.1859035

[23] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 951–971.

[24] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A concolic unit testing engine for C. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 263–272.

[25] Haoxin Tu, Seongmin Lee, Yuxian Li, Peng Chen, Lingxiao Jiang, and Marcel Böhme. 2025. Large Language Model-Driven Concolic Execution for Highly Structured Test Input Generation. *arXiv preprint arXiv:2504.17542* (2025).

[26] Vasudev Vikram, Caroline Lemieux, Joshua Sunshine, and Rohan Padhye. 2023. Can large language models write good property-based tests? *arXiv preprint arXiv:2307.04346* (2023).

[27] 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 preprint arXiv:2409.09271* (2024).

[28] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. 2024. HITS: High-coverage LLM-based Unit Test Generation via Method Slicing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1258–1268.

[29] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[30] Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. 2023. ChatUniTest: a ChatGPT-based automated unit test generation tool. *arXiv preprint arXiv:2305.04764* (2023).

[31] Chen Yang, Junjie Chen, Bin Lin, Jianyi Zhou, and Ziqi Wang. 2024. Enhancing LLM-based Test Generation for Hard-to-Cover Branches via Program Analysis. *arXiv preprint arXiv:2404.04966* (2024).