Agentic Concolic Execution

Zhengxiong Luo*, Huan Zhao*, Dylan Wolff*, Cristian Cadar†, Abhik Roychoudhury*
* *National University of Singapore*
† *Imperial College London*
luozx@nus.edu.sg, {zhaohuan, wolffd}@u.nus.edu, c.cadar@imperial.ac.uk, abhik@nus.edu.sg

***Abstract*—Concolic execution is a practical test generation technique that explores execution paths by coupling concrete execution with symbolic reasoning. It runs programs on given inputs while capturing symbolic path representations, then mutates and solves these constraints to generate new test inputs for alternative paths. This approach has several fundamental challenges, such as (C1) the inherent complexity of symbolically modeling diverse programming language constructs and environmental interactions, and (C2) the scalability issues of constraint solvers when handling large, complex formulas.**

In this work, we investigate whether LLM agents can help address these longstanding challenges in test generation. We propose a novel workflow which we call *agentic concolic execution*. Using an LLM agent for symbolization, our approach is language-agnostic and can handle environmental constraints without additional manual modeling effort. To ease pressure on the constraint solver, we allow an LLM agent to summarize and even reason about constraints directly in natural language. In a significant evaluation of 12 real-world subjects, our research prototype CONCOLLMIC attains significantly higher code coverage (115%-233% higher) than state-of-the-art symbolic executors like KLEE that have been painstakingly hand-crafted over many years, and identifies 11 new vulnerabilities. Our results show that multi-step planning and tool integration enable agents to effectively mitigate reliability issues inherent in LLM-based analysis and even reason *symbolically* about code.

1. Introduction

While security vulnerabilities have long plagued software systems, modern software stacks have grown increasingly complex and heterogeneous. It is not uncommon for codebases to exceed thousands of lines of code, include multiple programming languages, and involve complex interactions with external services and the broader environment.

Dynamic Symbolic Execution (DSE) is a widely-used program analysis technique designed to systematically explore execution paths [1]. It treats program inputs as symbolic variables, collects symbolic constraints along different control-flow paths, and solves these constraints to generate concrete inputs that exercise corresponding program behaviors. *Concolic execution*, a variant of DSE, couples concrete and symbolic execution: it starts with a concrete input, executes the program both concretely and symbolically to gather symbolic constraints representing the executed path, then negates selected constraints and solves for new inputs to explore alternative paths [2]. These approaches have achieved success in discovering vulnerabilities in real-world software [3], played key roles in competitions like DARPA Cyber Grand Challenges [4], [5], and proven effective in both FLOSS and commercial software [2], [6], [7].

Despite their success, these approaches still face two fundamental challenges when applied to modern software:

***C1**: Complex and Incomplete Implementations.* Modern DSE engines often integrate a symbolic interpreter with a virtual-machine-style architecture [3]. This approach enables them to collect symbolic constraints arising from diverse programming language constructs and environmental interactions, such as file and network operations. However, developing comprehensive symbolic modeling rules for all such behaviors leads to highly complex implementations. As a result, DSE engines typically lack full support for complex languages such as C and C++, offer no support for multi-language codebases, and provide only partial modeling of environmental interactions [3], [8], significantly hindering their practicality.

***C2**: Expensive Constraint Solving.* Even when a codebase is fully supported, the sheer scale of modern software presents challenges for constraint solving. Accurately representing a full execution path in a program often results in formulas with thousands of variables and conditions, an issue exacerbated by the verbose representation used by DSE engines, where, e.g., constraints for integers are typically represented as bitvectors [9], and arrays are modeled at the individual element level, rather than as aggregate structures [10], [11]. Moreover, many commonly used constraints, like those involving floating point or strings, cannot be solved quickly by existing approaches, even when the formulas involved are of only moderate size [12], [13].

Recently, Large Language Model (LLM) agents have emerged as a transformative technology across various disciplines of computer science and beyond. In particular, LLMs have demonstrated remarkable capabilities in general mathematical reasoning [14], and specifically in reasoning about real-world software artifacts [15]. LLM agents have led to substantial advances in many software engineering tasks such as program repair [16] and test execution [17]. One key advantage of such agents is their ability to invoke analysis tools autonomously to enhance the power of LLMs.

This raises the question: *Can LLM agents be leveraged for **symbolic reasoning** over complex software systems?*

Despite their remarkable capabilities, LLMs appear unsuitable for symbolic analysis at first glance. Firstly, LLMs offer no formal or probabilistic guarantees of correctness and often produce incorrect answers to queries. If components of a DSE engine are delegated to an LLM, these erroneous answers could easily lead to poor exploration of the actual program behaviors. Secondly, current state-of-the-art foundation models also typically have high latencies, taking time on the order of seconds to respond to a single query. With solver latency already being a primary bottleneck in traditional DSE approaches [9], [18], [19], it is unclear whether incorporating LLMs will be beneficial even if they are able to answer queries accurately.

In this work, we investigate whether LLM agents can perform symbolic reasoning. If so, this could dramatically simplify the design of new concolic executors for systems involving programming languages, constraint types, and environmental interactions poorly supported by existing engines, or even complex multi-lingual systems. Given that state-of-the-art DSE engines take years to construct [20], such an approach could have a substantial practical impact.

To combat the inherent untrustworthiness of LLMs, we instantiate agentic symbolic reasoning via concolic execution. The resulting agentic system, CONCOLLMIC, is the first *language-agnostic* concolic executor. Our approach consists of a collection of LLM agents and modules, each of which benefits from the rich problem-solving capabilities of backend LLMs. We design an instrumentation module that instruments the source code at strategic locations, enabling us to abstract the execution traces into a uniform representation at runtime. This abstraction is then symbolized by a summarization agent, whose symbolization focuses on program semantics by permitting one of many possible constraint representations, expressed in natural language, source-level syntax, or formal notations like SMT formulas, as needed. These constraints are then solved by a solving agent to generate new test inputs. To mitigate the inherent untrustworthiness of LLMs and encourage multi-step reasoning, these agents are each equipped with several grounded software tools, such as code retrieval and a state-of-the-art SMT solver [21]. As such, we deem our approach *agentic concolic execution*.

In a comprehensive evaluation across eight real-world C/C++ programs, four multi-lingual systems, and a specialized floating-point benchmark, we find that agentic concolic execution is remarkably effective. CONCOLLMIC is able to cover application domains unsupported by existing engines, attain 115%-233% higher code coverage than conventional DSE tools as well as 81% higher coverage than AFL++, and find 11 previously unknown bugs.

In summary, we make the following contributions:

1) We formulate a new paradigm of symbolic analysis using agentic reasoning to address the core challenges in DSE: **(C1)** symbolic modeling and **(C2)** constraint solving.
2) We build the first ever *language-* and *theory-agnostic* concolic execution engine, CONCOLLMIC, which is highly effective in automated test generation. To foster further research in this area, we have also made CONCOLLMIC publicly available as open source at https://github.com/ConcoLLMic/ConcoLLMic.
3) We conduct an empirical study to demonstrate that agentic access to multi-step planning and tools mitigates reliability concerns with LLMs in practice, and enables them to reason about real-world software *symbolically*.

2. Background and Motivation

Dynamic Symbolic Execution (DSE). As a variant of symbolic execution [22], [23], [24], DSE is one of the primary techniques to uncover software defects. To do so, DSE engines produce inputs to traverse each control-flow path in the program under test, by executing the program with *symbolic* (i.e., initially unconstrained) inputs. During execution, values of variables in the program are represented by expressions over these symbolic inputs. At each branch in the program, the symbolic executor accumulates the constraints necessary to follow the desired path in terms of these program variables. To materialize a *concrete* set of inputs which follow the path, the executor solves the accumulated constraints, typically by delegating to a dedicated constraint solver. Inputs provided by the solver that satisfy these constraints will necessarily follow the intended program path.

Gathering symbolic constraints is a core component of DSE, which translates behaviors from the concrete semantics of the program into logical constraints over symbolic inputs. For some behaviors—such as arithmetic operations over integers—this translation is direct and mechanical. In other cases, the translation process can be significantly more complicated [25]. Gathering these constraints more efficiently has been the subject of recent research [8], [26], often at the cost of increased implementation complexity **(C1)**. Furthermore, the program under test will typically interact with its *environment*, including the file system, hardware devices, networks, and external library calls, among many others. Often, the semantics of these interactions are either unavailable (e.g., due to proprietary implementations), too complex to model symbolically with precision, or require significant efforts from the developers [3], [27].

Constraint solving is also a core component of DSE. The logical constraints, if gathered successfully, are typically encoded in the form of Satisfiability Modulo Theories (SMT) formulas [28]. SMT formulas are first-order logical formulas which generalize Boolean Satisfiability (SAT) to support *theories* like arrays, arithmetic integers, and bitvectors. Indeed, the rise of practical SMT solvers, including Z3 [21], STP [29], and CVC4 [30], was instrumental in the success of symbolic execution engines [3], [6], [7], [10], enabling them to solve complex constraints at the scale of some real-world programs. However, given the complexity of these constraints and tight latency requirements in solving times [19], constraint solving remains a significant bottleneck **(C2)**. As a result, modern engines will often drop

Algorithm 1: Conventional Concolic Execution

Input : Program P, initial concrete input I_0

1 $\tilde{P} \leftarrow$ INSTRUMENT(P)
2 *WorkList* $\leftarrow \{I_0\}$
3 **while** *WorkList* $\neq \emptyset$ **do**
4 $\quad I \leftarrow$ SELECTNEXT(*WorkList*)
5 $\quad (O, T) \leftarrow$ EXECUTE($\tilde{P}$, I)
6 $\quad (PC, \sigma) \leftarrow$ SYMBOLIZE(T)
7 $\quad PC' \leftarrow$ FLIPCONSTRAINT(PC)
8 $\quad I' \leftarrow$ SOLVE(PC', σ)
9 $\quad$ *WorkList* $\leftarrow$ *WorkList* $\cup$ I'

constraints [26] or use incomplete solvers [18] to increase throughput at the cost of precision.

Concolic Execution. Concolic execution engines [2], [6], [7], [8] couple symbolic execution with concrete execution by executing the program both concretely and symbolically. Algorithm 1 shows the general procedure for concolic execution. First, the program is augmented to emit execution information at runtime (line 1), so that execution paths can be tracked. This augmentation is often an instrumentation pass over the original program [8]. Next, the augmented program $\tilde{P}$ is executed concretely to collect its output O and execution trace T (line 5), which will then be symbolized as path constraints (PC) and a symbolic store (σ) that maps variables to symbolic expressions over inputs (line 6). In modern concolic execution engines, lines 5-6 of the algorithm are often performed in lockstep, accumulating PC as program $\tilde{P}$ is being executed. To generate a new input which explores a different path, the executor chooses a prefix of PC and negates the last constraint in that prefix (line 7). The resulting path constraints PC' are passed to a constraint solver with the symbolic store σ (line 8), giving a new input I' which is added to the worklist (line 9).

2.1. Motivating Example

To illustrate the challenges faced by traditional concolic execution, consider the program in Figure 1 from FP-Bench [12]. This program takes two command-line string arguments, converts them to 32-bit floats via `atof`, and counts the representable floating-point numbers between them via function `count`. If this count does not exceed `FLOATS_BETWEEN_BUG` (i.e., 20), the program triggers a bug (line 16). Specifically, `count` function's loop body (lines 7–10) implements floating-point increment by treating the float's bit representation as an unsigned integer (i.e. type casting): it copies the float to integer `temp` via `memcpy` (line 8), increments the integer representation (line 9), and copies back (line 10) to get the next representable floating-point value [12].

Conventional Concolic Execution. Consider testing this code with an initial input I: $I[0]$="1.00", $I[1]$="1.00001". There are 84 representable floats in $[1.00, 1.00001)$. Hence, I's execution iterates through the for-loop 84 times and fails

```
1  #define FLOATS_BETWEEN_BUG 20
2  int cnt = 0;
3
4  int count(float start, float end) {
5   for (float cur = start; cur != end; cnt++) {
6    fprintf(stderr, "[src/count.c] enter count 1");
7    unsigned temp;
8    memcpy(&temp, &cur, sizeof(float));
9    temp++;
10   memcpy(&cur, &temp, sizeof(float));
11   // fprintf(stderr, "[src/count.c] exit count 1");
12  }
13
14  if (cnt <= FLOATS_BETWEEN_BUG) {
15   fprintf(stderr, "[src/count.c] enter count 2");
16   printf("BUG triggered!");
17   return -1;
18   // fprintf(stderr, "[src/count.c] exit count 2");
19  }
20 }
21
22 int main(int argc, char **argv) {
23  fprintf(stderr, "[src/count.c] enter main 1");
24  float start = atof(argv[1]), end = atof(argv[2]);
25  __assume__(start < end);
26  count(start, end);
27  // fprintf(stderr, "[src/count.c] exit main 1");
28 }
```

Figure 1: Motivating example from FP-Bench [12], with additional instrumentation statements inserted.

the condition at line 14. During concolic execution, a symbolic store σ: {program variable $\mapsto$ *symbolic expression*} tracks the symbolic states of variables. After i iterations, the symbolic store takes the form:

$$\sigma_i = \underbrace{\{\text{start} \mapsto \alpha,\ \text{end} \mapsto \beta\}}_{\text{initial state } \sigma_\circ} \cup \underbrace{\{\text{cnt} \mapsto i,\ \text{cur} \mapsto cur_i = f^i(\alpha)\}}_{\text{loop iteration } i}$$

Here, f represents a symbolic function capturing the loop body logic, including complex operations such as pointer manipulation, memory operations via `memcpy`, and type casting. $f^i(\alpha)$ denotes applying f repeatedly i times (i.e., $f^{i+1}(\alpha) = f(f^i(\alpha))$). Similarly, the symbolic path constraints PC accompanying I's execution take the form:

$$PC:\ \alpha = \texttt{atof}(I[0]) \wedge\ \beta = \texttt{atof}(I[1]) \\ \wedge\ \bigwedge_{i=0}^{n-1} (cur_i \neq \beta) \wedge\ cur_n = \beta \wedge\ cnt_n > 20$$

where $n = \min\{k \geq 0 : f^k(\alpha) = \beta\}$ denotes iterations needed from α to β via f. Concolic execution then negates the last clause in a prefix of PC to generate PC', and passes these constraints to a solver to generate the next input I'.

Problems. The above symbolic representation reveals several fundamental challenges:

(1) External functions. The constraints PC contain external functions like `atof` and `memcpy` (encoded in f), which are not part of the source code but are essential for symbolization **(C1)**. Handling these functions requires either: (i) making their implementations available, which substantially bloats code size while introducing many low-level implementation details that obscure high-level semantics; (ii) manually modeling them, which is expensive and

error-prone; or (iii) concretizing their arguments, which sacrifices completeness and can potentially render the formula unsolvable—as indeed occurs in our example.

(2) Verbose and complex formulas requiring specialized theory support. The constraints are already extremely verbose, containing hundreds of clauses and growing proportionally to loop iterations—making them much larger than the original program in Figure 1. Since floating-point numbers are encoded as bitvectors in SMT, even a single clause in PC expands into a collection of 32 bit-level constraints, further overwhelming the constraint solver **(C2)**. Moreover, state-of-the-art engines like KLEE [3] lack adequate floating-point support for such constraints. Finding the bug at line 16 requires a *specialized* KLEE version [12], which took over 10 months to develop (as per the authors) in addition to years of work on the original KLEE. Despite this substantial effort, it *still* fails for higher precision numbers such as 16-byte doubles due to incomplete theory support.

Our Approach. CONCOLLMIC circumvents these challenges by operating at higher semantic abstraction levels. For this example, instead of generating verbose implementation-level formulas that mechanically mirror execution flow, CONCOLLMIC distills the program's core intent into natural language (NL) constraints: *"find two floating-point numbers with fewer than 20 representable numbers between them"*. This representation directly captures program semantics, leveraging LLMs' reasoning capabilities and general knowledge of theories and library functions. Then, an autonomous solving agent bridges these high-level semantic constraints with precise computational tools. Instead of simply translating high-level constraints into traditional solver formats, this agent solves constraints by understanding their semantics, decomposing problems into sub-tasks, and invoking tools on demand to obtain grounded feedback that iteratively refines its reasoning. As a result, this agentic workflow enables CONCOLLMIC to efficiently explore complex paths that are intractable for traditional methods.

3. System Design

In this section, we first introduce the high-level workflow of our agentic framework for concolic testing, highlighting its key differences from conventional counterparts. Then, we elaborate on each component in detail.

3.1. System Overview

Workflow. Figure 2 depicts CONCOLLMIC's workflow, which resembles Algorithm 1 at a high level, featuring instrumentation and testing modules. However, in our approach, each of these major components is instantiated with LLM support (shaded in gray), rather than static algorithms.

(1) *Instrumentation Stage.* First, CONCOLLMIC instruments the target program to support tracking concrete execution flow. Unlike traditional approaches relying on language-specific parsers or runtime instrumentation, CONCOLLMIC

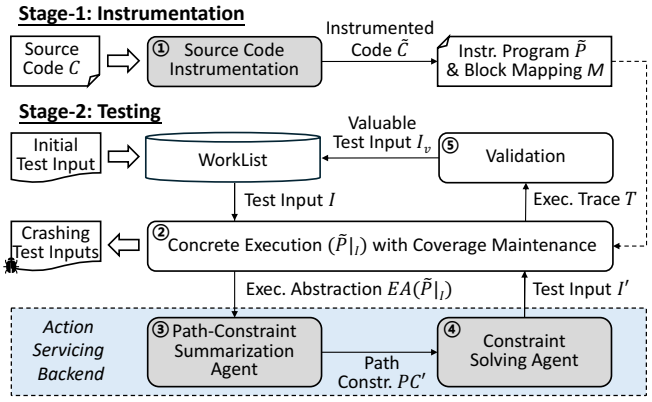


Figure 2: CONCOLLMIC workflow: *Stage-1*: ① source-code level instrumentation for execution tracing, and *Stage-2*: iterative testing which, for the input I to explore, ② traces its execution $EA(\tilde{P}|_I)$, ③ distills new path constraints PC' targeting an alternative path, ④ produces a satisfying input I' and ⑤ validates it. Modules in gray are powered by LLMs, with two agents supported by their specialized toolkits.

operates directly at the *source code level* (①). To support diverse programming languages, CONCOLLMIC employs instrumentation to make the program output uniform textual logs across different language runtimes. Figure 1 illustrates the instrumented code. Specifically, CONCOLLMIC adopts a block-based approach that identifies code units naturally bounded by control-flow constructs (e.g., conditional branches and loops) and instruments their boundaries via *flow-tracing statements*: pairs of logging statements (e.g., lines 6 and 11) that wrap each logical code unit (e.g., lines 7-10) with unique identifiers. We refer to the wrapped code units as *source blocks*. An LLM is employed to insert these flow-tracing statements conforming to each language's syntax, leveraging its multilingual understanding for this pattern recognition task. This design is deliberate: (i) the structured log format captures sufficient execution information for downstream reconstruction (ii) the instrumentation pattern maintains simplicity for reliable LLM-based insertion across diverse programming paradigms and (iii) the paired pattern coupled with block identifiers enables us to perform language-agnostic offline validation for the LLM's output, enhancing the overall reliability of this module.

This instrumentation process produces two artifacts enabling the subsequent execution tracing: (i) an instrumented program $\tilde{P}$ that emits execution traces at runtime, and (ii) a mapping $\mathcal{M}$ from source block identifiers to corresponding source lines (e.g., Table 1). These two artifacts enable tracking each execution trace and maintaining an internal line coverage bookkeeping for subsequent testing. Crucially, once instrumented, subsequent execution tracing becomes language-agnostic and can be performed without requiring per-execution LLM queries or language-specific parsers.

(2) *Testing Stage.* Then, CONCOLLMIC initiates iterative concolic exploration with an initial test input I_0 and maintains a *WorkList* of test inputs. In each iteration, CON-

TABLE 1: Mapping from source block identifiers to source lines for the instrumented code in Figure 1.

⟨filepath, blockID⟩	Source Lines
⟨src/count.c, (main, 1)⟩	src/count.c: 24-26
⟨src/count.c, (count, 1)⟩	src/count.c: 7-10
⟨src/count.c, (count, 2)⟩	src/count.c: 16-17

CoLLMic selects one input I from the list for exploration. Building upon the instrumented module, ConcoLLMic (②) can capture the concrete execution $\tilde{P}|_I$ and compress it into a concise *execution abstraction* $EA(\tilde{P}|_I)$. This abstraction preserves only essential information, including function call chains and *executed files (files containing code that was executed)*, where unexecuted source blocks are replaced with comments indicating their global coverage status. Figure 3 shows an illustrative example. *Removing unexecuted* blocks instead of *retaining executed* ones offers two advantages: (i) the condition statements remain visible, and (ii) the global definitions and variable declarations stay accessible. These elements provide essential supplementary information for understanding the execution flow.

The execution abstraction $EA(\tilde{P}|_I)$ is then forwarded to the Path-Constraint Summarization Agent (③). This is an autonomous agent tasked with selecting a branch b (along the execution trace) to flip and distilling symbolized constraints PC' to materialize this branch flip. The program $\tilde{P}$ is expected to exercise the *alternative branch* of b should PC' be satisfied, and the Constraint Solving Agent (④) is responsible for generating a test input I' that satisfies these constraints. Both agents are able to invoke various tools supported by the *action serving backend*, including code retrieval and the state-of-the-art constraint solver Z3 [21].

The generated test input I' is then executed to test the program and undergoes validation (⑤): if the execution trace indicates a successful branch flip or new code coverage, I' is added to *WorkList* for future exploration. ConcoLLMic repeats this process until the time budget is exhausted.

Key Differences from Conventional Concolic Execution Tools. While ConcoLLMic adheres to the overall workflow outlined in Algorithm 1, it diverges from it in two important aspects:

1) First, conventional concolic executors bundle the concrete and symbolic execution (line 5-6), whereas ConcoLLMic performs symbolization *post factum* on execution abstractions. This essentially decouples the symbolization from any specific language and enables a multilingual workflow via the instrumentation module's unified interface (**C1**).
2) Second, a primary goal of ConcoLLMic is to express constraints at arbitrary abstraction levels, harnessing LLM-driven semantic understanding (**C2**). To facilitate this representation, ConcoLLMic chooses a target branch to flip (line 7) *before* symbolization (line 6). It does so because the fully abstracted constraints (e.g., in NL) may no longer capture implementation-level branch details (e.g., `for` and `if` details in lines 5 and 14 in Figure 1). Reversing the order enables the LLM to directly synthesize high-level constraints that match the implementation-level target branch.

```
(1) Function Call Chain:
[src/count.c] main → [src/count.c] count

(2) Executed Files with Deleted Unexecuted Blocks and Their Cov Data:
// src/count.c (28 lines total)
#define FLOATS_BETWEEN_BUG 20
int cnt = 0;

int count(float start, float end) {
  for (float cur = start; cur != end; cnt++) {
    unsigned temp;
    memcpy(&temp, &cur, sizeof(float));
    temp++;
    memcpy(&cur, &temp, sizeof(float));
  }
  if (cnt <= FLOATS_BETWEEN_BUG) {
    // Unexecuted lines 16-17 removed. Line cov: 0/2
  }
}
int main(int argc, char **argv) {
  float start = atof(argv[1]), end = atof(argv[2]);
  __assume__(start < end);
  count(start, end);
}
```

Figure 3: Example of an *execution abstraction* when using (1.00, 1.00001) as input for the program in Figure 1.

3.2. Stage-1: Instrumentation

To address **C1**, ConcoLLMic treats different programming languages uniformly by leveraging the LLM's multilingual capabilities. At a high level, it abstracts the program as a collection of source lines and inserts appropriate flow-tracing statements. These flow-tracing statements can help track execution at runtime, enabling downstream agents to comprehend the program's execution flow.

Algorithm. This process is outlined in Algorithm 2, with LLM-powered procedures highlighted in gray boxes. Given the source code $\mathcal{S}$ of program P in any language, the module produces two artifacts: (1) instrumented source code $\tilde{\mathcal{S}}$ containing logging statements, which is subsequently compiled into $\tilde{P}$, and (2) a mapping $\mathcal{M}$ that translates a source block to its corresponding source code lines.

The instrumentation module processes each file $F \in \mathcal{S}$ independently (lines 2-13). Each file F is further split into manageable chunks not exceeding *ChunkSize* (default 800 lines) to accommodate the output token limit of the LLM (line 4). The chunks are aligned with function boundaries recognized by the LLM to ensure syntactic integrity. For each code chunk C_i, the instrumentation agent prompts an LLM to insert logging statements at strategic points in the code to track execution flow (line 9). The instrumentation follows this pattern:

```
enter func_name block_no
original_code
// exit func_name block_no
```

This instrumentation wraps sequences of executable statements with a pair of flow-tracing statements carrying a *block ID*—the tuple of `(func_name, block_no)`. We refer to

Algorithm 2: Source Code-Level Instrumentation

Input : $\mathcal{S}$ - Source code of program P
Output : $\tilde{\mathcal{S}}$ - Instrumented source code of $\tilde{P}$
$\mathcal{M}$ - Mapping from block IDs to code lines

1 $\tilde{\mathcal{S}} \leftarrow \emptyset$, $\mathcal{M} \leftarrow \emptyset$
2 **foreach** F *in* $\mathcal{S}$ **do**
3 $\tilde{F} \leftarrow []$
4 *chunks* $\leftarrow$ SPLIT(F, *ChunkSize*)
5 **foreach** C_i *in chunks* **do**
6 $\tilde{C}_i \leftarrow C_i$
7 *feedback* $\leftarrow$ Null, *isValid* $\leftarrow$ False
8 **while** $\neg$ *isValid* **do**
9 $\tilde{C}_i \leftarrow$ INSTRUMENT($\tilde{C}_i$, *feedback*)
10 *isValid, feedback* $\leftarrow$ VALIDATE($\tilde{C}_i$)
11 $\tilde{F}$.append($\tilde{C}_i$)
12 $\tilde{\mathcal{S}} \leftarrow \tilde{\mathcal{S}} \cup$ POSTPROCESS($\tilde{F}$, FILEPATH(F))
13 $\mathcal{M} \leftarrow \mathcal{M} \cup$ GETMAPPING($\tilde{F}$)
14 **return** $\tilde{\mathcal{S}}, \mathcal{M}$

each such wrapped sequence of executable statements as a *source block*. These flow-tracing statements naturally delineate control-flow constructs such as conditional branches (if-else, switch-case) and loops (for, while). Notably, the "exit" statements are commented out and serve only as markers for block boundaries in the source file. In particular, we use them as sanity checks for the correctness of the LLM-based instrumentation, as explained next.

The validation check (line 10) verifies the structural integrity of the instrumentation. Specifically, it checks that each "enter" logging statement must pair with *exactly one* "exit", and these pairs must nest consistently—akin to parenthesis matching. If the validation fails, the module emits diagnostic feedback detailing the relevant *block ID* and the line number of any unclosed or mismatched entry–exit pair. While this is not a guarantee of correctness, this feedback enables iterative refinement of the instrumented chunk (lines 8–10).

After all chunks in C have been instrumented and validated, a POSTPROCESS pass (line 12) combines them by appending the file path of F to the logging statements. At the end, each code block is indexed by a globally unique tuple ⟨*filepath*, *block ID*⟩, and is mapped to a unique range of source lines. This mapping is stored in $\mathcal{M}$ (line 13).

As an example, Figure 1 shows the instrumented code with inserted flow-tracing statements colored, and Table 1 shows the mapping of source blocks to source lines. By matching the output sequence of flow-tracing statements, CONCOLLMIC is able to reconstruct: (a) the function call chain, and (b) the set of executed source blocks, which are used to construct the execution abstraction and maintain global line coverage records. Note that the function call chain may be approximated in certain edge cases. Take Figure 1 as an example: assume that `cnt` is initialized with a value greater than `FLOATS_BETWEEN_BUG` at line 2, and `count(1,1)` is invoked. In this case, no instrumentation logs

Algorithm 3: Autonomous Workflow for the Constraint Summarization and Solving Agent

Input : $\mathcal{I}$ - Agent's task-specific input and instruction
$\mathcal{S}_{action}$ - Agent's action space
Output : $\mathcal{O}$ - Agent's output

$H \leftarrow [\mathcal{I}]$
action $\leftarrow$ Null
while $\neg$ (ISFINISHACTION(*action*) *or timeout*) **do**
  action $\leftarrow$ CHOOSEACTION(H, $\mathcal{S}_{action}$)
  result $\leftarrow$ PROCESSACTION(H, *action*)
  H.append(*result*)
$O \leftarrow$ EXTRACTRESULT(H)
return O

in `count` would be output, resulting in `count` being omitted from the call chain. Nevertheless, the Summarization Agent can still leverage the CODEREQUEST action to request the corresponding code for supplementing the context.

Benefits. In summary, the advantages of our instrumentation approach are threefold:

1) **Low manual effort**: Unlike traditional methods that depend on language-specific parsers or runtime-specific instrumentation, our approach can process the target program in a fully language-agnostic pipeline by leveraging LLMs' deep multilingual code understanding. This means that our approach is applicable on a wealth of popular programming languages with *zero* additional configuration.
2) **Cross-language tracing**: Perhaps more importantly, the instrumentation allows concrete executions to be analyzed in a uniform and language-agnostic manner. Our block-to-line mapping representation enables the analysis of the execution flow of complex *polyglot* systems that cross language boundaries.
3) **Modularity**: Our design also retains the modularity of conventional methods, thanks to independent per-chunk instrumentation. As a result, CONCOLLMIC can *incrementally* re-instrument the target program on subsequent modifications, adding logging only to the functions that have been changed. This incremental instrumentation can significantly reduce costs during the continuous development of large software projects. To demonstrate the feasibility and effectiveness of our approach to instrumentation, we report the cost and accuracy in §4.4.1.

3.3. Stage-2: Testing

The key components in the testing stage are the Path-Constraint Summarization Agent and the Constraint Solving Agent (Summarization Agent and Solving Agent for short). The Summarization Agent produces self-contained path constraints PC' that serve as the interface between the two; the Solving Agent then synthesizes a test input I' that satisfies all specified constraints. Unlike conventional symbolic executors that submit fixed-format queries to a static solver, our framework supports arbitrary abstraction levels (**C2**):

TABLE 2: Action space of the Summarization Agent, with the finishing action marked with *.

Tool Name	Arguments	Description
THINK	`reasoning`	Record reasoning and planning.
CODEREQUEST	A list of `file:[line_range]`	View the source lines and coverage status in `file`, optionally within specified `line_range`.
CHOOSEBRANCH	`target_branch`, `rationale`, `lines_to_cover`	Select a `target_branch` to flip and the expected `lines_to_cover` after its flipping.
SUMMARIZE	`path_constraint`	Generate `path_constraint` (in NL/PL/SMT) required to reach `target_branch`.
FINISH*	`task_completed`	Stop exploring the current test case.

TABLE 3: Action space of the Solving Agent, with the finishing action marked with *.

Tool Name	Arguments	Description
THINK	`reasoning`	Record reasoning and planning.
EXECUTECODE	`code`	Execute the given Python code and collect its output.
QUERYSMT	`SMT_formulas`	Solve SMT formulas using Z3.
GENERATETEST*	`is_satisfiable`, `exec_program`	Generate a test input `exec_program` (in *harness.py*) if the given constraint `is_satisfiable`.

constraints may be expressed as SMT formulas, programming language (PL), or even in natural language (NL), as deemed appropriate by the LLM agent.

Autonomous Agents. Algorithm 3 illustrates the shared autonomous workflow of both agents, with the procedure involving the LLM highlighted in a gray box. The workflow begins by initializing the action history H with the instructions and the initial input $\mathcal{I}$ (line 1). Then, the agent autonomously chooses its actions from its action space $\mathcal{S}_{action}$ based on the current history H (lines 3-6). These actions are supported by a toolkit of offline utilities listed in Table 2 and 3. Each action request is serviced locally (line 5), and its result is appended to the history H (line 6) for the next iteration. This iterative process continues until a finishing action or a timeout is reached (line 3), after which the output is extracted from the cumulative history H (line 7). With the provided toolkit, this workflow enables the LLM to (1) record *internal* planning and reasoning, and (2) interface with *external* sources (e.g., browsing or executing code) on demand, to carry out multi-step reasoning for the given task. We illustrate this agentic workflow in Figure 4 using our running example. Below we present the Summarization and Solving Agent in more detail.

Constraint Summarization. The Summarization Agent takes in as *inputs* the concrete test input I and its *execution abstraction* $EA(\tilde{P}|_I)$, which succinctly represents the concrete execution on I with essential information, including function call chains and the source lines essential for understanding the execution flow, as illustrated in Figure 3. This execution abstraction is derived by processing the emitted trace T (i.e. the output sequence of flow-tracing statements) using the instrumentation pattern and the mapping $\mathcal{M}$.

The *action space* of the Summarization Agent is detailed in the upper half of Table 2. It can plan its actions and reason about the code via THINK or browse code snippets that are not present in $EA(\tilde{P}|_I)$ by invoking CODEREQUEST. Based on this information, the agent's main task is to first select a `target_branch` through CHOOSEBRANCH, and eventually distill the symbolic `path_constraint` via SUMMARIZE. This process could be repeated to select multiple target branches before a voluntary FINISH or a timeout is reached. Note that we deviate from the conventional Algorithm 1 by selecting multiple (instead of one) target branches per iteration to amortize the token cost of *execution abstraction*.

The *outcome* is a list of triples denoted (`target_branch`, `path_constraint`, `lines_to_cover`). The `target_branch` is a condition along the concrete execution trace of $\tilde{P}|_I$ but with a flipped branch. This flipping is expected to introduce new code coverage at `lines_to_cover`. The `path_constraint` captures the symbolic constraints required to reach these target lines. We highlight that the generated constraints encapsulate both *input* and *environmental* constraints and are expressed in an appropriate notation, e.g., natural language (NL), programming language (PL), or formal SMT formulas. Moreover, the path constraint summarization implicitly leverages the LLM's knowledge of program semantics, symbolization rules, and common environmental interactions (e.g., system calls, `libc` built-in functions).

Constraint Solving. The Solving Agent takes as *inputs* the distilled constraints from the Summarization Agent, and the base concrete test input I as a reference. Its aim is to synthesize a satisfying test input I' for the specified constraints. Similar to the Summarization Agent, it can invoke multiple tools from its *action space* to achieve this goal (as shown in Table 3). These include planning and reasoning via THINK, submitting SMT queries to a trusted backend solver with QUERYSMT, and executing Python scripts it has written through EXECUTECODE. Collectively, these tools enable an analyze-hypothesize-validate workflow to reason precisely across a wide range of theories. As a final output artifact for a satisfiable constraint, the Solving Agent produces a Python "harness" via GENERATETEST. This harness executes the target program with all necessary inputs and the correct environment settings to flip the targeted branch. The `execute_program` function is extracted as the final *outcome* of the Solving Agent.

Workflow Example. Figure 4 illustrates the autonomous workflow of the two agents on our motivating example in Figure 1. Based on the execution abstraction shown in Figure 3, the Summarization Agent first analyzes uncovered branches and chooses to cover the `True` branch of the `if` condition. Then, it formulates path constraints expressed in natural language, capturing program requirements, variable relationships, and high-level semantics. The Solving Agent then takes these constraints and independently orchestrates

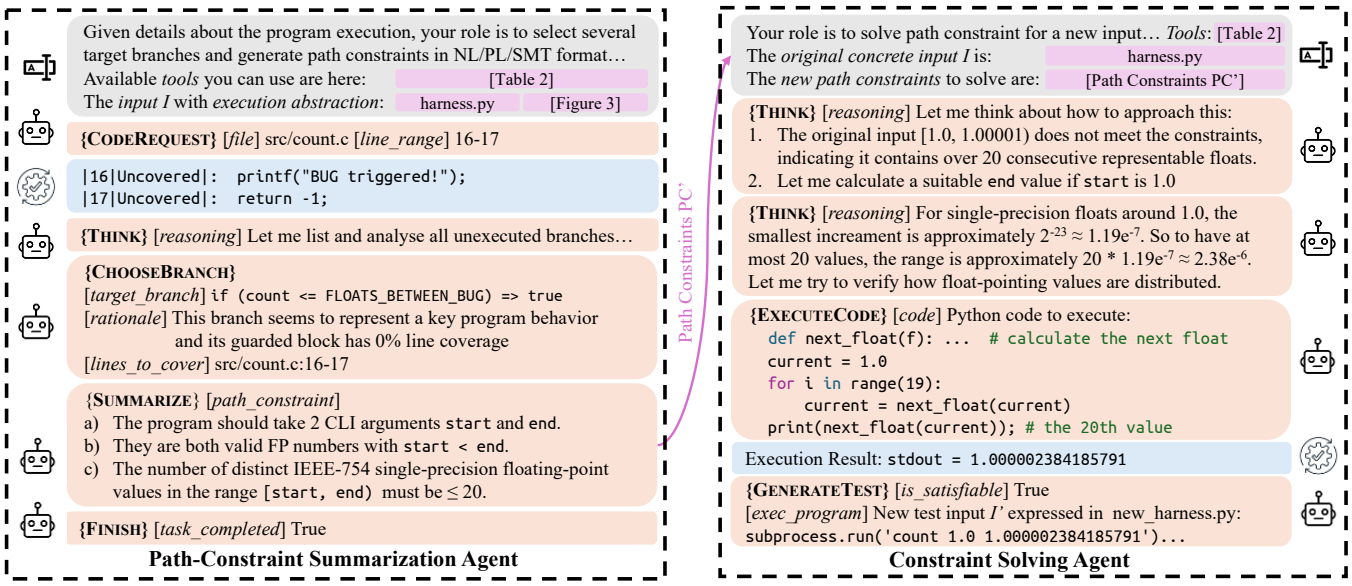


Figure 4: Agentic workflow of the Summarization (left) and Solving (right) Agents on our running example, including the initial instructions with inputs & available tools, actions (formatted as "{TOOLNAME}[*Arguments*]"), and grounded feedback from the tools. Some feedback are omitted for brevity (e.g., acknowledgments of recorded reasoning for THINK).

a sophisticated reasoning process, including analyzing the constraints, performing mathematical calculations to determine feasible input ranges, and resorting to Python execution to compute accurate concrete values. Finally, these two agents collaborate to successfully synthesize a test input I' that can trigger the bug.

Execution and Validation. After obtaining the new test input I', we execute I' and observe the behavior of P to detect potential bugs. To combat potential imprecision arising from the usage of LLM agents for summarization and solving, we perform an additional validation step on the emitted trace to reduce false positives. Namely, we check if the intended `lines_to_cover` are indeed reached in the concrete execution. We note that such validation does *not* provide strict guarantees of correctness, as it depends on our internal coverage tracking, which itself relies on the LLM's instrumentation granularity. We defer the discussions on validation accuracy to §4.4.1. Test inputs that have either successfully reached the target lines, or introduced new coverage elsewhere in the program, are retained in our *WorkList* for further exploration.

In the next iteration, CONCOLLMIC selects a new base input from the *WorkList* for exploration. Path selection is a persistent challenge in symbolic execution, with various strategies offering benefits in different contexts. Our tool directly instantiates this selection by consulting an LLM with contextual information about each input, such as its execution information and its path constraints. We include a preliminary evaluation of this component in Appendix A and observe no significant difference in the performance of our LLM-based selection relative to conventional search strategies like depth-first search, indicating the effect of this selection is not significant.

3.4. Implementation

We implement the prototype of CONCOLLMIC in 8.2k lines of Python code. For the foundational LLM infrastructure, we use claude-3.7-sonnet-20250219 [31] from Anthropic. For the LLM's temperature settings, we use 0 for the instrumentation module, and 0.5 for the two agents. This differentiated setting is because the instrumentation is a relatively deterministic process, while the other agents should be more "creative". To save cost and improve efficiency, we utilize the LLM backend's incremental prompt caching in the multi-turn conversation. Meanwhile, we implement a parallelization mechanism for the Solving Agent. Specifically, given that the Summarization Agent may select multiple target branches and generate multiple corresponding path constraints, we fork distinct solving pipelines for each path constraint, which compensates for the LLM's high response latency and improves the overall testing throughput. For more details, please refer to the open-source repository at https://github.com/ConcoLLMic/ConcoLLMic.

4. Evaluation

In this section, we aim to understand the effectiveness of our proposed approach through an extensive evaluation. After presenting our evaluation setup (§4.1), we first demonstrate CONCOLLMIC's effectiveness on a diverse set of application domains (§4.2), including monolingual (§4.2.1), polyglot (§4.2.2), and floating-point programs (§4.2.3). We then examine the effectiveness of our approach in bug discovery (§4.3). Finally, we assess our design for both stages in CONCOLLMIC to better understand its remarkable performance (§4.4).

TABLE 4: Real-world benchmarks, including monolingual C/C++ (top) and polyglot programs (bottom). Benchmarks marked with "*" were created in 2025, after claude-3.7-sonnet's training cut-off.

Subject	Version	Inputs	Language	# kLoC	Functionality
`woff2`	#0f4d30	Binary (font data)	C++	56	Font compression and decompression
`oggenc`	#235540	Binary (audio data), CLI arguments	C	83	Audio multimedia conversion for GNU
`bc`	1.08.1	Textual (math expression, script), CLI arguments	C	29	Arbitrary precision calc & scripting for GNU
`libmatheval`	1.1.11	Textual (math expression)	C	15	Mathematical expression evaluation for GNU
`libyaml`	#840b65	Textual (YAML file)	C	11	YAML parsing and emitting library
`libsoup`	#eb79a7	Network (HTTP message)	C	71	HTTP client/server library for GNOME
`krep*`	1.2.0	Textual (regex), CLI arguments	C	4	Optimized string search utility
`confetti*`	1.0.0-b4	Textual (customized config. file)	C	9	Unopinionated config. language and parser
`ultrajson`	5.10.0	Textual (JSON data), CLI arguments	Python, C	9	JSON parsing library
`jansi`	2.4.2	Textual (ANSI escape code), CLI arguments	Java, C	72	Console output formatting library
`py4j`	#cb9e39	Textual (Python program and Java object)	Python, Java	21	Python-JVM gateway for object access
`protobuf-go`	#e5d446	Textual (Protobuf schema file), CLI arguments	Go, C++	36	Cross-language data interchange format

4.1. Evaluation Setup

Benchmarks. To evaluate our approach's effectiveness and generality, we compile a set of benchmarks spanning both monolingual C/C++ programs and polyglot systems written in a combination of programming languages, as detailed in Table 4. For monolingual programs, we select diverse benchmarks by referring to prior research [12], [32], [33], [34] and OSS-Fuzz [35] and use existing seed inputs from these projects. For polyglot systems, we choose real-world programs involving complex cross-language interactions. Specifically, as shown in Table 4, our benchmarks exercise varied constraint theories including bit-level operations (binary font/audio data), arithmetic computations (mathematical expressions), structured data parsing (YAML/JSON/HTTP/Protobuf/customized format), string operations (regex patterns), and programming languages (scripts/Python/Java). These targets collectively cover diverse application domains (numerical computation, command-line utilities, parsing libraries, network protocols, and cross-language bridges) and take various input sources (stdin, file data, network messages, and environmental inputs like CLI arguments). To mitigate data leakage and demonstrate the generality of our approach, we also include two projects (`krep` and `confetti`) created *after* claude-3.7-sonnet's training cut-off [36]. Furthermore, we include a floating-point benchmark, FP-bench from Liew et al. [12], as an example of complex constraint reasoning evaluation. These benchmarks collectively highlight key challenges in DSE, such as intricate constraint reasoning and environmental interactions.

Comparison Tools. To the best of our knowledge, no existing symbolic execution frameworks support a wide variety of multi-language systems. We therefore compare against leading C/C++ concolic execution engines to ensure rigorous and meaningful comparison due to the proliferation of well-optimized tools targeting C/C++. To this end, we select the latest versions of:

1) *KLEE* [37], an established DSE engine;
2) *SymCC* [8], a compilation-based concolic executor; and
3) *SymSan* [32], a concolic executor that leverages LLVM's data-flow analysis to optimize constraint collection.

To capture domain-specific strengths, we further include two specialized KLEE variants:

4) *KLEE-Float* [12], augmenting KLEE with precise floating-point reasoning; and
5) *KLEE-Pending* [33], incorporating meta-search heuristics to guide path exploration, thereby improving scalability for larger projects.

In addition to DSE tools, we also include:

6) *AFL++* [38], a state-of-the-art coverage-guided greybox fuzzer. As fuzzing is a standard technique for bug detection, this comparison helps understand ConcoLLMic's effectiveness relative to other common testing methods.

Evaluation Environment. Each campaign runs in a Docker container with 2 CPUs and 8 GiB RAM. To facilitate thorough path exploration, we allocate 48 h for testing real-world subjects. For statistical significance, each tool is run five times per subject. Since ConcoLLMic is more costly to run for an extended period due to LLM API calls, we configure it to exit automatically if there is no increase in internal coverage within 30 min. In total, the combined scale of our evaluation exceeds one year of CPU time.

4.2. Support for Different Application Domains

4.2.1. Monolingual C/C++ Programs. We first compare ConcoLLMic against state-of-the-art DSE and fuzzing tools on monolingual C/C++ programs, using branch coverage reported by GCov [39] as the unified metric.

Effectiveness. Figure 5 presents the GCov branch coverage growth over time for each tool. Despite early termination, ConcoLLMic achieves substantially higher branch coverage than comparison tools after their full 48 h execution: 233%, 135%, 130%, and 115% *more* branches on average than KLEE, KLEE-Pending, SymCC, and SymSan, respectively. Impressively, across most subjects, the *minimum* coverage achieved by ConcoLLMic exceeds the *maximum* coverage of comparison DSE tools across repetitions, demonstrating substantial improvement. With statistical significance (Welch's t-test, p-value < 0.05), ConcoLLMic outperforms *all* competing DSE tools on 7 subjects (`libsoup` is unsupported by any other DSE tools). The only exception is `confetti`, where ConcoLLMic's

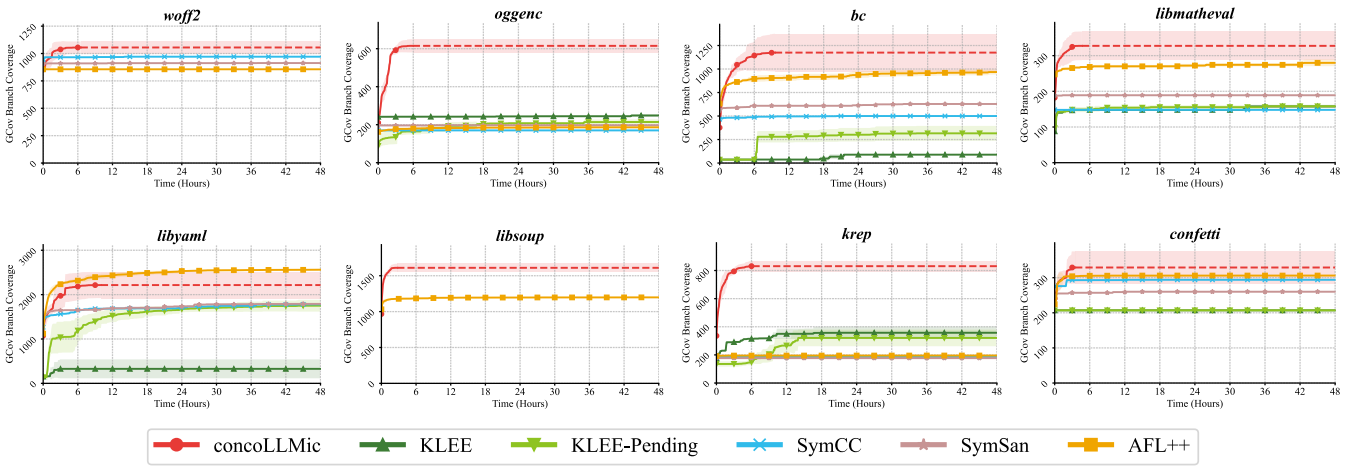


Figure 5: Monolingual C/C++ programs: GCov branch coverage growth over time. Solid lines and shaded areas represent the mean and standard deviation. Dashed lines indicate where ConcoLLMic exited due to 30-minute coverage stagnation. `libsoup` and `woff2` are partially supported by comparison tools due to their incomplete support for network input or C++.

performance is statistically indistinguishable from SymCC's with p-value = 0.2, but is still better than other DSE tools with statistical significance.

When compared to 48-hour AFL++ campaigns, ConcoLLMic achieves 81% *higher* coverage on average across all subjects. Specifically, ConcoLLMic achieves higher mean performance than AFL++ on 7 out of 8 benchmarks, with only `libyaml` showing lower coverage. Besides, we note that it is well-established that symbolic execution and fuzzing are often complementary [4], where fuzzing excels in high throughput, discovering unexpected program behaviors through carefully designed mutation operators [40], and efficiently testing complex system-level software like OS kernels [41], while symbolic execution can provide precise program reasoning for complex constraint scenarios [1], [3]. The two techniques are typically integrated in hybrid approaches to achieve synergistic benefits [26], [42].

Our investigation reveals that ConcoLLMic's superior performance is often due to a combination of high-level constraint reasoning and sophisticated environment handling.

ConcoLLMic is able to *reason symbolically in terms of high-level constraints*. Even for programs taking only file and stdin inputs (e.g., `libmatheval`), ConcoLLMic consistently achieves better coverage through its agentic design. Our design enables the LLM to grasp program semantics and formulate constraints at proper abstraction levels, and deploy an autonomous solving agent to bridge high-level reasoning with precise computational tools through continuous reasoning and problem decomposition (as shown in Figure 4). This differs from conventional symbolic executors that use verbose implementation-level symbolic representation, which places a burden on the solver.

ConcoLLMic has its ability to *handle diverse symbolic sources* that challenge conventional tools. Many programs take not only file and `stdin` inputs, but also various *environmental* inputs, for which existing tools offer incomplete support at best. These environmental inputs can be as simple as additional command line parameters in `oggenc`, for example, where we found that ConcoLLMic can generate precise combinations of arguments ("`-resample -1`" that leads to an abort) and boundary values (e.g., "`-resample 2147483647`" that leads to a crash). In other cases, the environmental inputs synthesized by ConcoLLMic are more complex, as we illustrate via the case study below.

Case Study on Symbolic Reasoning over Environments. To illustrate ConcoLLMic's capability to reason about and manipulate program environments, we present a case study observed when testing `bc`. Figure 6a shows the target branch ConcoLLMic decides to cover (line 4), which handles memory allocation failures in the `bc_malloc` function. This branch remains unexecuted under normal conditions since `malloc` rarely fails in typical testing scenarios, making it usually inaccessible to conventional DSE tools and fuzzers. To reach this target, the Summarization Agent identifies the path constraints: *(1) The program must call the* `bc_malloc` *function, which happens in various parts of the program during initialization and operation; (2) The call to* `malloc(size)` *within* `bc_malloc` *must return NULL, indicating a memory allocation failure.* Based on these constraints, the Solving Agent autonomously devises a sophisticated solution by constructing a test harness that manipulates the program environment. As shown in Figure 6b, the Solving Agent implements a multi-step approach: it creates a custom memory allocator wrapper in C (lines 2-31) that intercepts `malloc` calls and forces allocation failure after a predetermined number of invocations. The agent then compiles this wrapper into a shared library (lines 42-44) and employs `LD_PRELOAD` to inject it into the program's execution environment to replace the standard `libc`'s `malloc` implementation (lines 46-48). This approach resembles sophisticated fault injection techniques that developers might employ manually for testing error-handling paths. However, ConcoLLMic accomplishes this entire process *entirely autonomously*, demonstrating the LLM's rich en-

```c
void * bc_malloc (size_t size) {
  void *ptr = (void *) malloc (size);
  if (ptr == NULL) {
    // Unexecuted line 723 removed. Line cov: 0/1
  }
  return ptr;
}
```

(a) CONCOLLMIC chooses to target the branch on line 4.

```python
# Create a malloc wrapper that will fail after a certain number of calls
malloc_wrapper_c = """
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

static void* (*real_malloc)(size_t) = NULL;
static int malloc_count = 0;
static int fail_after = 30;  // Fail after this many calls

void* malloc(size_t size) {
// Initialize real_malloc on first call
if (real_malloc == NULL) {
  real_malloc = (void* (*)(size_t))dlsym(RTLD_NEXT, "malloc");
  if (real_malloc == NULL) {
    fprintf(stderr, "Error finding real malloc\\n");
    return NULL;
  }
}

malloc_count++;

// Fail after a certain number of calls
if (malloc_count >= fail_after) {
  return NULL;  // Simulate allocation failure
}

return real_malloc(size);
}
"""

def execute_program():  # The test harness produced by ConcoLLMic's GenerateTest action

  # Create a non-existent filename
  non_existent_file = "non_existent_file.bc"

  # Write the C code to a file
  with open("malloc_wrapper.c", "w") as f:
    f.write(malloc_wrapper_c)

  # Compile the malloc wrapper
  compile_result = subprocess.run(
    ["gcc", "-shared", "-fPIC", "malloc_wrapper.c", "-o", "malloc_wrapper.so", "-ldl"])

  # Run bc with our malloc wrapper
  env = os.environ.copy()
  env["LD_PRELOAD"] = "./malloc_wrapper.so"

  result = subprocess.run(
    ["./bc/bc", non_existent_file],
    env=env
  )
  # ...
```

(b) CONCOLLMIC-generated test input (in *harness.py*) to cover the above target branch. All the code is provided by the GENERATETEST action (shown in Table 3).

Figure 6: Case study: CONCOLLMIC generates a test input for `bc` to trigger the code path handling `malloc` failure, demonstrating its ability to reason about and synthesize complex program environments.

gineering capabilities in bridging high-level symbolic reasoning with precise environmental manipulation. The generated test input successfully triggers the previously unreachable branch, showcasing CONCOLLMIC's effectiveness in reasoning symbolically about program environments and synthesizing complex environmental conditions that extend beyond conventional input generation.

Generality. We highlight that our benchmarks comprise domains that are challenging for foundational language models themselves. For example, `oggenc` and `woff2` take *binary* inputs rather than textual content. Still, we observe that our agentic framework enables the effective generation of structured hexadecimal data. Our analysis of CONCOLLMIC's action logs reveals an iterative process where the LLM understands and formulates theories about binary structures, then validates and refines them through experimental hex stream probing. Upon identifying satisfiable solutions, CONCOLLMIC synthesizes field values and encodes them as valid binary data using Python scripts. This synergy between LLM reasoning and computational tools bridges the gap between abstract semantic understanding and precise binary manipulation, compensating for inherent language model limitations in processing raw binary data.

Additionally, CONCOLLMIC retains its competitive edge across different categories of programs: (1) *Network applications* like `libsoup` receive input data over the network socket, which is not supported by any other tools in our evaluation. In contrast, CONCOLLMIC attains meaningful coverage increases on this program and even discovers a new bug (see §4.3); (2) *C++ projects* like `woff2`, where traditional tools encounter limitations due to incomplete language and standard library support; and (3) *Unseen codebases* like `krep` and `confetti` are not included in claude-3.7-sonnet's training corpus, but we do not notice a degradation in performance and also discover two new bugs in them.

4.2.2. Polyglot Programs. With *zero* additional configuration, CONCOLLMIC fully supports *polyglot* systems comprising of more than one source programming language by using a uniform execution abstraction. Our benchmark subjects exercise complex cross-language interactions and logic (see Table 4): `ultrajson` and `jansi` interface C backends from Python and Java respectively; `py4j` facilitates access to JVM objects from the Python runtime; and `protobuf-go` augments a C++ compiler with a Go plugin.

Due to the lack of competing multi-language tools and difficulty in tracking cross-language coverage, we report CONCOLLMIC's internal line coverage (validated with 94% correlation to ground truth in §4.4.1). Figure 7 shows consistent coverage growth across all subjects. On average, coverage grows 3.5×, 8.2×, 1.9×, and 1.9× from the initial test inputs, on `ultrajson`, `jansi`, `py4j`, and `protobuf-go`, respectively. This demonstrates CONCOLLMIC's ability to explore unexplored behaviors in polyglot systems, which is not supported by existing tools. Additionally, CONCOLLMIC discovered previously unknown bugs in two of

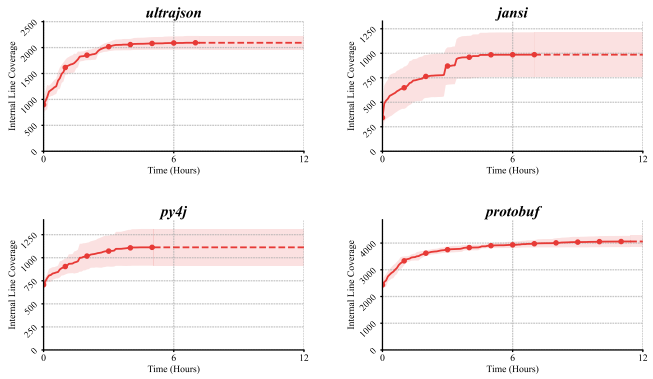


Figure 7: Multi-lingual programs: internal line coverage growth over time, with mean and standard deviation. Dashed lines indicate exits due to 30 minute coverage stagnation.

TABLE 5: Average per-program overhead and performance (GCov branch coverage) on FP-Bench (26 programs total).

CONCOLLMIC			KLEE-Float		KLEE	
Cov (%)	Time (s)	Cost ($)	Cov (%)	Time (s)	Cov (%)	Time (s)
77.87	464	1.05	64.81	46	37.67	0.03

these programs (see §4.3). The detailed cost statistics are reported in §4.4.2.

4.2.3. Floating-Point Code. To assess constraint distillation and solving capabilities on a challenging constraint solving theory (**C2**), we evaluate CONCOLLMIC on FP-bench [12], which consists of 30 synthetic floating-point programs (20-242 lines each) carefully curated to expose symbolic reasoning challenges on floating points. We exclude four programs unsuitable for test generation (e.g., those verifying non-termination). For this benchmark, we compare against KLEE-Float [12], a specialized tool for the floating-point domain, and also include its baseline KLEE to show the importance of theory support.

Table 5 shows CONCOLLMIC achieves 20% and 107% higher branch coverage than KLEE-Float and KLEE, respectively. The results reveal two key insights about constraint handling effectiveness:

1) *Theory support for constraint distillation.* The comparison between KLEE and KLEE-Float demonstrates the importance of theory-specific support. KLEE concretizes all paths due to inadequate floating-point support, leading to many unsolvable constraints and achieving low code coverage. Instead, KLEE-Float advances considerably (73% improvement over KLEE) through dedicated modeling and optimization for floating-point arithmetic. Leveraging the LLM's internal knowledge, CONCOLLMIC naturally grasps floating-point semantics.
2) *High-Level constraint reasoning.* More importantly, despite KLEE-Float's specialized theory support, CONCOLLMIC still outperforms it in terms of code coverage by over 20%. Manual log inspection reveals that KLEE-Float struggles with complex data structures containing floating-point numbers, which complicates traditional constraint distillation and stresses the solver, a fundamental challenge faced by existing tools. Instead, CONCOLLMIC addresses this challenge by working on high-level constraint reasoning that captures program semantics, coupled with an autonomous solving agent to understand and solve these constraints, as illustrated in §2.1 and Figure 4. While CONCOLLMIC's test generation is slower than KLEE-Float, our evaluation on real-world large programs shows this overhead stems primarily from LLM query latency (see §4.4.2), which disproportionately impacts smaller benchmarks.

4.3. Bug Discovery

Beyond code coverage improvement, we evaluate CONCOLLMIC's bug detection capabilities against comparison tools. To enable systematic and fair bug detection, we instrument all C/C++ benchmarks (Table 4) with AddressSanitizer [43] and UndefinedBehaviorSanitizer [44] (ASan and UBSan) to capture memory safety and undefined behavior violations, then use program crashes as the primary detection signal. For multi-language benchmarks, we detect bugs through runtime exceptions and crashes.

In total, CONCOLLMIC uncovered 11 previously unknown bugs across both the latest C/C++ and multi-language benchmarks, as detailed in Table 6. Each discovered bug is validated through concrete execution using a Python harness (in *harness.py*, as shown in Figure 6b) that captures all constraints (including inputs, environment settings, program arguments, etc.) for reproducible verification, ensuring no false positives. Hence, manual effort is limited to bug deduplication, which follows the same process as in existing testing tools by analyzing sanitizer or GDB stack traces.

Nine vulnerabilities have been confirmed or fixed by the developers. Bug #8 in `libsoup` has been assigned the identifier CVE-2025-4945, while others are undergoing the CVE application process after a coordinated disclosure. These subjects are well-tested production software, with four projects integrated into OSS-Fuzz [35] for continuous testing, underscoring CONCOLLMIC's effectiveness in reaching logic that existing tools struggle with. Notably, CONCOLLMIC also uncovered previously unknown bugs in polyglot systems like `jansi` and `ultrajson`, which involve complex interactions across language boundaries and require cross-language constraint reasoning.

Most bugs discovered by CONCOLLMIC are also hard to trigger with other tools—KLEE, KLEE-Pending, SymCC, SymSan, and AFL++ exposed only 1, 1, 4, 5, and 3 of them, respectively. Among these baseline tools, only KLEE-Pending identified one additional bug beyond CONCOLLMIC's discoveries, specifically in `krep` involving an invalid pointer passed to `realloc`. Our examination reveals that CONCOLLMIC also reached the same error code location, but did not construct the specific malicious input values needed to trigger that particular bug. This stems from CONCOLLMIC's current design, where the execution abstraction (e.g., Figure 3) provided to the LLM primarily focuses on coverage information to guide systematic path

TABLE 6: Previously unknown vulnerabilities exposed by CONCOLLMIC. Six of them cannot be found by existing tools.

#	Subject	Bug Description	Status
1	`oggenc`	Null pointer dereference in oggenc.c when encoding an audio file with invalid format	Fixed
2	`oggenc`	Signed integer overflow in oggenc.c when resampling the input audio	Reported
3	`oggenc`	Signed integer overflow when processing WAV file with malformed INFO chunk size	Fixed
4	`oggenc`	Memory leak in wav_open() when processing WAV files with invalid header structure	Fixed
5	`krep`	Incorrect handling of arguments PATTERN and STRING_TO_SEARCH in the String Mode -s	Fixed
6	`libyaml`	Memory leak due to missing yaml_parser_delete() in case of invalid UTF-8 input	Fixed
7	`libyaml`	Memory leak due to missing yaml_parser_delete() in case of incomplete UTF-8 octet sequence	Confirmed
8	`libsoup`	Integer overflow in soup_cookie_parse() when parsing a cookie with malformed "expired" value	Fixed
9	`confetti`	Memory leak in _readstdin.c and parse.c after parsing a partially malformed configuration input	Fixed
10	`ultrajson`	Wrong exception handling in python/JSONtoObj.c when parsing a JSON string with nested keys	Fixed
11	`jansi`	ClassCastException when processing ANSI escape sequences with quoted string arguments	Reported

exploration rather than explicitly instructing it to adopt an adversarial perspective for generating malicious inputs at potentially vulnerable code locations, though it may autonomously adopt such strategies based on its own reasoning. Incorporating explicit adversarial input generation strategies at strategic code locations represents an important direction for future research.

Through manual inspection, we find that bugs found only by CONCOLLMIC often require satisfying complex constraint combinations that challenge conventional tools. For example, bug #3 is only triggered with a precisely corrupted WAV file INFO chunk, bug #5 occurs only in a particular mode set by a CLI argument, while bug #8 involves both HTTP message parsing and arithmetic constraints on malformed cookie timestamps, highlighting CONCOLLMIC's capability in handling multi-faceted constraints.

4.4. Evaluating CONCOLLMIC's Design

4.4.1. Instrumentation Stage. First, we evaluate the cost, scalability, and fidelity of CONCOLLMIC's instrumentation module shown in §3.2.

Cost. Across all benchmarks, the instrumentation cost is $0.56 per 1 kLoC on average, with a detailed breakdown shown in Table 7. Costs vary depending on the program's language and logic complexity. As discussed in §3.2, we highlight that the full instrumentation represents a *one-time cost* per project, as CONCOLLMIC's instrumentation design allows for incremental re-instrumentation of *only* modified functions, reducing ongoing costs as the program evolves.

Scalability. As shown in column "Max. File" in Table 7, CONCOLLMIC has successfully processed individual files of up to 3,989 lines through function-granular instrumentation. The instrumentation design of decomposing large files into manageable syntactically-coherent chunks demonstrates scalability across diverse and complex projects.

Fidelity. CONCOLLMIC's internal coverage maintenance relies on the instrumentation module and is crucial for downstream testing. To this end, we empirically assess its fidelity against GCov on all real-world C/C++ benchmarks.

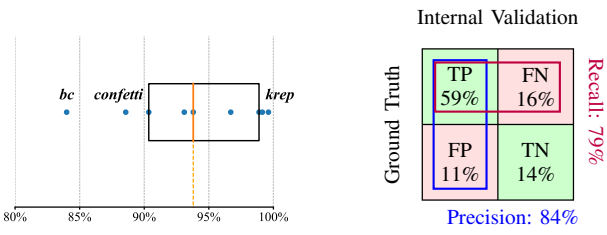
1) *Line Coverage Correlation.* First, we measure the overall correlation. We compute the Pearson's correlation coefficient between CONCOLLMIC's internal line coverage and GCov line coverage, as shown in Figure 8a. A mean and median correlation of 94% across all benchmarks indicates a very strong linear relationship. The observed divergence stems from two main factors: (i) a small portion of control flow constructs remain uninstrumented by the LLM. (ii) GCov excludes variable declarations and assignments (e.g., line 7 in Figure 1) while CONCOLLMIC includes them if they are wrapped by flow-tracing statements. Despite these differences, the consistently high correlation shows that our instrumentation accurately captures program execution flow.
2) *Validation Accuracy.* CONCOLLMIC's validation check (⑤ in Figure 2) verifies if newly-generated test inputs cover their intended target lines—a criterion for retaining them. Since target lines are typically few, evaluating this validation check's accuracy is more fine-grained. Figure 8b presents the results of this validation across the union of all generated test inputs across repetitions and benchmarks, where columns represent CONCOLLMIC's internal predictions and rows represent GCov coverage status. Our validation achieves a precision of 84% and an F1-score of 81%, indicating that 84% of positive predictions are correct while striking a balance between precision and recall. Importantly, CONCOLLMIC uses a dual-criterion approach that also retains a test input when it produces new global coverage. This dual-criterion approach, combined with highly correlated line coverage tracking (94%), alleviates the issue of overlooking valuable inputs (false negatives). This is because flipping new branches often results in new code being covered, which can be captured by an increase in overall line coverage, even when specific target lines are missed.

4.4.2. Testing Stage. We next evaluate the concolic testing stage shown in §3.3. Specifically, we evaluate the cost, throughput, and breakdown of LLM invocations during testing. We also conduct a case study to gain a deeper understanding of CONCOLLMIC's test input generation process.

Cost and Throughput. On average, CONCOLLMIC spends $0.21 and 69 seconds to generate each test input. Table 7 presents the detailed runtime statistics (column "Avg. Time"), total generated inputs (column "Avg. # Inputs"), and LLM costs (column "Avg. Cost") for each subject, with all statistics aggregated across repetitions. In terms of com-

TABLE 7: CONCOLLMIC's detailed instrumentation and testing statistics (aggregated across repetitions) for each benchmark. FP-Bench reports the aggregated data over all 26 programs. "# Instr. LoC" is the total number of lines of code instrumented. "Avg. Cost" is calculated per 1 kLoC for instrumentation or per test input for testing. "Avg. Time" is CONCOLLMIC's testing runtime per subject with early termination triggered after 30-min coverage stagnation. "Total Cost" is the sum of (i) instrumentation cost, and (ii) the average testing cost aggregated across repetitions.

Subject	Stage-1: Instrumentation			Stage-2: Testing			Total
	# Instr. LoC	Max. File (LoC)	Avg. Cost ($)	Avg. # Inputs	Avg. Time (h)	Avg. Cost ($)	Cost ($)
woff2	11,144	2,491	0.57	171	4.02	0.28	73.77
oggenc	8,620	1,961	0.52	257	4.03	0.18	66.01
bc	1,888	2,703	0.24	281	4.56	0.18	48.50
libmatheval	844	1,991	0.29	129	2.55	0.20	28.01
libyaml	3,592	3,598	0.66	248	5.81	0.26	74.43
libsoup	9,744	3,989	0.64	97	1.87	0.35	73.02
krep	1,820	3,945	0.89	216	3.85	0.17	43.72
confetti	678	2,035	1.19	119	2.12	0.14	20.23
ultrajson	904	1,001	0.27	255	5.09	0.21	54.65
jansi	2,002	973	0.41	255	4.17	0.21	57.33
py4j	2,084	1,024	0.29	134	3.15	0.27	39.54
protobuf-go	2,080	3,238	1.06	309	8.13	0.28	93.67
FP-Bench	670	242	0.31	334	3.37	0.08	27.33
Avg (Total)	**(46,070)**	**2,245**	**0.56**	**(2,805)**	**4.06**	**0.21**	**53.86**



(a) Box plot of correlation coefficients between internal and GCov line coverage.

(b) Confusion matrix of all validation results, with GCov as the ground truth.

Figure 8: Instrumentation fidelity on C/C++ Benchmarks.

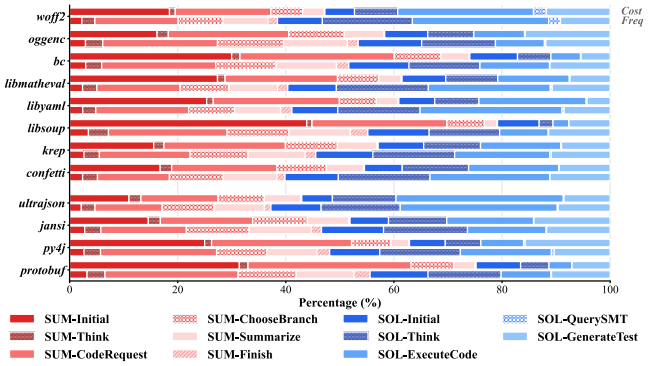


Figure 9: The cost (top) and frequency (bottom) of different LLM invocations across benchmarks, from either the summarization (SUM) or solving (SOL) agents. SUM/SOL-INITIAL refers to the initial input and instructions.

putational resources, CONCOLLMIC is very lightweight: 14% of one CPU core (2.7GHz) and 266MB RAM is consumed on average, indicating that CPU and memory are not performance bottlenecks. However, our approach is substantially slower than conventional tools due to the LLM's inherent latency. By comparison, SymCC generates 700+ unique inputs on bc in 60 seconds, being 800× faster. Nonetheless, our approach prioritizes *test input quality* over quantity. This quality-oriented strategy yields remarkable effectiveness: to achieve the final branch coverage attained by the **best-performing comparison DSE tools** on each subject after their 48 h testing, CONCOLLMIC requires only **31 test inputs** on average, costing approximately **$6.1** and taking just **32 min**. This reflects CONCOLLMIC's ability to generate highly targeted inputs, effectively reaching program paths that are out of reach for traditional tools (see §4.2.1). As a research prototype, our tool currently prioritizes correctness and generality over cost efficiency.

LLM Invocation Breakdown. Figure 9 shows the cost and frequency of different LLM components, including initial instructions and subsequent actions of the two LLM agents for constraint summarization (SUM) and solving (SOL). Overall, SUM-INITIAL and SUM-CODEREQUEST are the most expensive components, accounting for 23.0% and 22.2% of total cost, as their cost scales linearly with input execution abstraction size or requested code size. Notably, for the solving agent, SOL-THINK and SOL-EXECUTECODE represent significant portions, accounting for (8.4%, 15.0%) and (13.9%, 17.9%) of (cost, frequency), respectively. This relatively considerable cost of SOL-THINK—primarily reasoning and planning—demonstrates the LLM's extensive deliberation when tackling complex constraint-solving problems.

Our log analysis shows the LLM typically follows a think-execute-validate pattern: after each THINK action, it frequently invokes EXECUTECODE to compute or validate solutions, then returns to THINK to refine its approach iteratively. This iterative workflow reflects a key advantage of our *agentic* design: the LLM engages in multi-step reasoning with continuous validation, increasing its *trustworthiness* in complex analysis scenarios.

We observe significant variation in action selection across subjects. For example, QUERYSMT is repeatedly employed for `woff2`, yet rarely for other subjects. This is because `woff2`'s strict binary format requires satisfying multiple interrelated constraints across fields, a task better suited for SMT representation. This reflects the autonomy of CONCOLLMIC in carrying out *context-aware workflows* based on program semantics and constraint complexity.

Failure Case Analysis. To understand when and why CONCOLLMIC fails to generate effective test inputs, we conducted a detailed manual analysis of 50 randomly sampled failure cases from `oggenc`. Specifically, a test generation iteration is considered *successful* when the generated input reaches the target branch selected by the Summarization Agent, and a *failure* otherwise. Our analysis reveals five primary categories of failures:

1) *Infeasible Paths (36%):* The selected target branches are inherently unreachable under the current program configuration, such as those involving conditional compilation directives (e.g., `#ifdef`) that require recompilation.
2) *Bug-Induced Crashes (30%):* The program encounters a bug before reaching the intended target location, causing premature termination.
3) *Summarization Errors (24%):* The Summarization Agent generates incorrect or incomplete path constraints that prevent reaching the target branch.
4) *Solving Inconsistencies (6%):* The Solving Agent produces test inputs that are inconsistent with the path constraints summarized by the Summarization Agent.
5) *Execution Timeouts (4%):* The program's execution exceeds our 10-second timeout limit, typically due to computationally expensive operations triggered by the generated input.

5. Discussion and Future Work

Cost and Hybrid Integration. CONCOLLMIC is more expensive than traditional tools due to the high cost of LLM invocations. At the same time, it highly favors *quality* over quantity, as described at length in §4.4. Future work could investigate suitable hybrid approaches (e.g., combining CONCOLLMIC with fuzzers or conventional engines) to achieve the desired trade-off in terms of cost and effectiveness. Akin to established concolic executors [8], [32], CONCOLLMIC could also be employed in conjunction with fuzzing for higher throughput and lower cost [26], [42]. However, the integration needs to be carefully designed since fuzzers like AFL++ [38] are often language-specific (applicable to C/C++ only) and, more importantly, lack complete environmental manipulation support (e.g., CONCOLLMIC-generated test cases involving environmental manipulation as in Figure 6b cannot directly be used by AFL++ currently). Hence, finding a suitable integration remains an area for future work.

Model Context Window. Our approach is limited by the size of the context windows of state-of-the-art language models. While the context window size continues to grow [45], it is possible that this window could be exhausted for very large programs. In our evaluation of 12 real-world programs, most containing tens of kLoC, we did not observe any errors relating to Claude Sonnet 3.7's 200K token context window. For larger programs that generate extensive execution traces, while simple trace truncation to a prefix serves as a practical workaround, CONCOLLMIC can be further extended to employ incremental trace summarization that processes traces in sequential segments.

Unsoundness of LLMs and Our Mitigation. Our approach is also limited by the inherent unsoundness of LLMs (i.e., LLMs may give factually wrong answers to queries). This unsoundness means that concolic execution based on LLMs cannot be used for program *verification*. Our design combats such unsoundness in three ways:

1) *Input grounding with concrete execution*: We anchor LLM reasoning in concrete execution traces through a carefully-designed *execution abstraction* that captures program behavior and coverage information, reducing hallucination by grounding symbolic reasoning in observable facts;
2) *Output validation*: We implement comprehensive validation of LLM outputs, including (a) instrumentation validation and (b) end-to-end reachability validation in every iteration, filtering out incorrect reasoning to prevent error propagation;
3) *Tool-augmented reasoning*: Our agentic framework equips LLMs with trusted external tools—e.g., CODEREQUEST for code context understanding and EXECUTECODE for precise computation—enabling grounded feedback when reasoning.

As a result, we see consistent coverage growth across diverse subjects (§4.2), and the internal coverage based on our instrumentation correlates strongly with ground-truth coverage (§4.4.1).

Data Leakage. Our *agentic* concolic executor, CONCOLLMIC, is instantiated with Claude Sonnet 3.7, which has been trained on many open-source projects. Thus, it is possible that our evaluation benchmarks overlap with its training data, and this data leakage could inflate our results. To mitigate this, we included two projects (`krep` and `confetti`) which were not publicly available until after the training cut-off of the model [36]. In §4, we do not see any qualitative difference in the results on these two subjects.

Adversarial Robustness. A natural concern for LLM-based systems is whether malicious actors could deliberately induce hallucinations in critical reasoning processes, potentially causing incorrect path constraints or solutions. CONCOLLMIC's end-to-end reachability validation, adapted in every iteration, prevents error propagation—any test input failing to reach its intended target or increase coverage is discarded, containing the impact of potential adversarial manipulation. While such issues may still cause CONCOLLMIC to *miss bugs*—a limitation shared with other testing approaches—we see this as low impact relative to other applications of LLMs such as code generation, where hallucinations can *introduce new bugs* into the codebase.

Trade-Off between High- and Low-Level Constraints. High-level constraints (e.g., natural language) offer greater generality and semantic clarity, while low-level constraints (e.g., SMT formulas) provide implementation-specific precision and formal rigor. Our current design allows LLMs to autonomously select appropriate abstraction levels based on context and problem complexity. Despite the effectiveness of our flexible, semantics-oriented design, developing principled strategies to select an appropriate abstraction level represents a promising direction for future research.

6. Related Work

Symbolic and Concolic Execution. Symbolic execution has been extensively studied since its inception [22], [23], [24], and DART [6] introduced concolic execution, the paradigm on which CONCOLLMIC is based. KLEE [3], [20] remains the leading non-concolic SE tool in widespread use. We include direct comparison to modern compiler-based concolic executors [8], [32] and the KLEE family [3], [12], [33]. Prior attempts to address **(C1)** include environmental modeling [3], [27] and the adoption of hypervisor embedding [46]. To address **(C2)**, the research community has proposed numerous constraint solving optimization techniques, such as expression simplifications [10], counterexample caching [3], [47], [48], rewriting complex array constraints [11], interval solving [49], as well as incomplete and gradient-based solvers [18], [50], [51]. Compared to existing work in this area, CONCOLLMIC instead builds on LLMs' rich capabilities to construct the first concolic executor that is both language- and theory-agnostic.

Machine Learning for Symbolic Execution. Prior work has explored integrating machine learning techniques into DSE, primarily to replace static heuristics with learned components. MLB [52] applies machine learning to path feasibility analysis, while Learch [53], Cottontail [54], and SyML [55] propose learning-based strategies for path selection and exploration. These approaches are orthogonal to our core contributions of addressing symbolic modeling complexity and constraint solving scalability.

LLM-Assisted Software Testing. LLMs have shown initial promise in the broader domain of test generation techniques beyond DSE, such as in unit test generation [56], [57] and fuzzing [58], [59], [60]. In these approaches, LLMs are used to analyze source methods and to extract inputs or state-space specifications. All of these approaches, however, only use static prompt-engineering, rather than CONCOLLMIC's agentic framework or *symbolic* reasoning.

LLM-Reasoning. Recently LLM agents [16] and "Reasoning Models" have outperformed traditional training [61] or inference-time prompting [62], [63] on challenging software engineering [64] and mathematics [14] benchmarks. CONCOLLMIC continues this line of research. However, in this work, we assess LLM agents' ability to conduct *symbolic* reasoning on complex real-world software, which we see as complementary to mathematical reasoning tasks.

7. Perspectives

Software security is of vital importance, increasingly so with the rise of agentic, AI-assisted programming; the recent unprecedented growth in AI-generated code necessitates highly automated approaches to both prevent vulnerabilities and reduce the trust-deficit in these systems [65]. It is thus critical that the tools and techniques we propose as researchers can also be applied in the real world. Indeed, concolic execution has long been at the intersection of the theoretical and the practical—the symbolic and the concrete. In this work, we propose CONCOLLMIC and the paradigm of *agentic concolic execution*. Our results show that LLM agents —with access to planning and tools—are remarkably capable of program instrumentation, trace summarization, and symbolic constraint solving. In particular, we find that *concrete* executions and an agentic framework help ground LLMs to produce fewer faulty answers. Our work can help navigate the tension between "programming at scale" and "programming with trust" in future AI-assisted programming [66], where AI agents may significantly enable both code generation and code validation. We believe that CONCOLLMIC can serve as a foundational framework for future research, enabling more adaptive, context-aware, and scalable symbolic execution systems. Beyond software security, we see this as a compelling opportunity to explore and expand the boundaries of what kind of reasoning LLM agents are truly capable of, especially in domains requiring precise, structured, and symbolic analysis.

Acknowledgments

We sincerely thank the anonymous reviewers and our shepherd for their valuable feedback and guidance. This research is supported by the National Research Foundation, Singapore, the Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Fuzz Testing <NRF-NCR25-Fuzz-0001>), and the European Research Council under the European Union's Horizon 2020 research and innovation program (grant agreement 819141). Any opinions, findings and conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Research Foundation, Singapore, the Cyber Security Agency of Singapore or the European Research Council. The last author, Professor Abhik Roychoudhury, also serves as a senior advisor at SonarSource.

References

[1] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Communications of the Association for Computing Machinery (CACM 2013)*, vol. 56, no. 2, pp. 82–90, 2013.

[2] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of the 13th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE-13, 2005, pp. 263—272.

[3] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, Dec. 2008.

[4] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *Proceedings of the 23rd Network and Distributed System Security Symposium (NDSS'16)*, Feb. 2016.

[5] T. Avgerinos, D. Brumley, J. Davis, R. Goulden, T. Nighswander, A. Rebert, and N. Williamson, "The Mayhem cyber reasoning system," *IEEE Security & Privacy*, vol. 16, no. 2, pp. 52–60, 2018.

[6] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the Conference on Programing Language Design and Implementation (PLDI'05)*, Jun. 2005.

[7] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, pp. 20–27, 2012.

[8] S. Poeplau and A. Francillon, "Symbolic execution with SymCC: Don't interpret, compile!" in *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*, Aug. 2020.

[9] T. Kapus, M. Nowack, and C. Cadar, "Constraints in dynamic symbolic execution: Bitvectors or integers?" in *Tests and Proofs: 13th International Conference, TAP 2019, Held as Part of the Third World Congress on Formal Methods 2019, Porto, Portugal, October 9–11, 2019, Proceedings 13*. Springer, 2019, pp. 41–54.

[10] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler, "EXE: Automatically generating inputs of death," in *ACM Conference on Computer and Communications Security (CCS 2006)*, 10 2006, pp. 322–335.

[11] D. M. Perry, A. Mattavelli, X. Zhang, and C. Cadar, "Accelerating array constraints in symbolic execution," in *International Symposium on Software Testing and Analysis (ISSTA 2017)*, 7 2017, pp. 68–78.

[12] D. Liew, D. Schemmel, C. Cadar, A. F. Donaldson, R. Zahl, and K. Wehrle, "Floating-point symbolic execution: a case study in n-version programming," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 601–612.

[13] T. Kapus, O. Ish-Shalom, S. Itzhaky, N. Rinetzky, and C. Cadar, "Computing summaries of string loops in C for better testing and refactoring," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*, 6 2019, pp. 874–888.

[14] P. Lu, H. Bansal, T. Xia, J. Liu, C. yue Li, H. Hajishirzi, H. Cheng, K.-W. Chang, M. Galley, and J. Gao, "MathVista: Evaluating mathematical reasoning of foundation models in visual contexts," in *International Conference on Learning Representations*, 2023.

[15] R. Liu, H. Shi, S. Liu, C. Hu, S. Li, Y. Shen, R. Wang, X. Shi, and Y. Jiang, "Patchscope: Llm-enhanced fine-grained stable patch classification for linux kernel," *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, 2025.

[16] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "AutoCodeRover: Autonomous program improvement," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1592–1604.

[17] I. Bouzenia and M. Pradel, "You name it, I run it: An LLM agent to execute tests of arbitrary projects," *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, 2025.

[18] J. Chen, J. Wang, C. Song, and H. Yin, "Jigsaw: Efficient and scalable path constraints fuzzing," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 18–35.

[19] H. Palikareva and C. Cadar, "Multi-solver support in symbolic execution," in *International Conference on Computer Aided Verification (CAV 2013)*, 7 2013, pp. 53–68.

[20] C. Cadar and M. Nowack, "KLEE symbolic execution engine in 2019," *International Journal on Software Tools for Technology Transfer (2020)*, vol. 23, no. 6, pp. 867–870, 2020.

[21] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[22] L. A. Clarke, "A program testing system," in *Proceedings of the 1976 annual conference*, 1976, pp. 488–491.

[23] R. S. Boyer, B. Elspas, and K. N. Levitt, "Select—a formal system for testing and debugging programs by symbolic execution," *ACM SigPlan Notices*, vol. 10, no. 6, pp. 234–245, 1975.

[24] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[25] T. Kapus and C. Cadar, "A segmented memory model for symbolic execution," in *European Software Engineering Conference / ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*, 8 2019, pp. 774–784.

[26] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "Qsym: A practical concolic execution engine tailored for hybrid fuzzing," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*, Aug. 2018.

[27] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11, 2011, p. 183–198.

[28] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB Standard: Version 2.0," Department of Computer Science, The University of Iowa, Tech. Rep., 2010, available at `www.SMT-LIB.org`.

[29] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Computer Aided Verification*. Springer Berlin Heidelberg, 2007, pp. 519–531.

[30] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 171–177.

[31] Anthropic. (2025) Claude 3.7 sonnet. https://docs.anthropic.com/en/docs/about-claude/models/overview.

[32] J. Chen, W. Han, M. Yin, H. Zeng, C. Song, B. Lee, H. Yin, and I. Shin, "SymSan: Time and space efficient concolic execution via dynamic data-flow analysis," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2531–2548.

[33] T. Kapus, F. Busse, and C. Cadar, "Pending constraints in symbolic execution for better exploration and seeding," in *Proceedings of the 35th IEEE International Conference on Automated Software Engineering (ASE'20)*, Sep. 2020. [Online]. Available: https://doi.org/10.1145/3324884.3416589

[34] Z. Qi, J. Hu, Z. Xiao, and H. Yin, "SymFit: Making the common (concrete) case fast for Binary-Code concolic execution," in *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, 2024, pp. 415–432.

[35] K. Serebryany, "OSS-Fuzz: Google's continuous fuzzing for open-source software." Vancouver, BC: USENIX Association, Aug 2017.

[36] Anthropic. (2025) Claude's training data. https://support.anthropic.com/en/articles/8114494-how-up-to-date-is-claude-s-training-data.

[37] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2123–2138.

[38] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT'20)*, 2020.

[39] "Gcov Test Coverage," gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[40] Z. Luo, J. Yu, F. Zuo, J. Liu, Y. Jiang, T. Chen, A. Roychoudhury, and J. Sun, "Bleem: Packet sequence oriented fuzzing for protocol implementations," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4481–4498.

[41] H. Shi, S. Chen, R. Wang, Y. Chen, W. Zhang, Q. Zhang, Y. Shen, X. Shi, C. Hu, and Y. Jiang, "Industry practice of directed kernel fuzzing for open-source linux distribution," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024.

[42] L. Jiang, H. Yuan, M. Wu, L. Zhang, and Y. Zhang, "Evaluating and improving hybrid fuzzing," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023.

[43] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," *2012 USENIX Annual Technical Conference*, 2012.

[44] LLVM, "Clang 22.0.0 documentation, undefinedbehaviorsanitizer," https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[45] Anthropic, "Claude Sonnet 4 now supports 1M tokens of context," https://www.anthropic.com/news/1m-context.

[46] V. Chipounov, V. Kuznetsov, and G. Candea, "The S2E platform: Design, implementation, and applications," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 1, pp. 1–49, 2012.

[47] W. Visser, J. Geldenhuys, and M. B. Dwyer, "Green: reducing, reusing and recycling constraints in program analysis," in *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'12)*, Nov. 2012.

[48] A. Aquino, F. A. Bianchi, M. Chen, G. Denaro, and M. Pezzè, "Reusing constraint proofs in program analysis," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015, 2015, p. 305–315.

[49] O. S. Dustmann, K. Wehrle, and C. Cadar, "Parti: A multi-interval theory solver for symbolic execution," in *IEEE/ACM International Conference on Automated Software Engineering (ASE 2018)*, 9 2018, pp. 430–440.

[50] D. Liew, C. Cadar, A. F. Donaldson, and J. R. Stinnett, "Just fuzz it: solving floating-point constraints using coverage-guided fuzzing," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 521–532.

[51] J. Choi, J. Jang, C. Han, and S. K. Cha, "Grey-box concolic testing on binary code," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 736–747.

[52] X. Li, Y. Liang, H. Qian, Y.-Q. Hu, L. Bu, Y. Yu, X. Chen, and X. Li, "Symbolic execution of complex program driven by machine learning based constraint solving," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 554–559.

[53] J. He, G. Sivanrupan, P. Tsankov, and M. Vechev, "Learning to explore paths for symbolic execution," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2526–2540.

[54] H. Tu, S. Lee, Y. Li, P. Chen, L. Jiang, and M. Böhme, "Cottontail: Large Language Model-Driven Concolic Execution for Highly Structured Test Input Generation," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2026, pp. 1–19.

[55] N. Ruaro, K. Zeng, L. Dresel, M. Polino, T. Bao, A. Continella, S. Zanero, C. Kruegel, and G. Vigna, "Syml: Guiding symbolic execution toward vulnerable states through pattern learning," in *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, 2021, pp. 456–468.

[56] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, pp. 85–105, 2023.

[57] G. Ryan, S. Jain, M. Shang, S. Wang, X. Ma, M. K. Ramanathan, and B. Ray, "Code-aware prompting: A study of coverage-guided test generation in regression setting using llm," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 951–971, 2024.

[58] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[59] Z. Luo, Q. Du, Y. Wang, A. Roychoudhury, and Y. Jiang, "Enhancing protocol fuzzing via diverse seed corpus generation," *IEEE Transactions on Software Engineering*, 2025.

[60] Y. Oliinyk, M. Scott, R. Tsang, C. Fang, H. Homayoun *et al.*, "Fuzzing BusyBox: Leveraging LLM and crash reuse for embedded bug unearthing," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 883–900.

[61] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *Advances in neural information processing systems*, pp. 27 730–27 744, 2022.

[62] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[63] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[64] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, "SWE-bench: Can language models resolve real-world GitHub issues?" in *The Twelfth International Conference on Learning Representations*, 2024.

[65] Katalon, "The state of software quality report," 2025, available from https://katalon.com/reports/state-quality-2025.

[66] A. Roychoudhury, "Agentic AI for software: thoughts from software engineering community," *arXiv:2508.17343*, Aug 2025.

[67] S. Yao and D. She, " Empc: Effective Path Prioritization for Symbolic Execution with Path Cover ," in *2025 IEEE Symposium on Security and Privacy (SP)*, 2025.

Appendix A. Effect of LLM-Based Test Input Scheduling

TABLE 8: Internal line coverage for polyglot benchmarks with different scheduling strategies.

Subject	ConcoLLMic	ConcoLLMic^{DFS}	ConcoLLMic^{Random}
`ultrajson`	2093	1935	1616
`jansi`	986	930	826
`py4j`	1113	1148	746
`protobuf-go`	4060	3924	3994
Average	2063	1985	1796

To assess the effect of the LLM-based test input scheduling strategy, we conduct a preliminary ablation study by creating two ConcoLLMic variants: ConcoLLMic^{DFS} and ConcoLLMic^{Random}. These variants replace the LLM-based selection with classical *search heuristics*, selecting the next test input for exploration either as the most recent or randomly. The final internal line coverage achieved by different variants is shown in Table 8. From the results, we can see that (i) ConcoLLMic's LLM-based selection strategy is competitive with the best performance of traditional search heuristics (DFS and Random), and (ii) according to Welch's t-test, the average p-value for comparing ConcoLLMic against ConcoLLMic^{DFS} and ConcoLLMic^{Random} is 0.63 and 0.42, respectively. This suggests their relative difference is far from being statistically significant, and ConcoLLMic's LLM-based selection strategy does not contribute to its overall effectiveness. Since search strategy is a long-standing problem in DSE and is not our main focus, we leave finding optimal selection strategies for agentic concolic execution as future work.

Appendix B. Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

The paper develops a novel approach that leverages large language model (LLM) agents to address two key challenges of concolic execution: program environment modeling and path constraint solving. The authors implement this approach in a tool called CONCOLLMIC and demonstrate its effectiveness through an empirical evaluation on different types of programs.

B.2. Scientific Contributions

- Addresses a Long-Known Issue
- Provides a Valuable Step Forward in an Established Field
- Creates a New Tool to Enable Future Science
- Identifies an Impactful Vulnerability
- Establishes a New Research Direction

B.3. Reasons for Acceptance

1) The paper leverages LLM agents to address long-known issues in concolic execution (i.e., program environment modeling and path constraint solving) and therefore provides a valuable step forward in this area.
2) The authors create a new tool called ConcoLLMic that enables future science. The author has demonstrated the effectiveness of ConcoLLMic with an extensive evaluation.
3) ConcoLLMic identifies a non-trivial number of new vulnerabilities on real-world software, further showcasing its practical usefulness.
4) Given its novelty and timeliness, the paper could help establish a new research direction on integrating LLM agents into program analysis workflows.

B.4. Noteworthy Concerns

1) The set of programs used for evaluation seems limited, as they are small and may not be drawn from a standard benchmark suite. This raises questions about the generalizability and broader applicability of the proposed approach.
2) The paper lacks ablation studies to illustrate the trade-off between high-level semantic reasoning and low-level precision.
3) The reviewers also expressed concerns about the reproducibility of the experimental results, given that LLMs are non-deterministic.

Appendix C. Response to the Meta-Review

We thank anonymous reviewers for their constructive feedback and the shepherd for the meta-review. We acknowledge the noteworthy concerns and provide additional responses as follows.

1) Our C/C++ benchmark selection follows established practices in the field: except for `krep` and `confetti` (which serve the specific purpose of evaluating data leakage effects), all benchmarks are drawn from prior works in symbolic execution and fuzzing research. Specifically, `woff2` is used in SymSan [32] and integrated in OSS-Fuzz [35]; `oggenc (vorbis)` is used in SymSan [32] and KLEE-Pending [33] and integrated in OSS-Fuzz [35]; `bc` is used in KLEE-Pending [33] and Empc [67]; `libmatheval` is used in KLEE-Float [12]; and `libyaml` and `libsoup` are integrated in OSS-Fuzz [35]. More importantly, our benchmark selection was strategically designed to test CONCOLLMIC's capabilities across diverse constraint theories and challenging scenarios, as shown in Table 4 with detailed *Inputs* and *Functionality* descriptions and corresponding explanations in §4.1. We acknowledge the current limitation regarding large program processing due to model context window constraints. We have discussed this concern and provided concrete extension strategies to address this limitation in §5.
2) Our current design introduces high-level semantic constraint reasoning as a novel paradigm, allowing flexible constraint representation rather than being restricted to verbose implementation-level formulations, and demonstrates its feasibility and effectiveness. We acknowledge that systematic exploration of trade-offs between high-level semantic reasoning and low-level precision represents an important direction for future research, as discussed in §5.
3) We acknowledge this concern due to the inherent non-determinism of LLMs. To foster reproducibility and enable future research in this area, we have open-sourced our code at https://github.com/ConcoLLMic/ConcoLLMic and prepared a detailed user guide and documentations at https://ConcoLLMic.github.io. We will continuously maintain and update these resources.