

PathFix: Automated Program Repair with Expected Path

Abstract—Automated program repair (APR) techniques are effective in fixing inevitable defects in software, enhancing development efficiency and software robustness. However, due to the difficulty of generating precise specifications, existing APR methods face two main challenges: generating too many plausible patch candidates and overfitting them to partial test cases. To tackle these challenges, we introduce a new APR method named PathFix, which leverages path-sensitive constraints extracted from correct execution paths to generate patches for repairing buggy code. It is based on one observation: if a buggy program is repairable, at least one expected path is supposed to replace the fault path in the patched program. PathFix operates in four main steps. First, it traces fault paths reaching the fault output in the buggy program. Second, it derives expected paths by analyzing the desired correct output on the control flow graph, where an expected path defines how a feasible patch leads to the correct execution. Third, PathFix generates and evaluates patches by solving state constraints along the expected path. To further enhance repair performance and mitigate scalability issues introduced by path-sensitive analysis, we integrate a large language model (LLM) into our framework. Experimental results show that PathFix outperforms existing solutions, particularly in handling complex program structures such as loops and recursion.

1. Introduction

The software development process often faces inevitable software defects, and a recent study reveals that about 50% of the software development lifecycle (SDLC) on Linux is consumed to rectify these defects [1]. As software keeps updating, massive defects continuously emerge. Besides, the manual screening and debugging of defects are excessively time-consuming. Hence, automated program repair (APR) becomes a critical solution to streamline program repair processes, thereby enhancing development efficiency and software robustness [2], [3], [4], [5].

Automated program repair leverages program test and analysis skills to locate the fault and construct a correctness specification, which subsequently serves as the guidance for generating and validating patches [3]. Since the specification is either a set of test cases or state constraints, traditional APR methods can be classified into two general categories: test-driven and constraint-driven. Test-driven approaches generate patches by rectifying program behavior to align with the test suite [6], [7]. Constraint-driven approaches extract state constraints from program semantics and derive the patches by solving constraints [8], [9], [10]. With the rise

of the large language model (LLM), the APR task has also been packaged as a question-and-answer (QA) task, which has achieved promising performance [11], [12], [13].

However, existing APR methods face two main challenges: imprecise specification and overfitting. First, the imprecise specification leads to a redundant search space in generating patches for faulty programs. Specifications describe correct behaviors. In traditional solutions, the specifications are typically expressed as pairs of input and expected output without a specific path trajectory [10]. Such a path-insensitive design is a trade-off for scalability. As a result, numerous invalid paths become candidates for rectifying faults. For the LLM, lack of detailed specification in prompts is also the primary reason for the repair failure [5], [13]. Second, the overfitting problem in APR happens when the generated patch is only correct according to the specification, but neglects the overall correctness of the entire program. One cause of overfitting is incomplete specification, which arises when the specification fails to cover all failure cases. This issue is particularly common in test-driven approaches that rely on manually constructed test suites [3]. The LLM-based solution encounters a similar problem due to its lack of self-validation. A study [13] evaluates the performance of ChatGPT on APR tasks and reports that it could fix 31 out of 40 buggy programs through four rounds of prompting. Meanwhile, our analysis of the 9 failing patches reveals that 7 out of 9 failures are due to overfitting issues.

In this paper, we propose PathFix, a path-sensitive APR framework that leverages precise constraints derived from correct execution paths to repair faulty programs. Unlike previous path-insensitive approaches, PathFix employs path-sensitive specifications to improve repair accuracy. However, implementing PathFix solely with a static analysis framework presents scalability challenges. To address this, we integrate LLM to enhance repair performance, offering three key advantages. First, deriving and solving path-sensitive specifications can be difficult in complex scenarios, such as loops and function calls, but LLMs can effectively mitigate these limitations. Second, the static-analysis-based framework decomposes the APR task and provides precise path constraints, which serve as more explicit prompts, enabling the LLM to follow a structured, step-by-step chain of thought (CoT). Finally, the validation step in PathFix compensates for the lack of self-validation in LLMs, helping to prevent overfitting.

From the perspective of the static-analysis-based repair framework, PathFix consists of four components, i.e., *fault path identification*, *specification inference*, *patch generation*,

and *patch validation*. First, we conduct the equivalence check between two semantically consistent programs (target and reference). The reference program serves as a correctness judge to determine all faulty paths and provide correct outputs. Second, the specification inference module derives the expected paths and extracts the path constraints. The expected path refers to the path trajectory that starts from the input conditions, extends through the patch, and arrives at the correct output. We argue that after applying an appropriate patch, at least one expected path can be found to replace the fault path to reach the correct output. Given the bug position and the correct output, we can explore potential expected paths by tracing the control flow graph (CFG). Third, we solve the path constraints and leverage the component-based synthesis (CBS) approach [14] to generate the patches. Fourth, we verify the repaired program to prevent overfitting. If any issues arise, we return to the synthesis step with one additional constraint.

From the perspective of LLM, it can improve PathFix in the specification inference and patch synthesis modules. For the specification inference module, there are two challenges to implementing PathFix. First, not all potential expected paths are valid. Therefore, we need to examine the accessibility of expected path candidates, effectively filtering out invalid paths. Also, some expected paths, such as paths within loops or multiple function calls, are too complex to summarize and solve the path constraint. To manage this complexity, we slice expected paths and use LLM to summarize constraints by combining the variable states along paths and implicit context logic. For the patch synthesis module, traditional synthesis tools (e.g., CBS) are also incapable of expressing complicated structures, such as function, struct, and compound operation. LLM can extend the representation capability of traditional synthesis tools and synthesize more complex patch expressions.

We first implement the static-analysis-based framework of PathFix. Also, we exploit the LLM to enhance the implementation of PathFix for three modules, i.e., path slicing and path pruning modules in specification inference, as well as the patch generation module. We compare PathFix with a static analysis-based method Angelix [15] and SemGraft [10] and LLM-based work [13] on the purpose-designed benchmark QuixBugs [16] and some actual bugs. By solving more precise path constraints, PathFix outperforms SemGraft with a 25/40 repair success rate (SemGraft fixes 18/40) on the QuixBugs dataset. After combining with the LLM, PathFix can solve 12 more cases. For complex program structures, PathFix with LLM performs better when the defects are located in the recursion and loop body. Meanwhile, all the defects fixed by SemGraft can be fixed more efficiently by PathFix.

In summary, we make the following contributions:

- We propose a path-sensitive automated program repair framework (PathFix) to formalize precise expected path constraints, thereby increasing the repair efficiency and addressing the overfitting issue.
- We exploit the LLM to enhance the implementation of PathFix to alleviate the scalability and constraint

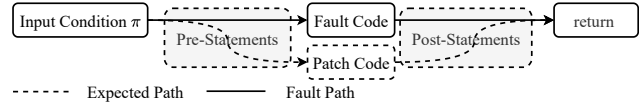


Figure 1. The expected and fault paths in the target program.

representation issues coming with precise path analysis.

- We implement the prototype of PathFix and evaluate that it outperforms the existing static-analysis-based and LLM-based solutions.

2. Preliminaries

2.1. Expected Path

If fixing a buggy program, the fault paths would be replaced by benign paths, which ensures the patched program behaves correctly. Starting from the input conditions, a benign path goes through the patched code and arrives at the correct output, as illustrated in Figure 1. We define these benign paths as expected paths (EP), which is the core idea of this paper. The specifications in previous APR methods only have input and corresponding expected output, while the specific trajectory of the expected path is unknown. This leads to inefficient and inaccurate patch generation.

Before generating a concrete patch, we can hypothetically insert a nondeterministic patch predicate into the expected path. By solving the expected path’s existence constraint, we can instantiate the predicate and obtain a usable patch. For each fault path, there should exist at least one expected path to replace it. Notably, there may exist more than one fault path in a buggy program. Correspondingly, we can generate multiple expected paths to replace them. A valid patch code should be able to satisfy all expected paths’ existence constraints simultaneously. The workflow is illustrated in Sections 3 and 4.

2.2. LLM enhanced Static Analysis and Program Repair

Leveraging extensive training data and strong semantic understanding, LLMs have demonstrated promising performance in code summarization and generation tasks. Several studies have explored using LLMs in APR task [11], [12], [13], [5]. Most of these approaches treat APR as an end-to-end question-answering (QA) task, where the buggy program is provided as input, and the LLM directly generates a patched version as output.

Despite this progress, APR is inherently a multi-step process. LLMs also exhibit several limitations, including hallucination, randomness, and an inability to self-verify [5], making such a one-hop system unreliable for ensuring patch correctness. In contrast, traditional APR methods typically follow a structured, step-by-step repair framework based on static analysis, which provides a logical reasoning chain to improve repair accuracy. At the same time, LLMs can complement static analysis-based approaches by addressing

```

1 int bin_search(int x, int a[], int len) {
2   int L = 0;
3   int R = len - 1;
4   int m;
5   do{ m = (L+R)/2;
6     if(x == a[m])
7       return m;
8     else if(x > a[m])
9       L = m + 1;
10    else
11      R = m - 1;
12  } while(L <= m); // (L <= R)
13  return -1;
14 }

```

Listing 1: Motivating Example: binary search program.

scalability challenges and assisting in complex code generation tasks [17], [18], issues commonly affecting static analysis-based APR solutions.

To leverage the strengths of both paradigms, our work integrates an LLM to enhance multiple components within our static analysis-based APR framework, improving both repair accuracy and efficiency.

3. A Motivating Example

We present a motivating example to illustrate existing methods’ limitations and outline the workflow of our proposed approach. Listing 1 shows a faulty target program, which is a binary search function to search for an element x in a sorted array $a[]$ and return its index if found, or -1 if not present. However, line 12 introduces a bug that disrupts the control flow and results in incorrect outputs.

3.1. Limitations of Existing Solutions

We apply two existing APR tools, namely, one test-driven method called Angelix [15] and one constraint-driven method called SemGraft [10], on this motivating example. We select these two methods since they are the state-of-the-art solutions that outperform previous well-known methods like Genprog [6], AE [15], and SPR [19].

Repairs by Angelix. Since Angelix requires test cases to trigger the bug, we first generate two test cases: (1) $\{x = 3, a[] = [1, 2, 3], length = 3\}$ and (2) $\{x = 0, a[] = [1, 2, 3], length = 3\}$. We observe that Angelix generates an incorrect patch (i.e., $L < 2$) when only case 1 is provided. However, even when both cases are provided, Angelix produces another plausible patch (i.e., $0 < m$). Notably, Angelix can only generate patches that pass the given test cases. To alleviate this overfitting problem, Angelix depends on the quality of additional test cases to produce a correct patch.

Repairs by SemGraft. SemGraft first uses the reference program to build the specification and the verification condition (VC). We select a linear search algorithm (Listing 2 in Appendix) as the correct reference program to fix this buggy binary search program. We observe that SemGraft also generates the plausible patch (i.e., $0 < m$). Note that the VC in SemGraft only focuses on fault paths, and it does not

TABLE 1. EQUIVALENCE CHECKING WITH SYMBOLIC EXECUTION.

Path ID	Path Input (π)	Output ($\theta_{ref}, \theta_{tgt}$)	Fault
P1	$x = a[1]$	(1,1)	✓
P2	$x = a[0]$	(0,0)	✓
P3	$x = a[2]$	(2,-1)	×
P4	$x > a[1] \wedge x \neq a[2]$	(-1,-1)	✓
P5	$x < a[1] \wedge x > a[0]$	(-1,-1)	✓
P6	$x < a[0]$	(-1,timeout)	×

verify the original benign paths after generating the patch. Thus, SemGraft also suffers from the overfitting issue. Besides, since the bug appears in the loop, SemGraft needs to contain the patch’s state in each iteration; however, multiple iterations may lead to inefficient repair performance.

3.2. PathFix Workflow

We also use Listing 1 as the target program to illustrate the workflow of PathFix. Motivated by SemGraft, we also employ the reference program to ensure we can collect all fault paths for the target program. If the reference program is unavailable, PathFix can also detect fault paths based on test cases provided by users or other tools, such as Fuzzing or symbolic execution (SE) tool. We first compare the equivalence between the target and reference programs. The comparison is under the input condition: $\{a[0] < a[1] < a[2], length = 3\}$, which simplifies path exploration in SE and aligns with test cases in Section 3.1. The results are presented in Table 1, where the failure cases are grouped into two fault paths (i.e., P3 and P6). The remaining four paths are benign, exhibiting consistent outputs between the two programs.

After identifying the fault paths, we proceed to derive the expected paths (EPs), which lead to the correct output by deploying an appropriate patch. Within the control flow graph (CFG), PathFix can analyze all possible trajectories between the fault point and all the program exit points, i.e., return statements. As shown in Figure 2, the CFG indicates four candidate paths, highlighted with different colors. To simplify the complexity of the loop, we cut the candidate paths to retain only the non-loop fragments between the fault points (node ⑦) and exit points (node ⑧ or ⑨) in the last round. Notably, the nodes ⑤ and ⑥ are included since they represent two distinct inputs for node ⑦, providing different pre-execution states. Among these four EP candidates, EP 1 (blue) and EP 2 (red) are potential alternatives for the fault path P3 in Table 1, as the reference program’s output (2) implies a correct program is expected to exit from node ⑨. Similarly, EP 3 (yellow) and EP 4 (green) are candidates for fixing the fault path P6.

For each candidate path, we formalize the variable state of each relevant statement and present the states as pre- and post-state pairs, as shown in four tables of Figure 2. Here, β denotes a nondeterministic patch predicate to be solved. The subscript of variables (e.g., L_1, m_1) indicates the number

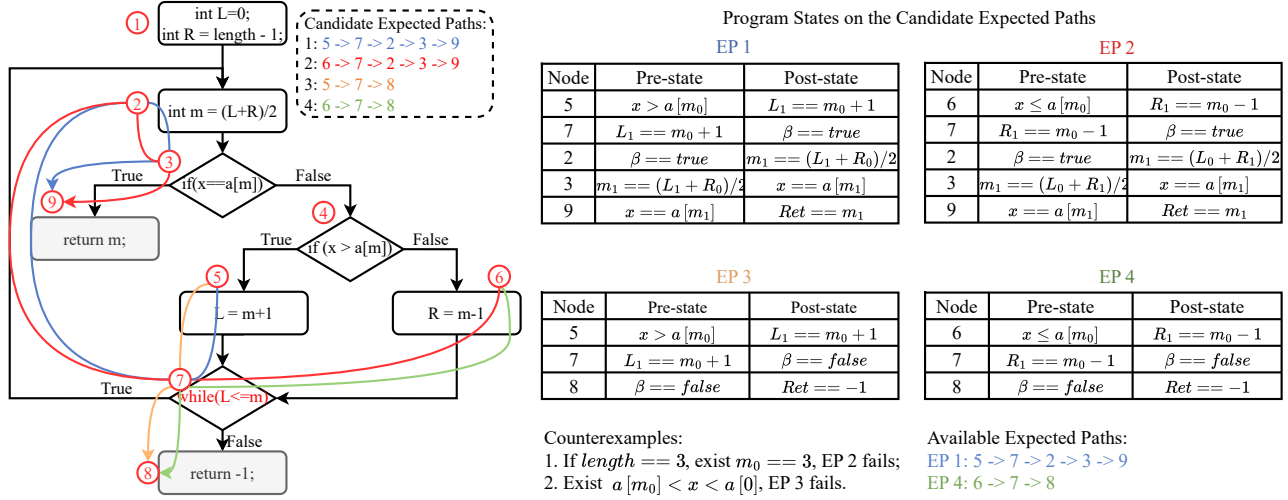


Figure 2. Specification inference (the expected paths and variable states in path constraints).

of times they have been modified along the execution path, e.g., m_1 indicates the m value after the first modification. Hence, a path constraint is formed by the conjunction of states along the expected path. In essence, generating a patch to fix a fault means finding a solution that satisfies all path constraints. However, solving the constraints directly is challenging since not all candidate paths are valid. Therefore, we must prune invalid candidate paths.

We prune invalid EPs based on the path reachability. Specifically, we examine whether the variable states on the candidate path can be satisfied. We utilize the expected return value to infer if unsatisfiable variable states are on the expected path. For instance, towards EP 2, the expected return value should be 2. However, we determine the state of node ⑥ as $m_0 = 3$, which violates the constraint of $0 \leq m \leq 2$. In the data flow, m depends on R and L , and then R depends on $length$. Thus, accessing $a[m]$ will lead to an out-of-bounds error since $length = 3$. Similarly, if EP 3 is valid, a scenario may arise where $a[m_0] < x < a[0]$, violating the constraint that the array is sorted. Hence, we exclude these two expected paths from candidates since they lead to invalid states.

After filtering invalid paths, we summarize path constraints for valid EP 1 and 4. That is the conjunction of all pre and post-states along the expected path. Then, we transform the conjunction into an implication with the patch predicate β to solve.

Given instantiated input constraints:

$$\{length = 3, a[] = \{-1, 0, 1\}\}$$

We can infer expected path constraints:

$$\exists \beta = f(L, m, R, x, a[], length),$$

$$\mathbf{EP\ 1:} \quad x = 1 \wedge x > a[m_0] \wedge L_1 = m_0 + 1 \wedge \beta = True \\ \wedge m_1 = (L_1 + R_0)/2 \wedge x = a[m_1] \rightarrow m_1 = 2$$

$$\mathbf{EP\ 4:} \quad x = -2 \wedge x \leq a[m_0] \wedge R_1 = m_0 - 1 \wedge \beta = false \\ \rightarrow True$$

(1)

In the path constraints, we instantiate x , $length$, and $a[]$ to simplify the solving process. To solve the path constraints, we adopt the z3 solver [20] and a component-based synthesis approach [14], which selects suitable variables and operators to form a concrete expression $\beta = f(L, m, R, x, a[], length)$ that meets the constraints.

We optimize the component-based synthesizer by prioritizing components (variables and operators). For example, in the first round, we provide L and m , along with logic operators, to the synthesizer, considering their involvement in the original defect. The synthesizer generates an expression: $L > m$; however, the expression fails when verifying along the fault paths. In the second round, we input L , m , and R into the synthesizer. The synthesizer generates an expression $L = R$, which successfully passes the verification on the fault paths.

We integrate LLM to address the uncertainties when pruning the expected path, summarizing constraints, and generating patches. When pruning the expected path, we need context information to determine if the path induces potential conflicts. LLM can supplement the program's extra constraint when summarizing the path constraint, such as the loop invariant. When synthesizing the patch, LLM can generate more complicated expressions, such as the function and compound operation. We leverage an LLM to address this issue. For all three tasks, our experiments indicate that LLMs provide insightful intuitive analysis, thereby yielding accurate intermediate and final results.

4. System Design

In this section, we demonstrate the framework of our PathFix system. As shown in Figure 3, PathFix consists of four modules, namely, fault path identification, specification inference, patch synthesis, and patch verification, where the LLM is adopted to enhance the specification inference and patch synthesis modules.

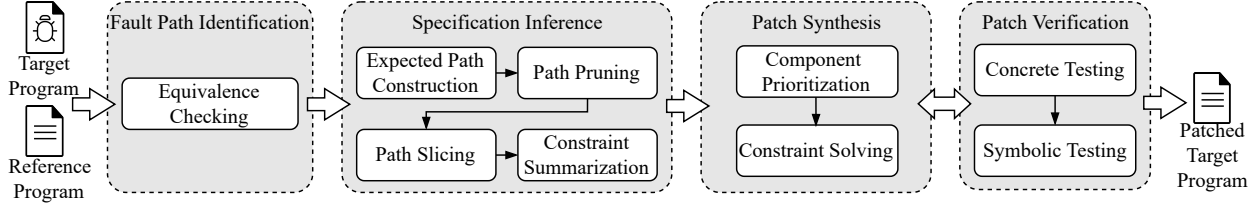


Figure 3. The static-analysis-based framework of PathFix. (LLM is plugged into 3 steps: Path Pruning, Constraint Summarization, and Patch Synthesis)

4.1. Fault Path Identification

As demonstrated in Section 3.2, we exploit the reference program and symbolic execution (SE) to identify fault paths. Notably, this step is optional for PathFix. In practice, if a reference program is unavailable, fault paths can still be identified using test cases provided by users. We design an equivalence-checking method to identify the inconsistent paths between the reference $prog_{ref}$ and target programs $prog_{tar}$. We verify the functional equivalence by providing both programs with the same symbolic input and asserting that they produce the same output, i.e., $assert(prog_{ref}(x) == prog_{tar}(x))$. For each input condition π ($x \in \pi$), the execution follows a path reaching this assertion and checks the output consistency. If a violation of the assertion is detected, the path is labeled with an inconsistency flag δ . Notably, the equivalence checking method may have limited path exploration under certain scenarios, e.g., the reference target programs might not be strictly semantically equivalent for all input conditions. Also, the existing SE tools do not support arrays of unknown length to prevent branch exploration in a loop structure [21]. To address this issue, we incorporate a user-defined pre-input condition ϕ_{pre} to constrain the exploration space for such scenarios. Finally, the output of the equivalence checking algorithm is a triplet for all paths, including the input conditions π , the output θ^{out} , and the inconsistency flag δ . The output θ^{out} comprises the reference output θ^{ref} and the target output θ^{tar} . The flag δ can be used to distinguish the benign paths and fault paths, as shown in the third and forth columns of Table 1.

4.2. Specification Inference

Then, we extract expected paths and build path constraints through 4 steps.

Step 1: Expected Path Construction. The expected path starts from the input condition π_i , extends through the patch predicates β , and arrives at the correct output θ^{ref} . If such an expected path exists, we can instantiate a patch predicate β to fix the fault; the expected path will substitute the fault path in the repaired program. Thus, the program repair is transformed into solving the expected path’s existence constraint.

We first explore all possible expected paths by searching in the control flow graph. The expected path can be categorized into three patterns based on the different exit points reached by the expected output and fault output, as

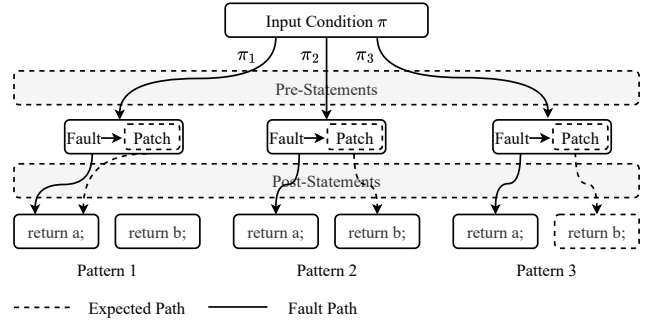


Figure 4. Three patterns of expected paths in the target program.

illustrated in Figure 4. Pattern 1 refers to the expected path reaching the same exit point (but at a different value) as the fault path. In this case, the trajectories of the expected path and the fault path may overlap completely. For example, the bug occurs in sequential statements, such as assignment statements. Pattern 2 refers to the expected and fault paths reaching different exit points. In this case, the trajectories will only partially overlap. For example, the bug occurs in the if condition or the loop condition, and the program has multiple return statements. Pattern 3 refers to the expected path reaching an exit point that does not exist before. This indicates that we need to create a new exit point. For example, if the bug causes the program to crash, we can patch it with a return as an error handler. We determine the expected path’s exit point according to the three patterns. A particular case arises when the return type of the target program is `void`. In such cases, we need to identify the implicit termination, typically the leaf node in the control flow graph or the last assignment statement before the leaf node.

We develop a matrix-representation-based path exploration algorithm to derive expected paths automatically. Specifically, we transform the control flow graph G_{ctrl} into a line graph $L(G_{ctrl})$, in which the nodes correspond to the edges of G_{ctrl} , while the edges are converted to nodes. The line graph is represented as an adjacency matrix, where two nodes are adjacent if their corresponding edges in G_{ctrl} connect to the same node. To explore possible expected paths, we integrate the node transition with a depth-first search (DFS) algorithm, starting from the fault location and ending at the expected output.

Step 2: Path Slicing. The complexity of path constraint based specification can be attributed to the long expected paths, especially multiple iterations of loop procedures, posing challenges for constraint solving. To address this

issue, we introduce the concept of **sliced expected paths** to shorten the path constraint. A sliced expected path retains a partial path containing the post-statements behind the fault statement (replacing by the patch). If the fault statement occurs in a loop, the sliced expected path only preserves the last round of the loop. Specifically, we slice the expected paths based on two observations.

First, *given a fault path, the fault impacts only the variable states after it is triggered rather than those before it*. So, we can slice the fault path behind the fault statement, hence narrowing down the scope of states included in the constraint. However, if the fault statement is within a loop body, since it is uncertain which iteration triggers the fault, it becomes hard to determine which rounds are impacted by the fault (i.e., bug-related). Nevertheless, regardless of the iteration in which the fault is triggered, the variable states in the final round must be critical to the fault. Thus, we have the second observation. *Given an expected path within a loop, if a patch fixes the fault statement, the variable states in the last iteration must be correct*.

The second observation can be inferred using the mathematical induction method. If the patched target program is proven correct, the intermediate state of the patch is always correct from the $(n-1)^{th}$ iteration to the n^{th} iteration (see detailed proof in Appendix 10.3). In cases where the fault statement is inside a loop, we only consider the acyclic path in the expected path constraint, i.e., variable states in the last iteration. Also, another reason why we only retain the last iteration is the non-deterministic nature of the expected path. Compared with the fault path, the program may no longer execute along the original path after being fixed. For example, the number of iterations and branches passed may change.

Step 3: Path Constraint Summarization. Given an expected path, we model the path constraint as a finite state machine, where each statement’s execution corresponds to the state transition between statements. Here, the states refer to the variable states before and after the statement’s execution. In this context, we use the Hoare triplet $\{P\}c\{Q\}$ [22] to depict the state transition process, where P is defined as the pre-execution state, i.e., the variable state before the current statement execution. Similarly, the post-execution state Q represents the variable state after the statement execution. c represents the operation performed by the current statement. Our work redefines operation c as the state transition predicate C . For example, supposing operation $c : m = m + 1$, we transfer it as a predicate $C : m_1 == m_0 + 1$. Thus, the modified Hoare triple can be expressed as the following implication.

$$\{P\}c\{Q\} \Leftrightarrow P \wedge C \Rightarrow Q, \quad (2)$$

The post-state (Q) can be implied by the conjunction of pre-state (P) and transfer predicate (C). This is formally different from the original inference rule in Hoare logic. We depict detailed proof that such a transition is equivalent to the original rule and how to extend such transitions into sequential execution, conditional execution, and cyclic execution in Appendix 10.2.

Taking the sequential execution (Seq) as an example, we can further extend the implication (Formula 2) to longer paths. For a path consisting of M statements, its path constraint can be expressed as follows.

$$P \wedge \bigwedge_{i \in [1, M]} C_i \Rightarrow Q \quad (3)$$

Recall that, the fault statement needs to be replaced by a patch to ensure the correctness of the execution path. Hence, we can transform the implication into an existence constraint.

$$\exists [C_t/\beta], P \wedge \beta \wedge \bigwedge_{i \in [1, M], i \neq t} C_i \Rightarrow Q \quad (4)$$

Here, existence constraints define the objective of solving the patch predicates (β). Note that, apart from the fault statement (C_t), all other statements (C_i) can be instantiated. Therefore, we can simplify these constraints ($i \in [1, M], i \neq t$) and optimize the complexity of the solution.

To formalize such constraints, we denote variable states as properties of nodes in $L(G_{ctrl})$, and each edge represents the execution of a statement. When we explore the expected paths, the adjacent node properties could be accumulated. We can track the state transition via matrix multiplication by extracting the adjacency matrix M from the line graph $L(G_{ctrl})$. The one-hop jump of nodes can be expressed as $V_{i+1} = V_i \cdot M$. V_{i+1} represents the reachable nodes of n_i after one hop. Thus, we model the line graph $L(G_{ctrl})$ as a finite state machine, where the node transition along the expected path can be viewed as accumulating variable states.

We encountered more uncertainty when accumulating these variable states in real cases. For example, there may be extra functions lying on the expected paths. If we dive into these dependency functions, it will bring redundant variable states into the path constraint, potentially leading to solving failures. Besides, since the expected path is sliced in a loop structure, the path constraint cannot describe the context information, such as the loop variant logic. Therefore, we exploit LLM to enhance the constraint summarization, as shown in Figure 6 (in Appendix). LLM can simplify the possible dependency function logic and context logic, thereby generating more comprehensive path constraints.

Step 4: Path Pruning. As demonstrated in Section 3.2, we observe that not all expected paths are valid in the context of the target program. Therefore, we need to verify the expected paths and simplify the path constraints. To prune invalid paths, we verify their accessibility. We check if the variable states along the paths satisfy the input constraints and the local variable constraints along the data flow. We also apply the graph property analysis method to the data flow graph to obtain the accessible variable list for each statement and the context of variable modification. The invalid path will be removed when a candidate path violates the data flow constraint. The detailed derivation process is showcased in Section 3.2.

In real-world cases, many invalid path candidates cannot be filtered without the global context information. Therefore, we use LLM to boost the path pruning, and one prompt example is displayed in Figure 5 (in Appendix). In addition,

another benefit of using LLM is that it can simplify the path constraints by instantiating partial variables based on the input condition and correct output values. Note that the LLM operates path pruning and constraint summarization in the reverse order. This is because the LLM does not need to explicitly generate constraints to determine a path’s validity. Moreover, if constraints are generated for invalid paths first, the LLM may misunderstand the program.

4.3. Patch Synthesis

Our synthesizer is customized based on the component-based synthesis (CBS) [23] method. Specifically, for each expected path constraint, we instantiate potential expressions that may appear in the patch predicate (β) and embed them into the constraint. These instantiated constraints then guide the synthesis of patch expressions. The synthesis process iterates until the generated expression satisfies the expected path constraints, ensuring that the patch passes verification.

The original CBS algorithm randomly selects candidate variables and operators for synthesis. To accelerate patch generation, we prioritize accessible variables and potential operators based on four principles:

- (1) Variables in the original faulty expression are considered first.
- (2) Variables modified in expected paths take precedence.
- (3) Variables modified closer to the patch location are preferred.
- (4) Other variables involved in modifications are also considered.

Operators are categorized into arithmetic, logical, and bitwise types. Heuristics guide operator selection based on patch context, for example, logical operators are preferred in loops or conditional statements.

The algorithm takes as input the instantiated specification, an initial set of variables N , and possible operator types T_{ops} . It initializes variable components C_{var} and iterates through synthesis and verification. First, the synthesis function $CBS()$ generates candidate patches. If no valid patch is found, the algorithm expands the candidate variable and operator sets. If a patch is generated, it is verified against the expected path constraints ϕ_{exp} . If the constraints are met, the synthesized patch β_{impl} is returned; otherwise, a counterexample is generated to refine the instance specification. The component lists (C_{var}, T_{ops}) are then reset to optimize subsequent synthesis iterations.

CBS is effective in synthesizing primitive expressions and operator combinations. However, it may fail when handling complex components, such as functions. To address this, we incorporate LLM to enhance patch generation, as illustrated in Figure 7 (in Appendix).

4.4. Patch Verification

While the synthesized patch satisfies the constraints of the expected paths, two key issues remain. First, the patch may introduce overfitting, failing to preserve the correctness

of the original benign paths. Second, the sliced path specification serves as a necessary but insufficient condition for overall program correctness. Therefore, additional verification is required.

To address these issues, our verification process consists of two interconnected components. The first involves symbolic execution on the faulty paths, which is integrated into the patch synthesis process and iteratively collaborates with the synthesizer to refine the patch. The second focuses on verifying the patched program against its original benign paths to prevent overfitting. To improve efficiency, we leverage test cases generated during the equivalence checking phase to validate benign paths. If a test case fails, the synthesis process is re-entered, using the failing case as an additional constraint to guide subsequent patch generation.

4.5. LLM Enhancement

Building on the static-analysis-based framework, we integrate LLM to enhance PathFix’s performance in three key steps: invalid path pruning, constraint summarization, and patch generation, aligned with the design in Figure 3. We structure these steps into three progressive tasks, where responses from earlier steps inform subsequent ones, facilitating a logical chain of thought. The prompt templates, with placeholders (), are adapted for each case. We list the prompt setting for three steps in Appendix 10.4.

For instance, in Listing 1, we provide the function definition and expected paths to identify invalid paths, along with fault path conditions to aid constraint summarization. Additional information, such as function dependencies or struct definitions, is supplied as needed when relevant to the expected paths or patch expression. Comparing LLM’s responses with static analysis results, we observe that LLM reasoning aligns more closely with human intuition. In patch generation, instead of directly solving constraints in a strict format, LLM analyzes the general logic to ensure β satisfies the expected path constraints.

However, two challenges arise. First, a simple prompt template does not guarantee consistent performance across different cases. Second, LLM responses often lack a standardized format, sometimes providing unstructured analytical content or generating patch expressions at incorrect locations.

To address these issues, we apply two strategies. First, we encode examples of expected path constraints into the system prompt, leveraging few-shot in-context learning to help LLM recognize patterns and improve constraint summarization. Second, we fine-tune the model’s temperature setting, which controls randomness and reasoning ability. Lower temperatures (0–0.5) degrade performance, so we retain the default (1.0 in GPT-4o) to ensure high-quality responses.

Additionally, we enforce structured output formatting. Expected paths are stored in JSON, path constraints in Z3 format, and patch expressions in C syntax. This ensures consistency and facilitates further automated processing.

TABLE 2. REPAIR PERFORMANCE ON REAL-WORLD BUGS.

Buggy Prog.	Ref. Prog.	PathFix	SemGraft	Angelix	PathFix w/o Ref.	
BusyBox	sed	sed of GNU sed	✓	✓	✓	✓
	sed	sed of GNU sed	✓	✓	×	✓
	sed	sed of GNU sed	✓	✓	×	✓
	sort	sort of Coreutils	✓	✓	×	✓
Coreutils	mkdir	mkdir of Busybox	✓	✓	×	✓
	mkfifo	mkfifo of Busybox	✓	✓	×	✓
	mknod	mknod of Busybox	✓	✓	×	✓
	copy	copy of Busybox	✓	✓	✓	✓
	md5sum	md5sum of Busybox	✓	✓	✓	✓
	cut	cut of Busybox	✓	✓	×	✓

5. Evaluation

In this section, our evaluation mainly investigates 3 research questions:

- RQ1:** How well does PathFix perform, compared with the previous method?
RQ2: How much enhancement can LLM bring to PathFix for each steps?
RQ3: What are the reasons for the repair success and failure in PathFix?

5.1. Experimental Setup

Dataset. We conduct experiments on QuixBugs [16] and 10 real bugs in the busybox and coreutils. First, QuixBugs contains 40 programs of classic algorithms. We choose programs sharing similar functionality to serve as reference programs, e.g. mergesort and bucketsort. Second, we also evaluate PathFix under real-world scenarios by collecting 10 bugs from Busybox and GNU Coreutils, which are also evaluated by SemGraft ¹.

Comparison Setting. We compare PathFix with SemGraft and Angelix. SemGraft relies on reference programs and the CBS synthesizer, while Angelix relies on test cases. We also compare the success repair rate with the pure LLM-based solution, in which the user feeds the buggy code and fault cases into LLM and requires the LLM to generate a patch in a one-step conversation. Notably, we use pass@1 metric to denote the success repair rate for the LLM-based solution.

Environment Setting. All the experiments are performed on a Ubuntu 22.04 server with an Intel Xeon 2620 CPU at 2.4 GHz and 16 GB RAM. PathFix is implemented in Python with dependency on existing tools, including Klee, Joern, Z3, and Brahma. Regarding LLM, we adopt the gpt-4o API.

5.2. Performance Comparison

Comparison on Real Bug Cases. We make comparisons on 10 real programs. As shown in Table 2, both PathFix with LLM and SemGraft repair all bugs successfully, but Angelix only fixes 3 bugs. Notably, we also collect complete test suites for Angelix and PathFix w/o Ref. The results show that PathFix can still generate correct expected paths and patches without a reference program. Regarding repair

1. SemGraft presents commit IDs and project versions for 12 real bugs; however, we find 2 commit IDs are invalid.

TABLE 3. REPAIR PERFORMANCE ON QUIXBUGS BENCHMARK.

Method	Generated Repairs			
	Fixed	Overfitting	Synthesis Error	Constraint Error
SemGraft	18	3	6	13
LLM	31	7	2	0
PathFix w/o LLM	25	0	10	5
PathFix w/ LLM	37	0	0	3

efficiency, PathFix outperforms SemGraft on repair time and the number of paths included in the constraint. PathFix required 27 minutes to repair these real bugs, while SemGraft required 45 minutes. SemGraft claimed it needs to traverse up to 250 paths during inferring specification in a buggy program, while PathFix with LLM only needs to consider up to 4 paths after path pruning. That is because SemGraft needs to consider more intermediate states in its verification condition, especially inferring β of the predicate β in multi-loops. While PathFix only contains the β state in the last round.

Comparison on Quixbugs. We further evaluate PathFix’s performance on the Quixbugs. Table 3 demonstrates the repair performance comparison on Quixbugs. Among the 40 buggy programs, SemGraft only fixes 18 programs. Even without LLM enhancement, PathFix performs significantly better than SemGraft, fixing 25 programs. For the programs fixed by SemGraft, PathFix can also fix all of them. The pure LLM solution can fix 31 cases. In 9 failure cases, there are still 7 overfitting errors and 2 synthesis errors. In this context, overfitting errors mean that LLM generates an executable program, but it can only fix partial fault paths or bring new faults. Synthesis errors mean that the patched programs generated by LLM cannot be executed. For PathFix with LLM, we observed that PathFix performs best compared to all 3 baselines. There are only 3 failure cases due to the constraint errors, as listed in Table 5.

In summary, both PathFix with LLM and SemGraft can repair real-world bugs, and PathFix exhibits higher efficiency. With the LLM enhancement, PathFix further outperforms SemGraft and pure LLM-based solutions. Such an observation supports the idea that more precise constraints can produce more accurate patches. Besides, the task decomposition contributes to the performance of LLM on APR tasks.

5.3. Enhancement from LLM

To investigate how LLM helps PathFix improve the repair performance, we measure the LLM’s contributions in the 3 subtasks: path pruning, constraint summarization, and patch synthesis. As illustrated in Table 4, The findings underscore an overall trend of enhanced performance by integrating the 3 steps.

First, we observed that LLM can prune all invalid expected paths of 23 cases. As we mentioned in Section 4, expected path pruning is complicated, especially when the expected path is sliced due to the loop structure. Without

TABLE 4. COMPARISON OF REPAIR PERFORMANCE WITH AND WITHOUT LLM INTEGRATION.

Method	Path Pruning	Constraint Summarization	Patch Synthesis
PathFix w/o LLM	9	17	25
PathFix w/ LLM	23	35	37

LLM, PathFix can only prune invalid expected paths of 9 cases when the program structure is simple. These undetected invalid paths caused multiple iterations or synthesis errors. Even if 100% coverage cannot be achieved, LLM can also prune partially invalid paths in 15 cases. Compared with PathFix without LLM, it brings better accuracy and efficiency.

Second, we observed that LLM can successfully summarize precise constraints in 35 cases. We also noticed that even if some invalid expected paths are fed into this step, the LLM may still summarize correct constraints. LLM corrects these constraints based on the context information; thereby, generated constraints don't follow the invalid expected path but another valid path, fixing the same fault path.

Third, we observed that LLM can synthesize correct patches in 37 cases. In contrast, PathFix can only summarize constraints and synthesize correct patches in 25 cases. The most significant improvement happens in programs involving buggy function calls, as shown in Table 5. However, there are still 3 failure cases: 2 failed in path pruning, and 1 failed due to an incorrect path constraint.

In summary, LLM can enhance all 3 steps in PathFix and contribute to the overall performance. We noticed that LLM is a resilient system. Even if some invalid expected paths bypass the path pruning, the final patches generated by LLM are still correct. This is because LLM generates the patch based on the expected paths and the context information of the entire buggy program, e.g., the loop or recursion invariant logic.

5.4. Success and Failure Reason Analysis

We first investigate three failure reasons, as shown in Table 3. For the overfitting issue, PathFix can avoid plausible patches due to its verification on all paths; however, SemGraft generates 3 plausible patches because it only focuses on the fault path. The pure LLM-based solution fails in 7 cases due to the hallucination issue. An example of a coincidence is, in the function `detect_cycle(Node* node)` (Listing 3 in Appendix), SemGraft and pure LLM generate the same patch `h==NULL`, which can fix the bug but cause new issues when `h->successor==NULL`. However, we found that LLM realized this buggy condition in its thought, but still generated the wrong patch. PathFix also generates a similar candidate in the initial round, but it fails to pass the verification on the benign paths, so the correct patch is regenerated in a later round. The second type of failure occurs when the synthesizer cannot generate a patch, either due to timing out or failing to converge. Limited by the capabilities of the component-based synthesizer, the synthesizer may still fail to find a suitable expression after traversing all

TABLE 5. REPAIR PERFORMANCE ON DIFFERENT BUG POSITIONS.

Defect Position		PathFix w/o LLM		SemGraft		PathFix w/ LLM	
		Fixed	Unfixed	Fixed	Unfixed	Fixed	Unfixed
Assignment	sequence body	3	0	3	0	3	0
	loop body	6	5	4	7	10	1
	recursion body	5	3	2	6	7	1
	if branch in loop	1	0	1	0	1	0
Condition	for condition	3	0	3	0	3	0
	while condition	3	0	3	0	2	1
	if condition	0	1	0	1	1	0
	if condition in loop	3	1	2	2	4	0
	if condition in recursion	1	0	0	1	1	0
Function	recursion entry	0	5	0	5	5	0

accessible variables and operators. Also, the path constraint is not precise enough, which is another reason because there may be an invalid expected path that is not pruned. PathFix fails to synthesize on 10 programs, and SemGraft fails on 6 programs. These programs usually require complicated patch expressions, such as multiple variables, ternary operators, and built-in functions. Third, no valid constraint can be constructed due to the complex program context. Complex context mainly refers to multi-hop function calls, multi-layer nested loops, and recursive functions. PathFix fails on 5 programs, and SemGraft fails on 13 programs.

We further analyze the impact of defect type and location on repair performance. As shown in Table 5, Most defects in the QuixBugs are located in the loops or recursion structures. Correspondingly, the main reason for the high failure rate of SemGraft in the loop body is state explosion. SemGraft considers patch (β) states in each loop round, indicating the high constraint-solving overhead. For instance, the program `shortest_path_lengths` (Listing 6 in Appendix) contains a three-layer loop. Even if we limit the input graph containing only 5 nodes, the execution may reach up to 125 rounds, which means 125 (β_{1-125}) states in the constraint of SemGraft, thereby leading to 125 potential paths. To alleviate the complexity caused by the loop structures, PathFix constructs the sliced expected paths by intercepting the last round of the loops. The execution mechanism for the recursion functions is that the first self-invocation would be executed in the last round. In other words, the first self-invocation is equivalent to the last round in the loop. Therefore, we retain the first self-invoked execution flow to construct the sliced expected paths. However, neither PathFix nor SemGraft can generate correct patches at recursion entry. Recursion entry refers to the fault that occurs on input parameters of the recursive function. Such failures indicate that the improvement in path pruning and constraint summarization is urgent for PathFix.

In contrast, PathFix with LLM significantly improves the repair performance when the error occurs in these challenging locations, i.e., the loop and recursion bodies. When the expected path is sliced, the path constraint may be under-approximated. Without LLM, PathFix may fail to solve such constraints. In this case, LLM can supplement the constraints according to the loop logic when summarizing the path constraints to generate patches better. For recursive entry, LLM's main contribution is the representation of

function entry, which is not possible with component-based synthesis methods. However, LLM may make mistakes when handling programs with similar function names and implementations. For example, one failure case occurs when processing the `find_first_in_sort()` function (Listing 5); the implementation is similar to `find_in_sort()` (Listing 4). When summarizing constraints, LLM needs to additionally consider the situation where the array contains duplicate elements, but it constructs incorrect constraints similar to these in `find_in_sort()`.

6. Discussion

In this paper, we collect the reference program before using PathFix. We assume that the reference program is bug-free, ensuring the completeness of detected fault paths. However, our evaluation demonstrates that a reference program is not strictly required for PathFix. Instead, test cases can be used to infer fault paths, shifting the responsibility to users to provide comprehensive test coverage. We also explored leveraging LLMs to generate reference programs. For instances in the QuixBugs dataset, the LLM successfully generated 30 correct reference programs. However, since the reference program must be entirely correct, the 75% accuracy rate (30 out of 40) is insufficient for reliable use. Additionally, we experimented with using LLMs to detect fault paths without a reference program, but this approach also failed to guarantee complete path coverage. In future work, we plan to optimize prompt design to improve accuracy.

Another limitation is that the sliced expected path represents an upper approximation of the path constraint. The constructed path constraints form a superset of the expected paths, disregarding the influence of statements preceding the sliced path. As a result, the synthesis process may fail to converge. To address this, we plan to incorporate loop invariants to extend path constraint coverage for loop variable states. Prior research [17] has shown that LLMs are effective in summarizing invariants, which could further enhance our approach.

7. Related Work

Test-Driven Approaches. Test-driven approaches rely on test suites to form correctness criteria, guiding fault localization, patch generation, and verification, such as GenProg [6], AE [24], RSRepair [25], and ACS [26]. GenProg [6], [27] a landmark work. It uses genetic programming to search for repairs in buggy programs. The performance bottleneck lies in the efficiency of the search algorithm and precise coverage of the candidate space, inspiring continuous improvement in subsequent work [25], [24]. RSRepair [25] shows random search outperforms genetic algorithms in patch search. AE [24] optimizes the search space by merging semantically consistent but syntactically different mutations. APR tools are also integrated into CI/CD pipeline in industry, e.g., Meta’s SapFix tool [28], which achieve good performance leveraging high-quality internal codebases.

Semantics-Driven Approaches. Semantics-driven methods infer constraint-based specifications that formally cover fault conditions [3]. The key insight lies in constructing more precise constraints. One straightforward way is to summarize constraints from test cases. Semfix [8] formulates given test cases into repair constraints using symbolic execution. Another approach builds the constraint based on user’s experience [9], [7], [29]. DirectFix [9] proposes that the best repair involves the least changes and then optimize the patch by minimizing changes. Property-based APR [29] summarizes specific safety properties for different bug types. The third way is to learn from the correct equivalent code, which provides a functional reference. Relifix [30] utilizes previous code to repair bugs in a new version; however, it fails to cover new features due to its reliance on syntactic similarity. SemGraft [10] extends the similarity of reference program from syntactic consistency to semantic consistency. **AI-Driven Approaches.** With the rise of large language model (LLM), the AI-based APR has shown promising results [11], [12], [13], [5]. We also noticed that with the rise of the large language model (LLM), applying LLM to the APR task has also achieved promising performance. Coconut [11] firstly applies a machine translation architecture to build the mapping from buggy code to benign code. Cure [12] improves performance by fine-tuning Generative Pre-trained Transformer (GPT) [31]. The commercial services like Codex [32] and ChatGPT [33] have also demonstrated outstanding APR performance. With the improvement of LLM’s code generation capability, current LLM-based code repair focuses on prompt engineering to improve repair capabilities. Misu *et al.* [34] exploited the LLM to boost the formal verification based on Dafny language, which is usually used in APR tasks. However, we argue that AI-based APR is not ready to completely replace test-driven and semantics-driven approaches. First, Code generated by LLM, including patches, still relies on human validation due to the unreliable nature of LLM. Second, AI models are limited in context length, hindering large project repairs.

8. Conclusion

In this work, we introduced PathFix, a novel automated program repair framework designed to address two significant challenges in existing APR methods: imprecise specification and overfitting issues. By leveraging static analysis techniques, including symbolic execution and constraint-solving, PathFix formalizes expected path constraints, enabling more precise and efficient patch generation. We integrate LLM to improve scalability and constraint handling in PathFix, facilitating repair performance in complex bug scenarios such as loops and recursions. The evaluation of PathFix demonstrates its superiority over static analysis-based and LLM-based solutions, highlighting its potential to improve automated program repair processes and promote more reliable software development practices.

9. Acknowledgment

We use ChatGPT-4o to polish the expression and reduce grammatical errors, mainly in the Introduction Section.

References

- [1] L. Gazzola, D. Micucci, and L. Mariani, “Automatic software repair: A survey,” *IEEE Trans. Softw. Eng.*, vol. 45, no. 1, p. 34–67, jan 2019.
- [2] R. Alur, R. Singh, D. Fisman, and A. Solar-Lezama, “Search-based program synthesis,” *Commun. ACM*, vol. 61, no. 12, p. 84–93, nov 2018.
- [3] C. Le Goues, M. Pradel, A. Roychoudhury, and S. Chandra, “Automatic program repair,” *IEEE Software*, vol. 38, no. 4, pp. 22–27, 2021.
- [4] E. Winter, V. Nowack, D. Bowes, S. Counsell, T. Hall, S. Haraldsson, and J. Woodward, “Let’s talk with developers, not about developers: A review of automatic program repair research,” *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 419–436, 2023.
- [5] S. B. Hossain, N. Jiang, Q. Zhou, X. Li, W.-H. Chiang, Y. Lyu, H. Nguyen, and O. Tripp, “A deep dive into large language models for automated bug localization and repair,” *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, jul 2024.
- [6] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, “Automatically finding patches using genetic programming,” in *2009 IEEE 31st International Conference on Software Engineering*. Vancouver, BC, Canada: IEEE, 2009, pp. 364–374.
- [7] D. Kim, J. Nam, J. Song, and S. Kim, “Automatic patch generation learned from human-written patches,” in *2013 35th International Conference on Software Engineering (ICSE)*. San Francisco, CA, USA: IEEE, 2013, pp. 802–811.
- [8] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, “Semfix: Program repair via semantic analysis,” in *2013 35th International Conference on Software Engineering (ICSE)*. San Francisco, CA, USA: IEEE, 2013, pp. 772–781.
- [9] S. Mehtaev, J. Yi, and A. Roychoudhury, “Directfix: Looking for simple program repairs,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, vol. 1. Florence, Italy: IEEE/ACM, 2015, pp. 448–458.
- [10] S. Mehtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury, “Semantic program repair using a reference implementation,” in *The 40th International Conference on Software Engineering*, ser. ICSE ’18. New York, NY, USA: ACM, 2018, p. 129–139.
- [11] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, “Coconut: Combining context-aware neural translation models using ensemble for program repair,” in *The 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: ACM, 2020, p. 101–114.
- [12] N. Jiang, T. Lutellier, and L. Tan, “Cure: Code-aware neural machine translation for automatic program repair,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. Madrid, Spain: IEEE/ACM, 2021, pp. 1161–1173.
- [13] D. Sobania, M. Briesch, C. Hanna, and J. Petke, “An analysis of the automatic bug fixing performance of chatgpt,” 2023.
- [14] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, “Oracle-guided component-based program synthesis,” in *The 32nd ACM/IEEE International Conference on Software Engineering*, ser. ICSE ’10. New York, NY, USA: ACM, 2010, p. 215–224.
- [15] S. Mehtaev, J. Yi, and A. Roychoudhury, “Angelix: Scalable multiline program patch synthesis via symbolic analysis,” in *The 38th International Conference on Software Engineering*, ser. ICSE ’16. New York, NY, USA: ACM, 2016, p. 691–701.
- [16] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, “Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge,” in *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, ser. SPLASH Companion 2017. New York, NY, USA: ACM, 2017, p. 55–56.
- [17] K. Pei, D. Bieber, K. Shi, C. Sutton, and P. Yin, “Can large language models reason about program invariants?” ser. ICML’23. JMLR.org, 2023.
- [18] H. Li, Y. Hao, Y. Zhai, and Z. Qian, “Enhancing static analysis for practical bug detection: An llm-integrated approach,” *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA1, apr 2024.
- [19] F. Long and M. Rinard, “Staged program repair with condition synthesis,” in *The 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, p. 166–178. [Online]. Available: <https://doi.org/10.1145/2786805.2786811>
- [20] M. Research, “Uninterpreted functions and constants,” 2023, <https://microsoft.github.io/z3guide/docs/logic/Uninterpreted-functions-and-constants/>.
- [21] S. Zhou, Z. Yang, D. Qiao, P. Liu, M. Yang, Z. Wang, and C. Wu, “Ferry: State-Aware symbolic execution for exploring State-Dependent program paths,” in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX, Aug. 2022, pp. 4365–4382.
- [22] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Commun. ACM*, vol. 12, no. 10, p. 576–580, oct 1969.
- [23] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, “Synthesis of loop-free programs,” in *The 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’11. New York, NY, USA: ACM, 2011, p. 62–73.
- [24] W. Weimer, Z. P. Fry, and S. Forrest, “Leveraging program equivalence for adaptive program repair: Models and first results,” in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Silicon Valley, CA, USA: IEEE/ACM, 2013, pp. 356–366.
- [25] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, “The strength of random search on automated program repair,” in *The 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, p. 254–265.
- [26] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, “Precise condition synthesis for program repair,” in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 416–426.
- [27] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, “Genprog: A generic method for automatic software repair,” *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.
- [28] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, “Sapfix: Automated end-to-end repair at scale,” in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. Montreal, QC, Canada: IEEE/ACM, 2019, pp. 269–278.
- [29] Z. Huang, D. Lie, G. Tan, and T. Jaeger, “Using safety properties to generate vulnerability patches,” in *2019 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, 2019, pp. 539–554.
- [30] S. H. Tan and A. Roychoudhury, “relifix: Automated repair of software regressions,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, vol. 1. Florence, Italy: IEEE/ACM, 2015, pp. 471–482.
- [31] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, “Improving language understanding by generative pre-training,” 2018.
- [32] OpenAI, “OpenAI Codex,” 2023, <https://openai.com/blog/openai-codex>.
- [33] —, “OpenAI ChatGPT,” 2023, <https://chat.openai.com/>.
- [34] M. R. H. Misu, C. V. Lopes, I. Ma, and J. Noble, “Towards ai-assisted synthesis of verified dafny methods,” *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, jul 2024.

10. Appendix

10.1. Code Cases

```
1 int lin_search(int x, int a[], int len)
2 {
3     int m;
4     for(m = 0; m < len; m++) {
5         if(x == a[m]){
6             return m;
7         }
8     }
9     return -1;
10 }
```

Listing 2: Linear search.

```
1 bool detect_cycle(Node* node) {
2     Node* h = node;
3     Node* t = node;
4     while (true) {
5         // fix: ...||h==NULL
6         // Plausible fix: h==NULL
7         if (h->successor == NULL)
8             return false;
9         t = t->successor;
10        h = h->successor->successor;
11        if (h == t)
12            return true;
13    }
14 }
```

Listing 3: detect cycle.

```
1 int find(int arr[], int x, int l, int r)
2 {
3     if (l == r)
4         return -1;
5     int m = l + (r - l) / 2;
6     if (x < arr[m])
7         return find(arr, x, l, m);
8     else if (x > arr[m])
9         return find(arr, x, m, r);
10    // fix: find(arr, x, m+1, r)
11    else
12        return m;
13 }
```

Listing 4: find_in_sort.

```
4 int find_first_in_sorted(int arr[], int n, int
5     x) {
6     int lo = 0;
7     int hi = n;
8     while (lo <= hi) { // while (lo < hi)
9         int mid = (lo + hi) / 2;
10        if (x == arr[mid] && (mid == 0 || x !=
11            arr[mid - 1])) {
12            return mid;
13        }
14        else if (x <= arr[mid]) {
15            hi = mid;
16        }
17        else {
18            lo = mid + 1;
19        }
20    }
21    return -1;
22 }
```

Listing 5: find_first_in_sorted.

```
9 void shortest_path_lengths(int n, double
10     length_by_edge[MAX_NODES][MAX_NODES]) {
11     for (int i = 0; i < n; i++) {
12         for (int j = 0; j < n; j++) {
13             if (i == j) {
14                 length_by_path[i][j] = 0;
15             } else if (length_by_edge[i][j] !=
16                 0) {
17                 length_by_path[i][j] =
18                     length_by_edge[i][j];
19             } else {
20                 length_by_path[i][j] = DBL_MAX;
21             }
22         }
23     }
24     double new_dist = 0.0;
25     for (int k = 0; k < n; k++) {
26         for (int i = 0; i < n; i++) {
27             for (int j = 0; j < n; j++) {
28                 new_dist = length_by_path[i][k]
29                     + length_by_path[k][j];
30                 // new_dist =
31                 length_by_path[i][k] +
32                 length_by_path[k][j];
33                 length_by_path[i][j] = (new_dist
34                     < length_by_path[i][j])
35                     ? (new_dist) :
36                     length_by_path[i][j];
37             }
38         }
39     }
40 }
```

Listing 6: shortest_path_lengths.

10.2. Path Constraint Inference Rules based on Hoare Logic

In this paper, we utilize Hoare triple and inference rules to construct the path constraint. The Hoare triple $\{P\}c\{Q\}$ formalizes the execution of a program statement, where P and Q are first-order predicates representing the pre- and post-states, respectively, and c denotes a single statement. The inference rules extend the execution formalization to the entire program, which consists of various program logic like sequential, conditional, and loop statements. The inference rules are based on the substitution rule and backward derivation. Assuming operation $c : m = m + 1$, then $\{P\}c\{Q\}$ can be converted into $[m + 1/m]Q\{m : m + 1\}Q$. This means replacing the variable $m + 1$ in state Q with m , then the state P is satisfied. On the contrary, if $\{P\}c\{Q\}$ holds true, replacing the variable m in state P with $m + 1$ will result in state Q .

In our work, we hope to retain all states on the expected path instead of replacing them since any state may be related to patches. So we convert operation c into a state $C : m_1 == m_0 + 1$, then we have an implication: $P \wedge C \Rightarrow Q$. This transformation process is equivalent to the inference rules of Hoare logic. Because the substitution rule still holds on the implication. With the inference rules, we can extend the state transitions to sequential execution, conditional execution, and cyclic execution as implications, as shown in the following.

(1) Sequence Structure:

$$\frac{\{P\}c_1\{R\} \{R\}c_2\{Q\}}{\{P\}c_1; c_2\{Q\}} \Leftrightarrow P \wedge C_1 \wedge C_2 \Rightarrow Q \quad (5)$$

(2) If-branch Structure:

$$\frac{\{P \wedge b\}c_1\{Q\} \{P \wedge \neg b\}c_2\{Q\}}{\{P\} \text{if}(b)\{c_1\} \text{else}\{c_2\}\{Q\}} \Leftrightarrow P \wedge (b \wedge C_1 \vee \neg b \wedge C_2) \Rightarrow Q \quad (6)$$

(3) While-loop Structure:

$$\frac{\{P \wedge b\}c\{P\}}{\{P\} \text{while}(b)c\{P \wedge \neg b\}} \Leftrightarrow P \wedge b \wedge C \Rightarrow P \wedge \neg b \quad (7)$$

10.3. Proof of Observation 2

Proof. Define a proposition $P(n)$: Given an execution path with n iterations, if there is a patch that fixes the fault, all states (S_n) in the n^{th} iteration is correct.

Base Case ($P(1)$): Suppose $n = 1$, the current path is equivalent to a loop-free path. Given the patch fixes the bug, S_1 should be correct. Hence, $P(1)$ is true.

Inductive Step ($P(k+1)$): Assume $P(k)$ is true for arbitrary positive integer k . This means if there is a patch that fixes the fault, then S_k is correct. Assuming that the patch has addressed the fault, we need to prove $P(k+1)$. Since S_k is correct, the input for the iteration ($k+1$) is correct as well, as it is derived from the previous iteration. Therefore, based on the correctness of S_k , the state S_{k+1} will also be correct. Thus, if $P(k)$ is true, then $P(k+1)$ is also true.

Final Step ($P(n)$): By the principle of mathematical induction, $p(n)$ is also true, as $k+1$ approaches n . \square

10.4. Prompts in LLM-enhanced workflow

Step 1: Identify invalid expected paths

Q: Looking at this buggy function `{binary_search(int x, int a[], int length)}`, We found there are 4 possible expected paths:

```
{
Expected Path 1: line 9 -> 13 -> 5 -> 6 -> 7;
Expected Path 2: line 11 -> 13 -> 5 -> 6 -> 7;
Expected Path 3: line 9 -> 13 -> 14;
Expected Path 4: line 11 -> 13 -> 14.
}
```

Note that the expected path is sliced; it only includes the last round of iterations. Please help me identify which path is invalid to fix the program.

A:

Analysis:

Expected Path 1: line 9 -> 13 -> 5 -> 6 -> 7

- Line 9 sets $L = m + 1$, which narrows the search to the upper half.
- In the next iteration, L would potentially match m such that $x == a[m]$, so the correct output 2 (index of $a[2]$) would be returned.
- This path seems **possible**.

Expected Path 2: line 11 -> 13 -> 5 -> 6 -> 7

- Line 11 sets $R = m - 1$, which narrows the search to the lower half.
- However, since $x == a[2]$, reducing R (i.e., moving to the lower half) would prevent the algorithm from finding $a[2]$.
- Thus, this path would not result in the correct output.
- This path is **invalid** for this input.

.....

Conclusion:

Valid Paths: Expected Path 1, Expected Path 4.
Invalid Paths: Expected Path 2, Expected Path 3.

Figure 5. LLM-enhanced Workflow: Step 1. Identify invalid expected paths.

Step 2: Summarize path constraint

Q: Looking at these expected paths. Combing with the input condition of fault paths:

```
{
Fault Path 1. Input condition: [a[0]<a[1]<a[2], x == a[2]],
Fault Path 2. Input condition: [a[0]<a[1]<a[2], x<a[0]].
}
```

Please help me summarize the constraint about the patch on the path. The patch can be expressed as β . Write each path as a constraint that can be solved to infer the expression of β

A:

Expected Path 1: line 9 \rightarrow β \rightarrow line 5 \rightarrow line 6 \rightarrow line 7

Constraints:

- Initial state:
 - $L_0 = 0;$
 - $R_0 = \text{length} - 1;$
 - $m_0 = (L_0 + R_0) / 2;$
 - $x \neq a[m_0].$
- After first update (line 9):
 - $L_1 = m_0 + 1.$
- Apply patch β :
 - $\beta = \text{true};$
 - $m_1 = (L_1 + R_0) / 2.$
- Correct result condition:
 - $x = a[m_1];$
 - Return the index m_1

Figure 6. LLM-enhanced Workflow: Step 1. Identify invalid expected paths.

Step 3: Generate the patch

Q: Looking at these expected path constraints, please generate an expression of β satisfying all path constraints.

A:

- For Expected Path 1 (where x is found):
 - $L_1 = m_0 + 1$
 - $m_1 = \frac{L_1 + R_0}{2}$
 - $x = a[m_1]$
 - The patch β must allow this progression and result in the correct index being returned. Thus, β should allow the algorithm to continue searching and re-compute m .
- Expected Path 4 (where x is not found and the search exits):
 - Path 4: $R_1 = m_0 - 1$ and $L_0 > R_1$
 - The patch β should allow the algorithm to terminate when the exit condition $L > R$ is met, leading to the return of -1 .

General Logic for

The patch β should ensure that the binary search continues until either:

- $x = a[m]$, and the correct index is found (Expected Path 1) or
- $L > R$, and the search exits (Expected Path 4).

Thus, the expression for β should be based on the following conditions:

- β should allow iteration to continue if $L \leq R$ (i.e., the search space is not exhausted).
- β should allow termination when $L > R$ (i.e., the search space is exhausted).

Final Expression for

Given these considerations, β can be expressed as the following logical condition:

$\beta = (L \leq R)$

Figure 7. LLM-enhanced Workflow: Step 1. Identify invalid expected paths.