

Large Language Model assisted Hybrid Fuzzing

Ruijie Meng, Gregory J. Duck, and Abhik Roychoudhury

Abstract—Greybox fuzzing is one of the most popular methods for detecting software vulnerabilities, which conducts a biased random search within the program input space. To enhance its effectiveness in achieving deep coverage of program behaviors, greybox fuzzing is often combined with concolic execution, which performs a path-sensitive search over the domain of program inputs. In hybrid fuzzing, conventional greybox fuzzing is followed by concolic execution in an iterative loop, where reachability roadblocks encountered by greybox fuzzing are tackled by concolic execution. However, such hybrid fuzzing still suffers from difficulties conventionally faced by concolic execution, such as the need for environment modeling and system call support. In this work, we explore the potential of developing “smart” concolic execution empowered by Large Language Models (LLMs), leveraging their knowledge of code semantics during constraint computing and solving. When coverage-based greybox fuzzing reaches a roadblock in terms of reaching certain branches, we conduct a slicing on the execution trace and suggest modifications of the input to reach the relevant branches. The LLM is used as a solver to generate the modified input to reach the desired branches. Compared with state-of-the-art hybrid fuzzers COFUZZ, INTRIGUER, and QSYM, our LLM-based hybrid fuzzer HYLLFUZZ (pronounced “hill fuzz”) covers 31.43%, 44.56%, and 59.48% more code branches, respectively. Furthermore, the LLM-based concolic execution in HYLLFUZZ takes a time that is 3–19 times faster than the concolic execution running in existing hybrid fuzzing tools. In extensively tested real-world subjects, HYLLFUZZ exposed seven previously unknown bugs. This experience shows that LLMs can be effectively inserted into the iterative loop of hybrid fuzzers to efficiently expose more program behaviors.

Index Terms—Hybrid Fuzzing, Concolic Execution, Large Language Models.

I. INTRODUCTION

Greybox fuzzing [1], [2] is one of the most widely used techniques today to detect security vulnerabilities in software systems. At its core, greybox fuzzing performs an adaptive and biased random search over the input space to find crashing inputs. It conducts lightweight compile-time instrumentation on selected program elements (e.g., branches), and at runtime records which instrumented elements are exercised for a given input. This runtime feedback then guides the input mutation to generate new ones and explore more program behaviors.

Although greybox fuzzing is highly effective, it often gets stuck at complex branches that are rarely evaluated as true, such as those guarded by magic bytes (e.g., `if(input == "\uDB16\uDC06")`). In such cases, concolic execution is more effective in generating inputs to cover the branches. Concolic execution [3], [4], [5] searches the

domain of inputs like greybox fuzzing, except that navigation is achieved by mutating path constraints instead of mutating inputs. A concolic execution engine executes the program with a concrete input while symbolically emulating the program along the same execution path to compute path constraints. By negating these path constraints, it transitions to alternative branches. A constraint solver is then used to solve the negated constraints and generate concrete inputs that drive the execution along new paths. However, its effectiveness often comes at the cost of efficiency [6], [7].

Hybrid fuzzing [7] combines greybox fuzzing with concolic execution to leverage the strengths of both techniques and maximize code coverage. It was first introduced in DRILLER [8] and has subsequently been followed by many testing engines [9], [10], [11]. In hybrid fuzzing, greybox fuzzing proceeds until it encounters roadblocks in unraveling a branch, which is evaluated to be almost always true or false. This roadblock branch captures the frontier of the greybox fuzzing search. Concolic execution is then used to overcome this frontier. Once the roadblock is overcome, greybox fuzzing can take over again until another roadblock is encountered.

Unfortunately, hybrid fuzzing remains limited due to the underlying concolic execution. Traditional concolic execution *mechanically* computes constraints and solves them for satisfiability, without knowledge of the program semantics. This leads to several challenges. For example, real-world constructs, such as external libraries and system calls, require explicit modeling to allow further analysis, which may require years of manual effort [12], [13], [14]. Moreover, even when constraints are satisfiable, traditional solvers (e.g., Z3 [15]) cannot prioritize solutions that also conform to the required input structure, potentially generating a number of syntactically invalid inputs. Compounding the problem, program paths in programming languages (e.g. C/C++ and Java) must be symbolically translated as logic constraints that are solvable for solvers. However, both constraint computing and constraint solving are time-consuming [9], [10], [5].

Recent advances in Large Language Models (LLMs) open up opportunities for developing *smart* concolic execution to address these challenges. Similar to traditional constraint solvers, LLMs have shown the capability of analyzing execution paths and then generating inputs [16], [17]. However, unlike traditional solvers, LLMs have a knowledge of the program semantics. LLMs trained on terabytes of data corpus can understand external libraries [18], eliminating the need for manual modeling, and can infer code functionalities [19], [20], allowing the generation of syntactically valid inputs. Furthermore, LLMs are more adaptive and capable of directly reasoning in high-level programming languages. This allows the LLM-based approach to bypass the translation of source code to logic formulas, thus accelerating the analysis process.

R. Meng is with the National University of Singapore, Singapore. E-mail: ruijie_meng@nus.edu.sg.

G. Duck is with the National University of Singapore, Singapore. E-mail: gregory@comp.nus.edu.sg.

A. Roychoudhury is with the National University of Singapore, Singapore. E-mail: abhik@nus.edu.sg.

Building on this, we propose HYLLFUZZ, a hybrid fuzzer that integrates LLM-based concolic execution. Specifically, when greybox fuzzing reaches a coverage plateau, where it has difficulty covering new program paths, HYLLFUZZ identifies the roadblocks to overcome. The subsequent steps align with the principles of traditional concolic execution. HYLLFUZZ executes the input that reaches the branch condition, but fails to satisfy it. Meanwhile, HYLLFUZZ constructs a dynamic slice of the relevant execution path. Instead of translating the execution path into logic formulas, HYLLFUZZ preserves the path as-is in the form of a source code slice. The LLM is then prompted to modify the input based on the sliced code path and generate new inputs to overcome roadblocks. These LLM-generated inputs are added to the seed corpus, where they are retrieved and evaluated by greybox fuzzing. Whenever greybox fuzzing hits a coverage plateau again, this process is repeated.

The embodiment of our approach in the form of HYLLFUZZ, shows strong promise in delivering *effectiveness*, *efficiency*, and *usability*. To the best of our knowledge, previous hybrid fuzzers have not considered all three of these important dimensions. In terms of effectiveness, HYLLFUZZ achieves 31–59% more branch coverage on average compared to state-of-the-art hybrid fuzzers. In terms of efficiency (median time), each concolic execution run of HYLLFUZZ consumes 3–19 times less time compared to those of state-of-the-art hybrid fuzzers. Finally, in terms of usability, HYLLFUZZ by definition does not involve any symbolic modeling of instructions or building hand-crafted environment models, thereby naturally lowering the barrier to entry.

In summary, we make the following main contributions:

- We propose an LLM-based concolic execution engine, with knowledge of the code semantics during input generation. This approach addresses the problems that affect traditional concolic execution engines.
- We implemented our approach in a novel hybrid fuzzer HYLLFUZZ, which combines greybox fuzzing with LLM-based concolic execution invoked whenever a roadblock is encountered. Our tool is publicly available at: <https://github.com/mengrj/hyllfuzz>
- We evaluated HYLLFUZZ on several widely used benchmarks, and compared it with state-of-the-art hybrid fuzzers. The results demonstrate that HYLLFUZZ delivers significantly greater effectiveness, efficiency, and usability.

II. MOTIVATING EXAMPLE

To motivate our approach, we use the `libxml2` example to demonstrate how LLM-based concolic execution is integrated into hybrid fuzzing to address the challenges. `libxml2` [21] is a widely used XML parser and also a common subject for evaluating fuzzing performance [22], [23], [24]. The code from `parser.c` in Fig. 1 shows how `libxml2` parses an XML comment. The magic values (line 25) and nested conditions (lines 10–12) are roadblocks for greybox fuzzing, and hybrid fuzzing thus invokes concolic execution to solve them.

Let us first take a look at how traditional hybrid fuzzing works. To apply concolic execution to the real-world code, traditional engines must abstract any system calls and external

```

1 /**
2  * Parse an XML (SGML) comment
3  * Comment::='<!--'((Ch-'-')|('-(Ch-'-')))*-->'
4  */
5 void xmlParseComment(xmlParserCtxtPtr ctxt){
6     ...
7     do{
8         // Skip 20+ path conditions
9         printf("%c", *in);
10        if (*in == '-') {
11            if (in[1] == '-'){
12                if (in[2] == '>'){
13                    return;
14                } in++;
15            } goto get_more;
16        }
17    } while (((*in>=0x20) && (*in<=0x7F)) || (*in==0x09
18        ) || (*in==0x0a));
19    ...
20 }
21 void xmlParseContentInternal(xmlParserCtxtPtr ctxt){
22     const xmlchartr *cur = ctxt->input->cur;
23     ...
24     /* This case: a comment */
25     if ((*cur == '<') && (NXT(1) == '!') && (NXT(2) ==
26         '-') && (NXT(3) == '-')){
27         xmlParseComment(ctxt);
28     }
29 }

```

Fig. 1: Code fragment extracted from `parser.c` in `libxml2`.

libraries (e.g., `printf` in Fig. 1) behind models, which allows symbolic reasoning over execution paths. These models are typically handwritten. For example, KLEE [4] includes over 2500 lines of C code to support only 40 system calls, although the Linux kernel has more than 300 system calls. Writing these models is usually labor-intensive, even taking multiple person-years in some cases [12].

During fuzzing, greybox fuzzing generates inputs that reach complex branch conditions (e.g., line 25 in Fig. 1) but fail to satisfy them, as they are always evaluated to false. In such cases, concolic execution is invoked. Starting from an input generated by greybox fuzzing, concolic execution executes it while emulating the code path to compute symbolic constraints, such as `(= (select input i) 60)` for `(*cur == '<')`. However, symbolic emulation can introduce significant time overhead, such as being several orders of magnitude slower than native execution [5], [10], [9]. The condition on line 25 checks if the string represents the opening delimiter (`<!--`) of an XML comment, which involves four separate path constraints. Since concolic execution can only negate and solve one constraint per round, covering the full branch requires multiple rounds. Furthermore, a single round of constraint solving may take up to 500 seconds [10].

In addition, constraint solvers solve constraints just for satisfiability, without considering syntactic validity. When solving the constraints for line 17, solvers prioritize the value `0x20`, thereby generating syntactically invalid inputs that lack the closing delimiter (`-->`). Such invalid inputs are immediately rejected by the parser and thus fail to cover deeper program behaviors. Although the additional path conditions (lines 10–12) can eventually guide the solver toward generating valid inputs, doing so requires satisfying more than 20 path condi-

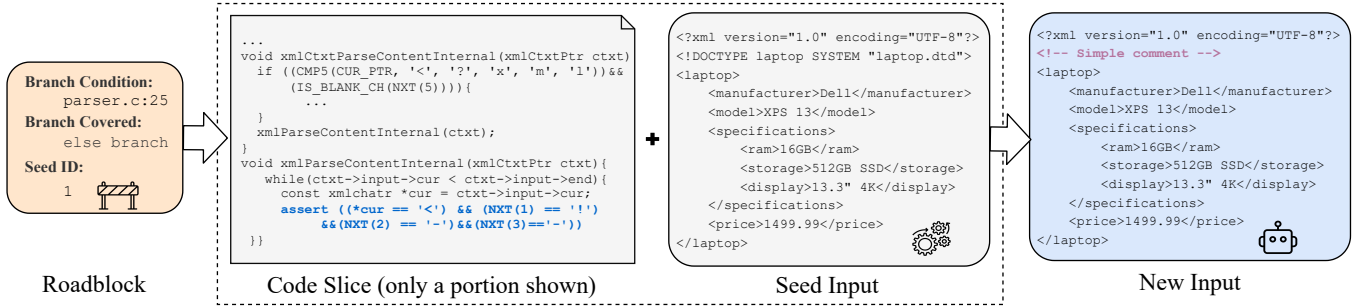


Fig. 2: Workflow of solving the roadblock at `parser.c:25` based on a seed input (selected from the seed corpus of the greybox fuzzer) and the constraints related to the roadblock (sliced from source code) to generate a new input by the LLM.

tions (as shown in Fig. 1). This further reduces the efficiency of concolic execution. In our experiments, traditional concolic execution failed to generate a syntactically valid XML comment for input. In the best case, the engines only generated inputs with a partial opening delimiter (e.g., `<!`), which are unable to explore subsequent program behaviors.

In this paper, we propose a novel hybrid fuzzer HYLLFUZZ, which integrates LLM-based concolic execution to address the above problems. The workflow of HYLLFUZZ to solve the roadblock in `parser.c:25` is shown in Fig. 2. HYLLFUZZ relies on LLMs to directly reason about the underlying functionalities of external constructs, without requiring manual modeling. During fuzzing, when the greybox fuzzer reaches a coverage plateau, HYLLFUZZ activates its LLM-based concolic execution. It first identifies the roadblock condition (`parser.c:25`) and determines the already covered branch (`else branch`). It then finds the corresponding input that reaches this condition, denoted ID:1 by the greybox fuzzer. All of this information forms *Roadblock*.

HYLLFUZZ next performs a concrete execution using the selected input and collects the corresponding execution trace. It performs backward code slicing along the trace. Specifically, it expresses the target condition as an assertion and then slices the relevant code that affects the variables in this assertion (i.e., `cur` and `NXT`) through the control and data flow. This *code slice* represents the path constraints. Since LLMs can directly reason over source code written in programming languages, HYLLFUZZ just keeps them as-is, and there is no need for constraint translation. Using the code slice and the seed input, the LLM is prompted to generate a *new input* to satisfy the constraints, which is then evaluated by the greybox fuzzer. In our experiments, for a single execution, our code slicing combined with LLM-based solving is several times faster than traditional constraint computing and solving.

HYLLFUZZ treats the entire condition, including four constraints, as a single solving target. Furthermore, from context information, such as function names (e.g., `xmlParseComment`) and code comments (e.g., `Parse an XML (SGML) comment`), LLMs infer that the code is parsing the opening delimiter of an XML comment. It then automatically appends the closing delimiter, generating a syntactically valid input that naturally covers subsequent branches (e.g., lines 10–12). In contrast,

traditional solvers are limited to modifying only symbolized data fields. This capability further enhances the effectiveness of HYLLFUZZ. In `libxml2`, HYLLFUZZ can cover twice as many code branches as state-of-the-art hybrid fuzzers.

III. LLM-ASSISTED HYBRID FUZZING

A. Overview

HYLLFUZZ is a hybrid fuzzer that combines greybox fuzzing with LLM-based concolic execution to maximize code coverage. It is designed as a plug-in that can seamlessly integrate with greybox fuzzers, without requiring any modifications to their source code. The overall workflow of HYLLFUZZ is illustrated in Fig. 3. The process is initiated by the greybox fuzzer, which takes in as input the program under test and initial input seeds provided by the user. The greybox fuzzer iteratively selects inputs, generates new inputs via mutation, and saves “interesting” inputs (that cover new branches) into the seed corpus for further examination.

Running in parallel, HYLLFUZZ continuously analyzes the code covered by the seed corpus and maintains a report of uncovered branches (Sec. III-B). When the greybox fuzzer struggles to cover new branches within a given time, HYLLFUZZ steps in by identifying roadblocks from the maintained coverage report. Each roadblock consists of a partially covered branch condition, the branches already covered for this condition, and an input from the seed corpus that reaches this condition. HYLLFUZZ then simulates traditional concolic execution by running the program with the selected input and generating the relevant path slice from the source code (Sec. III-C). HYLLFUZZ negates the target branch condition and prompts the LLM to generate a new input that covers the previously uncovered branch (Sec. III-D). The newly generated input is added back into the seed corpus, assisting the greybox fuzzer in overcoming the roadblocks. The seed corpus forms a bridge between the greybox fuzzer and the LLM-based concolic execution engine.

B. Code Coverage Analysis

1) *Generating Coverage Report*: The first step of HYLLFUZZ is to analyze branch coverage using inputs from the seed corpus generated by the greybox fuzzer. This aims to identify the branch conditions that (1) are reached by at least

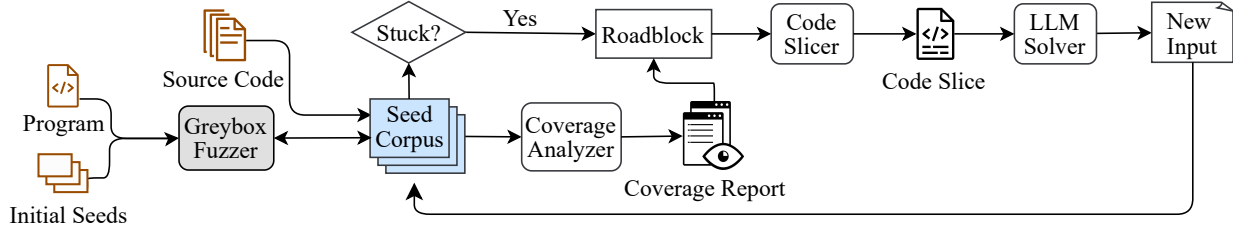


Fig. 3: Overall workflow of HYLLFUZZ including greybox fuzzing and LLM-based concolic execution.

one input, and (2) have at least one branch not covered yet by the greybox fuzzer. Such conditions become potential targets for LLM-based concolic execution. HYLLFUZZ does not rely on the coverage bitmap recorded by the greybox fuzzer, as this bitmap only provides an approximate measure of branch coverage and is known to suffer from collisions [25]. Instead, HYLLFUZZ maintains an independent branch coverage report by directly analyzing inputs stored in the seed corpus. This process is illustrated in Algorithm 1.

HYLLFUZZ works at the source code level. It first builds the Abstract Syntax Tree (AST) for the program (line 3). Next, HYLLFUZZ retrieves the inputs from the seed corpus S of the greybox fuzzer and executes each to collect the corresponding execution trace (line 5). Here, the trace is represented as a sequence of code statements arranged in execution order, where each statement is labeled by its location in source code.

For each statement in the trace, HYLLFUZZ checks if it is a conditional statement (e.g., if- or switch- statement) (line 7). If so, HYLLFUZZ further determines which branches of this condition are covered by the trace (line 8). Simultaneously, HYLLFUZZ computes the total branch number for this condition, which is used to determine whether all branches are covered by the full inputs from S . This information is then recorded in the coverage report \mathcal{R} (line 9). HYLLFUZZ also tracks the corresponding seed in \mathcal{R} for concolic execution. Furthermore, HYLLFUZZ maintains partially covered conditions in a separate structure \mathcal{I} (lines 10–12), which supports the selection of roadblocks in the next phase. It is worth noting that, similar to traditional concolic execution, HYLLFUZZ only tracks conditions that are actually reached by seeds. Any condition that has not been reached yet is not a target for any concolic execution.

2) *Selecting a Roadblock*: HYLLFUZZ continuously monitors whether the greybox fuzzer fails to cover new branches within the given time. To do this, HYLLFUZZ checks the time elapsed since the greybox fuzzer found the last new code path. If the duration exceeds the allotted threshold, the greybox fuzzer is considered to have entered a coverage plateau. At this point, HYLLFUZZ iteratively performs a code coverage analysis and identifies roadblocks from the coverage report \mathcal{R} and partially covered conditions \mathcal{I} .

To ensure fair selection, HYLLFUZZ arranges \mathcal{I} as *priority queue*, assigning the highest priority to newly added conditions. Once a condition is selected, its priority is reduced to avoid starvation. However, if solving a condition successfully generates a new input that increases the code coverage, as validated by the greybox fuzzer (illustrated in Sec. III-D), its

Algorithm 1: Analysis of Coverage and Roadblocks

Input : Program \mathcal{P} , source code \mathcal{C} , seed corpus S
Globals: Coverage report \mathcal{R} , partially covered conditions \mathcal{I}
Output: Roadblock \mathcal{B} to be solved by concolic execution

```

1  $\mathcal{R} \leftarrow \emptyset; \mathcal{I} \leftarrow \emptyset$ 
2 func AnalyzeCodeCoverage( $\mathcal{P}, \mathcal{C}, S$ ):
3    $ast \leftarrow \text{BuildAST}(\mathcal{C})$ 
4   for  $seed \in S$  do
5      $trace \leftarrow \text{execute}(\mathcal{P}, seed)$ 
6     foreach  $stmt \in trace$  do
7       if  $\neg \text{isCond}(stmt, ast)$  then continue
8        $branchCovd \leftarrow \text{AnalyzeBrn}(stmt, ast, trace)$ 
9        $\mathcal{R} \leftarrow \mathcal{R} \cup \{(stmt, branchCovd, seed)\}$ 
10      if  $\text{isFullyCovered}(stmt, \mathcal{R})$  then
11         $\mathcal{I} \leftarrow \mathcal{I} \setminus \{stmt\}$ 
12      else if  $stmt \notin \mathcal{I}$  then  $\mathcal{I} \leftarrow \mathcal{I} \cup \{stmt\}$ 
13 func RetrieveRoadblock( $\mathcal{P}, \mathcal{C}, S$ ):
14   AnalyzeCodeCoverage( $\mathcal{P}, \mathcal{C}, S$ )
15    $condStmt \leftarrow \text{pop}(\mathcal{I})$ 
16    $branchCovd \leftarrow \text{GetBrnCovered}(\mathcal{R}, condStmt)$ 
17    $seed \leftarrow \text{GetSeedForBrn}(\mathcal{R}, condStmt, branchCovd)$ 
18    $\mathcal{I} \leftarrow \text{DecreasePriority}(\mathcal{I}, condStmt)$ 
19   return  $\mathcal{B}(condStmt, branchCovd, seed)$ 

```

priority is increased. This helps prioritize conditions that result in more coverage. Finally, once all branches of a condition are covered, it is removed from \mathcal{I} , since it no longer contains roadblock branches (line 11).

After selecting a partially covered condition $condStmt$ from \mathcal{I} (line 15), HYLLFUZZ determines which of its branches has been covered ($branchCovd$) (line 16). For example, for an if-statement, it indicates which branch, then-branch or else-branch, is covered. HYLLFUZZ then invokes concolic execution by negating the branch condition to explore previously uncovered branches. It also retrieves the corresponding input for the concrete execution (line 17). These three elements together define a *roadblock*: (1) a partially covered condition ($condStmt$), (2) the branches covered ($branchCovd$), and (3) an input ($seed$) that reaches this condition (line 19).

C. Code Slice Generation

Once a roadblock is selected, the next step is to generate a representation of the relevant path constraints. Since HYLLFUZZ prompts the LLM for input generation, the path can be represented in the original programming language (e.g., C, C++, or Java), which the LLM can directly reason over. Our

Algorithm 2: Dynamic Code Slice Generation

Input: Program \mathcal{P} , source code \mathcal{C} , roadblock \mathcal{B}
Output: Dynamic code slice (path constraints)

```

1 func BuildSlice( $\mathcal{P}$ ,  $\mathcal{C}$ ,  $\mathcal{B}$ ):
2    $\langle condStmt, branchCovd, seed \rangle \leftarrow \mathcal{B}$ 
3    $ast \leftarrow \text{BuildAST}(\mathcal{C})$ 
4    $\langle s_1, \dots, s_n, condStmt, \dots \rangle \leftarrow \text{execute}(\mathcal{P}, seed)$ 
5    $stmts \leftarrow \{\text{assert}(!branchCovd)\}$ 
6    $vars \leftarrow \text{getVars}(condStmt)$ 
7   for  $i \in n..1$  do
8     if  $s_i$  modifies any variable  $v \in vars$ , or
9       any condition from  $s_i$  reads any  $v \in vars$ 
10    then
11       $stmts \leftarrow stmts \cup \{s_i\}$ 
12       $vars \leftarrow vars \cup \text{getDependencies}(v, s_i)$ 
13   $slice \leftarrow \emptyset$ 
14  for  $s \in stmts$  do
15     $slice \leftarrow slice \cup \{s\} \cup \text{getCtx}(ast, s)$ 
16  return FlattenSlice( $slice$ )

```

approach differs from traditional concolic execution engines, which translate the path into a new representation (e.g., an SMT formula) suitable for solving.

HYLLFUZZ designs a modified *dynamic back slicing* algorithm over a concrete execution trace (shown in Algorithm 2). The algorithm takes the roadblock $\mathcal{B} \langle condStmt, branchCovd, seed \rangle$ obtained from the previous step. It constructs the AST for the source code (line 3) in advance. It performs a concrete execution using the selected *seed* to get the execution trace that executes *condStmt* at least once (line 4). The algorithm then generates a modified dynamic slice along the concrete execution trace as path constraints, suitable for LLM prompting.

It then initializes a set of statements of interest *stmts* (line 5) and variables of interest *vars* (line 6). The original *condStmt* is negated by replacing it with the assertion: `assert(!branchCovd)`. This assertion serves as the *target* for the LLM-based concolic execution, which must be held in order to cover new branches. Specifically, given (if (*c*) then ... else ...), then:

- `assert(!c)` is generated if the then-branch is covered;
- `assert(c)` is generated if the else-branch is covered.

For (switch (*x*) { case v_1 : ... case v_n : ... default: ... }), the assertion is generated as:

$$\text{assert}(x = w_1 \ || \ \dots \ || \ x = w_m \ || \ dflt)$$

where (1) the subset $\{w_1, \dots, w_m\} \subseteq \{v_1, \dots, v_n\}$ corresponds to the non-covered case-branches from the switch-statement, and (2) *dflt* will be the expression $(x \neq v_1 \ \&\& \ \dots \ \&\& \ x \neq v_n)$ if the default-branch both exists and is non-covered, or *dflt* will be *false* (unused) otherwise.

Once these are built, the modified dynamic slice is constructed. It iterates over the execution trace in reverse order, and collects all statements *stmts* that *condStmt* is (transitively) data dependent (lines 7–12). It then accumulates the *slice* from the data-dependent statements *stmts* (lines 13–15). This essentially accumulates fragments of the original source file(s) into a coherent slice. For each statement $s \in stmts$, it

You are an expert in concolic execution, working with a variant of this technique that generates new inputs based on the code sliced from the program and a given input.	①
I will provide the sliced code along with an input. The input may be valid or malformed. Your task is to generate a new valid input that satisfies the assertion in the sliced code. The assertion is reached during the execution of the provided input, but is not satisfied.	②
Follow these steps to generate a new input: 1. Review the sliced code and the provided input. 2. Modify specific fields of the input as needed, or completely havoc the whole input to satisfy the assertion. 3. Ensure the new input satisfies the assertion and is syntactically valid.	③
The following are the sliced code and the provided input. Generate a new input based on the sliced code to satisfy the assertion. Your answer should be the generated input.	④
Please keep the same format as the given input. Write your answer in the consistent format: ``my_answer: <Put your answer here>``.	⑤
The sliced code is as follows: ``{sliced_code}``	⑥
The provided input is as follows: ``{seed_content}``	⑦

Fig. 4: Prompt template for generating new input based on the given input and relevant code slice.

accumulates *s* as well as any *context* of *s*. Broadly, the context includes all elements of the AST:

- 1) control-flow structure enclosing *s*, including if-, while-, for-, and switch- statements;
- 2) enclosing function definition, including types/parameters;
- 3) declarations (local and global) for each variable used by *s*;
- 4) syntax necessary to parse any of the above, including braces ({}) and semicolons (;);
- 5) formatting (whitespace) and comments.

Finally, the collected set of file ranges in *flattened* into a string suitable for prompt generation, preserving the original file formatting and ordering (line 16).

D. LLM-based Input Generation

After collecting the relevant code slice, the LLM is prompted to generate a new syntactically valid input aimed at covering the previously uncovered branch for the roadblock condition. The template of the prompt is shown in Fig. 4, which consists of several parts (①–⑦).

The parts ①, ②, and ③ together form the system prompt. Part ① defines the LLM’s persona as an expert in concolic execution, working with a variant of this technique. Based on our observations, the LLMs have sufficient knowledge regarding concolic execution. Since HYLLFUZZ simulates traditional concolic execution, Part ① can help the LLM understand the overall task. Part ② gives a more detailed description of the task: generating a new syntactically valid input that satisfies the assertion in the sliced code. The whole task of HYLLFUZZ is relatively complex and includes several steps. To maintain the effectiveness of the LLM, Part ③ includes detailed instructions that break down the task into simpler subtasks, guiding the LLM step by step. While generating a

new input, the LLM can slightly modify or havoc the given input, but ensure that it is syntactically valid.

The parts ④–⑦ are included in the user prompt. Part ④ describes the task and highlights that the answer is a concrete input instead of other formats (e.g., one program that generates inputs). This part also prevents the LLM from printing intermediate results, helping reduce token usage. Part ⑤ further specifies the expected output format, which simplifies response extraction and ensures that the generated inputs are immediately usable. Parts ⑥ and ⑦ include the code slice and the given input, respectively, each enclosed within delimiters for clear separation. The input given is one that reaches the condition but fails to cover the target roadblock branches. Our insight is that slightly modifying a few bytes can often solve the roadblock, which can be simpler than generating a new input from scratch. However, if slight modifications cannot achieve the goal, causing havoc to the input becomes necessary, although this is a secondary option. Furthermore, the input provided serves as an example to illustrate the correct input format (i.e., “one-shot” prompt) and offers more context to help the LLM reason through the task.

After receiving the input generated by the LLM, HYLLFUZZ adds it to the seed corpus, where it can be automatically retrieved and executed by the greybox fuzzer. The greybox fuzzer is configured as a distributed fuzzer, with a key feature being its ability to analyze seeds from one or more collaborators. Before the generated input is added to the seed corpus, there is no additional validation; instead, this task is delegated to the greybox fuzzer. The fuzzer executes the new input and saves it in the seed corpus if it covers new branches.

IV. EVALUATION

To evaluate the efficacy of HYLLFUZZ, we seek to answer the following research questions:

RQ.1 (Branch Coverage and Bug Finding) Can HYLLFUZZ cover more code branches compared to the baselines? Can HYLLFUZZ find additional bugs over the baselines?

RQ.2 (Time Usage of Concolic Execution) How much time does LLM-based concolic execution in HYLLFUZZ take to generate a single input? Is it faster than the baselines?

RQ.3 (Coverage Improvement Analysis) How does HYLLFUZZ achieve higher code coverage than the baselines? Does the code coverage improvement stem from solving a larger number of roadblocks? Do the efficiency and the solving capability of LLM-based concolic execution contribute to the code coverage improvement? Does the hybrid architecture of LLM-based concolic execution contribute to the improvement?

RQ.4 (Unsolved Roadblock Analysis) How does HYLLFUZZ fail to solve certain roadblocks? How many roadblocks are selected for solving yet remain unsolved in HYLLFUZZ? Does the number of solving attempts affect the results? Are there any roadblocks that are solved by the baselines but remain unsolved by HYLLFUZZ?

RQ.5 (Case Study) Can HYLLFUZZ expose the bug in Jenkins implemented in a different programming lan-

TABLE I: Detailed information about benchmark subjects used in our experiments for RQ.1–RQ.4.

Subject	#LOC	Version	Description
as-new	96.78k	v2.43	Assemble code into object files
bc	11.32k	v1.08.2	Perform mathematical calculations
calc	62.03k	v2.16.0.0	Perform advanced calculations
cflow	37.39k	v1.7.90	Generate call graphs for C code
cJSON	17.23k	v1.7.18	Parse JSON files
curl	188.4k	v8.13.0	Transfer data with URL syntax
cxxfilt	96.78k	v2.43	Demangle C++ symbol names
infotocap	100.48k	v6.4	Convert terminfo into termcap
jq	34.11k	a8f27cc	Parse JSON files
libxml2	201.95k	v2.13.4	Parse XML files
Lua	35.22k	v5.4.7	Interpret Lua scripts
MuJS	16.77k	v1.3.5	Interpret Javascript files
php	2123.39k	v8.4.6	Interpret PHP files
SQLite3	593.90k	3.48.0	Manage SQL database files

guage released from the 2024 DARPA and ARPA-H’s Artificial Intelligence Cyber Challenge?

A. Experimental Setup

1) *Comparison Baselines:* As baseline tools, we selected three state-of-the-art hybrid fuzzers for comparison: QSYM [9], INTRIGUER [10], and COFUZZ [7]. DRILLER [8] is the first hybrid fuzzer that combines greybox fuzzing with concolic execution; however, it does not scale to large real-world benchmark subjects, as reported in previous works [9], [7]. Among existing hybrid fuzzers, only QSYM and INTRIGUER align with HYLLFUZZ in their goal of improving concolic execution integrated in hybrid fuzzing, although they do so by incorporating fast symbolic emulation and constraint-solving strategies. COFUZZ is the most recent hybrid fuzzer, developed after comprehensively evaluating all previous greybox fuzzers [11], [26], [27], which have also shown lower effectiveness compared to COFUZZ. Each of these hybrid fuzzers integrates its own concolic execution engine with the greybox fuzzer AFL. For a fair comparison, we also integrated AFL into HYLLFUZZ to build a hybrid fuzzer.

2) *Benchmark Subjects:* Table I lists the benchmark subjects used in our evaluation to answer the research questions RQ.1 to RQ.4. Our benchmark comprises 14 widely-used real-world programs, including three parsers (for JSON and XML), two binutils utilities (as-new and cxxfilt), three language interpreters (for Lua, PHP and JavaScript), two transformation tools (for URL and terminfo), one GNU utility (cflow), one database engine (SQLite3), and two mathematical calculators (bc and calc). These subjects involve both string-based and arithmetic path constraints.

For each subject, we report the details on the code lines, the version used in our evaluation, and their primary functionality. These subjects are widely utilized in practice and represent diverse functionalities and codebase sizes. Our benchmark includes all text-based input subjects that are incorporated into popular fuzzing benchmark suites: MAGMA [23], UNIFUZZ [24], and FUZZBENCH [22], as well as all those used in

TABLE II: Average number of branches covered by our HYLLFUZZ, and the baselines COFUZZ, INTRIGUER, and QSYM.

Subject	HYLLFUZZ	Comparison with COFUZZ			Comparison with INTRIGUER			Comparison with QSYM		
		COFUZZ	Improv	SpeedUp	INTRIGUER	Improv	SpeedUp	QSYM	Improv	SpeedUp
as-new	8145.2	6150.0	32.44%	4.85×	5640.2	44.41%	6.70×	4831.9	68.57%	25.71×
bc	1203.7	984.1	22.31%	5.49×	1015.1	18.58%	5.40×	982.3	22.54%	5.49×
calc	7194.5	4724.3	52.29%	18.49×	5328.1	35.03%	5.74×	4795.1	50.04%	17.36×
cflow	1238.2	1167.2	6.08%	11.16×	1194.1	3.69%	6.43×	1152.9	7.40%	14.55×
cJSON	379.9	375.5	1.17%	22.50×	347.4	9.36%	57.60×	343.9	10.47%	80.00×
curl	3789.0	3285.0	15.34%	6.75×	3379.0	12.13%	1.35×	3281.0	15.48%	6.76×
cxxfilt	2448.4	2199.6	11.31%	2.78×	2039.1	20.07%	3.80×	1951.0	25.49%	4.39×
infotocap	1729.0	1545.4	11.88%	2.84×	1474.9	17.23%	3.72×	933.2	85.28%	240.00×
jq	2076.0	2026.2	2.46%	30.64×	2039.7	1.78%	27.17×	2018.6	2.84%	34.29×
libxml2	6811.6	3282.4	107.52%	32.00×	3163.8	115.30%	49.66×	3092.3	120.28%	51.43×
Lua	3529.3	2923.8	20.71%	5.45×	2438.0	44.76%	12.20×	1951.5	80.85%	24.83×
MuJS	3666.0	2454.0	49.39%	7.70×	2078.0	76.42%	14.85×	2096.0	74.90%	14.69×
php	17268.7	8926.5	93.45%	6.26×	5554.2	210.91%	53.33×	5683.5	203.84%	45.00×
SQLite3	19746.0	17369.5	13.68%	1.75×	17300.3	14.14%	1.76×	11990.0	64.69%	96.00×
Average	-	-	31.43%	11.33×	-	44.56%	17.84×	-	59.48%	47.18×

existing hybrid fuzzing works [9], [10], [7]. In addition to these string-heavy programs, we include two widely-used numerical-heavy programs (i.e., `bc` and `calc`) to evaluate the generality of HYLLFUZZ.

3) *Configuration Parameters*: For the subject programs, we adopted the same configuration parameters as those used in previous work [22], [23], [24], [7]. We used the `gpt-4o` model as the large language model, with the temperature set to 0.5 for precise and factual responses. Additionally, to determine if the greybox fuzzer is stuck, we set a time threshold of 90 seconds; if the fuzzer fails to cover new code paths within this period, the concolic execution engine is triggered. This time frame aligns with the default settings used by the baseline tools. Following suggestions from the fuzzing community [6], [28], each experiment was run for 24 hours, and we report the average over 10 runs to mitigate the impact of randomness. All experiments were conducted on an Intel® Xeon® Platinum 8468V CPU with 192 logical cores clocked at 2.70GHz, 512GB of memory, and Ubuntu 22.04.3 LTS.

B. Branch Coverage and Bug Finding (RQ.1)

The primary goal of hybrid fuzzers is to maximize code coverage, making it a key metric for evaluating the performance of hybrid fuzzers. In our evaluation, we measured the branch coverage achieved by HYLLFUZZ and the state-of-the-art hybrid fuzzers using `gcovr`. To ensure a fair comparison, we used the same seeds for each fuzzer. Each fuzzer was run on each subject for 24 hours and repeated 10 times; we report the average data (i.e., mean values) as the final results. Since higher code coverage typically increases the likelihood of exposing bugs [28], we also report the number of bugs discovered during these runs.

1) *Results on Branch Coverage*: Table II shows the final code coverage achieved by HYLLFUZZ compared to three baseline fuzzers. We report the average number (i.e., mean values) of branches covered by each fuzzer in 24 hours across 10 runs. To quantify the improvement of HYLLFUZZ over the

baseline tools, we report the percentage increase in terms of branch coverage in 24 hours (*Improv*), and how much faster HYLLFUZZ can achieve the same branch coverage as the baseline tools (*SpeedUp*).

Overall, HYLLFUZZ achieved more branch coverage than all state-of-the-art hybrid fuzzers on all subjects, with improvements ranging from 1.17% to 210.91%. Compared to COFUZZ, INTRIGUER, and QSYM, HYLLFUZZ covered 31.43%, 44.56%, and 59.48% more code branches on average, respectively. In addition, HYLLFUZZ covered the same number of branches 11.33× faster than COFUZZ, 17.84× faster than INTRIGUER, and 47.18× faster than QSYM.

To account for randomness, we measured the Vargha-Delaney [29] effect size \hat{A}_{12} , which indicates the probability that a random campaign of HYLLFUZZ outperforms a random campaign of comparison tools. In all subjects, $\hat{A}_{12} = 1$ (that is, $\hat{A}_{12} \geq 0.70$), indicating that HYLLFUZZ consistently achieved higher branch coverage in all runs than each baseline. This shows a significant improvement of HYLLFUZZ over baselines in code coverage.

In addition, we measured both the monetary costs and the token usage required to run these experiments. It is worth noting that the monetary cost is relatively low: in our experiments, running a subject for one hour only cost 0.40 USD on average. The required token count was also modest. The average token count per prompt was 1,570, with the maximum observed being 3,020 tokens in `php` containing millions of code lines. We did not encounter cases where the generated code slices exceeded the LLM’s context window limit. These results further demonstrated the practical utility of HYLLFUZZ.

2) *Results on Bug Finding*: Although these subjects have been extensively tested by existing fuzzers, HYLLFUZZ found seven unique and previously unknown bugs in their latest version: two in `cxxfilt`, two in `php`, and three in `calc`. In contrast, the baseline fuzzers only found two (one in `php` and one in `calc`) of these bugs in total, and also did

not find any additional bugs. We reported these bugs to the respective developers, and all have been confirmed. These bugs include null pointer dereference, undefined behavior, and buffer overflows.

Upon closer investigation, we found that the missed bugs were hidden behind complex path constraints. For example, exposing the bugs in `cxxfilt` requires solving deep path constraints—involving 13 and 16 function calls, respectively. The baselines faced a similar difficulty in exposing the bugs in `calc`. Concolic execution engines in the baselines often drop some path constraints for better solving efficiency, but also compromise the performance. We observed that the solving capability of the baselines significantly declined with the increase in the path constraint depth. This observation is consistent with the finding of a recent study [30]. In addition, exposing the `php` bug missed by the baselines requires generating a valid function in the PHP input: `function name() {}`, where each byte corresponds to one branch condition. Generating this segment requires multiple rounds of constraint computing and solving. Traditional concolic execution just managed to generate a partial segment (e.g., `fun`), thereby failing to trigger the bug.

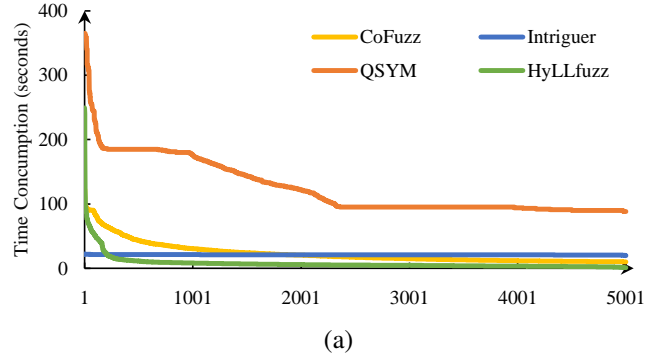
HYLLFUZZ achieves significantly higher code coverage than COFUZZ, INTRIGUER, and QSYM, covering 31.43%, 44.56%, and 59.48% more code on average, respectively. Moreover, HYLLFUZZ achieves the same number of branches 11.33 times, 17.84 times, and 47.18 times faster than these baselines. In extensively tested projects, HYLLFUZZ exposed seven unique and previously unknown bugs, while baselines only found two of them without finding any additional bugs.

C. Time Usage of Concolic Execution (RQ.2)

To evaluate the efficiency of LLM-based concolic execution, we measured the time required for each concolic execution run to generate an input. Specifically, for the baselines INTRIGUER, QSYM, and COFUZZ, we measured the time used for constraint computing alongside concrete execution and constraint solving via SMT solvers to generate an input. Similarly, for HYLLFUZZ, we measured the time used for dynamic code slicing along the concrete execution, and constraint solving using an LLM to generate an input—from the prompt construction to obtaining a concrete input from the LLM.

We randomly sampled 5000 concolic execution runs of HYLLFUZZ and three baselines COFUZZ, INTRIGUER, and QSYM across all subjects, respectively. Fig. 5 (a) shows the time usage of each fuzzer within these sampled runs, while Fig. 5 (b) shows the average time and median time usage. As we can see, compared to all these baselines, HYLLFUZZ substantially reduces time usage, although these baselines adopt optimization strategies.

In this experiment, we reported and compared the wall-clock time used, as it is a standard metric for evaluating fuzzers and is also particularly important for end-users seeking to mitigate security risks. We acknowledge that, due to the use of an LLM, HYLLFUZZ incurs additional GPU resource



HYLLFUZZ	vs COFUZZ	vs INTRIGUER	vs QSYM
Med	Med Speedup	Med Speedup	Med Speedup
4.97	17.39 3.50×	20.95 4.22×	95.00 19.11×
Avg	Avg Speedup	Avg Speedup	Avg Speedup
7.66	23.72 3.10×	20.96 2.74×	125.26 16.35×

(b)

Fig. 5: Wall-clock time used by the baselines COFUZZ, INTRIGUER, QSYM, and our HYLLFUZZ for each run of concolic execution (seconds).

consumption compared to the baselines; however, we believe this cost is justified by the substantial gains in efficacy.

Each concolic execution run in HYLLFUZZ takes 4.97 seconds in the median, which is 3.50 times, 4.22 times, and 19.11 times faster than that of COFUZZ, INTRIGUER, and QSYM, respectively.

D. Coverage Improvement Analysis (RQ.3)

We further conducted an in-depth analysis to understand how HYLLFUZZ achieved a significant improvement in code coverage compared to the baselines. In this analysis, we compared HYLLFUZZ with hybrid fuzzers COFUZZ, INTRIGUER, and QSYM to evaluate the performance of different concolic execution engines. In addition, we included an additional comparison with the grammar-aware fuzzers NAUTILUS [31] and SUPERION [32] to evaluate whether the LLM-based constraint reasoning and solving contribute to the code coverage improvement.

1) *Number of Solved Roadblocks*: We started by analyzing whether the increase in branch coverage correlates with the number of roadblocks solved. In Table III, we report the average number of roadblocks solved per run by HYLLFUZZ and the baselines based on 10 runs of 24 hours. On average, HYLLFUZZ solved 411.4 roadblocks, while the baselines solved 212.4, 181.6, and 144.3 roadblocks, respectively. In certain subjects (e.g., `jq`, `curl`, and `cJSON`), HYLLFUZZ solved relatively few roadblocks, while HYLLFUZZ solved more than 800 roadblocks in subjects `calc`, `libxml2` and `php`. In all subjects, HYLLFUZZ consistently outperformed all baseline fuzzers in terms of the number of roadblocks solved within the same time frame. When comparing the performance

TABLE III: Average number of roadblocks solved by our HYLLFUZZ and baselines COFUZZ, INTRIGUER and QSYM.

Subject	HYLLFUZZ	COFUZZ	INTRIGUER	QSYM
as-new	585.0	298.7	250.4	155.1
bc	200.0	74.5	135.0	70.0
calc	880.1	177.0	361.0	172.0
cflow	100.0	55.4	59.3	26.9
cJSON	84.5	72.9	18.4	0.0
curl	76.3	89.0	36.6	56.7
cxxfilt	193.8	139.7	106.3	104.8
infotocap	309.0	265.0	248.1	165.0
jq	44.0	1.3	4.4	0.1
libxml2	842.0	224.0	158.1	269.0
Lua	624.3	301.1	290.1	112.0
MuJS	758.6	454.8	315.1	378.2
php	837.1	698.3	423.0	511.0
SQLite3	224.6	121.6	136.1	0.0
Average	411.4	212.4	181.6	144.3

of HYLLFUZZ in different subjects, we observe that a larger number of solved roadblocks almost corresponds to a greater increase in code coverage. These results demonstrate a positive correlation between the number of roadblocks solved and the improvement in code coverage.

2) *Contribution of Fast Concolic Execution:* We further analyzed whether the increase in branch coverage is due to the fast concolic execution engine. To this end, we measured the number of inputs generated by the concolic execution engines of HYLLFUZZ and the baselines within 24 hours. On average, HYLLFUZZ generated 2828 inputs, while COFUZZ, INTRIGUER, and QSYM generated 2494, 2154, and 1475 inputs, respectively. HYLLFUZZ significantly outperformed QSYM, generating more than twice as many inputs in the same time frame. However, compared to COFUZZ and INTRIGUER, HYLLFUZZ only achieved a slightly higher input number. This result indicates that fast concolic execution contributes to improved code coverage. However, the performance gain cannot be fully attributed to the fast LLM-based engine of HYLLFUZZ, as COFUZZ and INTRIGUER generated a comparable number of inputs.

3) *Contribution of LLM-based Solving:* We next analyzed whether the solving capability of HYLLFUZZ contributed to increasing code coverage. We examined this from two aspects. First, we used the concolic execution engines of the baselines to solve the roadblocks selected by HYLLFUZZ, and then compared the number of roadblocks solved by different techniques under the same time budget (i.e., 24 hours). This analysis is intended to demonstrate that the performance gains of HYLLFUZZ stem from LLM-based solving rather than from the differences in roadblock selection. Second, we conducted a manual analysis of the branches covered by HYLLFUZZ but missed by the baselines to elucidate the role of LLM-based solving in this process. To this end, we selected two string-based subjects php and jq, where HYLLFUZZ achieved a higher increase in code coverage in php but only a smaller improvement in jq, as well as two arithmetic-based subjects

TABLE IV: Number of roadblocks solved by baselines using HYLLFUZZ-selected roadblocks (*baseline-modified*) and baselines with their original designs (*baseline-original*), and our HYLLFUZZ.

Tool	php	jq	bc	calc
baseline-modified	205.3	9.0	117.0	369.0
baseline-original	698.3	4.4	135.0	361.0
HYLLFUZZ	827.1	44.0	200.0	880.1

```

1 yy523:
2   yych = *++YYCURSOR;
3   if (yych == 'R') goto yy597;
4   if (yych == 'r') goto yy597;
5   goto yy140;
6
7 yy597:
8   ++YYCURSOR;
9   if ((yych = *YYCURSOR) <= '^') {...}
10  else {
11    if (yych <= 'd') { ... }
12    else {
13      if (yych <= 'z') goto yy655; ...
14    }
15  }

```

Fig. 6: Code fragment from php showing input interpretation.

bc and calc.

In Table IV, we report the results of using the concolic execution engines of the baselines to solve the roadblocks selected by HYLLFUZZ. The best-performing baselines could only solve 205.3, 9.0, 117.0, and 369.0 roadblocks in php, jq, bc, and calc, respectively. These results are similar to those obtained using their original designs, in which the best baseline solved 698.3, 4.4, 135.0, and 361.0 roadblocks, respectively. In contrast, our HYLLFUZZ using LLM-based solving solved substantially more roadblocks, whose numbers are 827.1, 44.0, 200.0, and 880.1 in the same subjects, respectively. These results indicate that the improvement of HYLLFUZZ in code coverage stems from its LLM-based solving ability rather than from the use of a different roadlock-selection strategy.

We further manually analyzed the branches covered by HYLLFUZZ but missed by the baselines. Fig. 6 illustrates how php interprets keywords containing the byte ‘R’. In our evaluation, the baselines failed to generate the simple keyword `try`, which is necessary to explore certain program behaviors. After interpreting the input “`tr`”, HYLLFUZZ inferred the code semantics and correctly generated the missing letter “`y`” to solve the condition on line 13, generating a syntactically valid input and achieving global optima. In contrast, the baselines focus only on local optima and prioritize the value “`z`”, generating “`trz`”. This example demonstrates how HYLLFUZZ generates a single keyword effectively. The format php includes more complex keywords like “`return`”, “`break`”, and “`function`”. Without knowledge of the program semantics, the baselines struggled to generate them. Conversely, the jq inputs are JSON files with relatively simple structures; for example, “`{}`” represents key-value pairs and “`[]`” represents arrays. This makes HYLLFUZZ just perform slightly better than the

TABLE V: Average number of branches covered our HYLLFUZZ, and grammar-aware fuzzers NAUTILUS and SUPERION.

Subject	NAUTILUS		SUPERION	
	#Branches	#Branches Improv	#Branches	Improv
libxml2	6811.6	3126.0 117.90%	2503.1	172.13%
Lua	3529.3	2358.7 49.63%	-	-
MuJS	3666.0	2234.7 64.05%	2014.9	81.94%
php	17268.7	13011.5 32.72%	-	-
Average	-	- 66.07%	-	127.04%

baselines in jq. For the arithmetic-based subjects bc and calc, we observed that the concolic execution engines of the baselines failed to solve the deep path constraints. The success rate of constraint solving significantly decreased as the depth of path constraints increased, which is also reported in the recent study [30].

4) *Contribution of LLM-based Deeper Reasoning:* From the previous analysis, we observed that one advantage of LLM-based solving is its awareness of input structures compared to traditional SMT-based solvers, which contributes to the coverage improvement to some extent. This raises the following questions: What is the contribution of the HYLLFUZZ’s deeper reasoning over the path constraints? Can the improvement be achieved solely through the ability to generate syntactically valid inputs?

To answer these questions, we included two state-of-the-art grammar-aware fuzzers NAUTILUS [31] and SUPERION [32] for comparison. Grammar-aware fuzzers are specifically designed to handle structured input formats. Such a comparison can highlight the contribution of the deeper reasoning over path constraints during input generation. We compared the branch coverage achieved by HYLLFUZZ with that achieved by grammar-aware fuzzers. LANGFUZZ [33] is not included, as its source code is not publicly available. Both NAUTILUS and SUPERION rely on input specifications to generate new inputs, and these specifications are manually written. By default, NAUTILUS only supports the javascript, php, lua, and xml inputs, while SUPERION supports the javascript and xml inputs. We also attempted to use LLMs to automate specification generation; however, the current LLMs fail to achieve this. As a result, the grammar-aware fuzzers could only fuzz the subjects libxml2, Lua, MuJS, and php. We ran each fuzzer on each subject for 24 hours and repeated 10 times.

The average number of branches covered by each fuzzer is shown in Table V. As we can see, HYLLFUZZ covers substantially more code branches than both baselines. Specifically, compared to NAUTILUS and SUPERION, HYLLFUZZ covered 66.07% and 127.04% more branches, respectively. These results indicate that to achieve this code coverage improvement, only handling the input structures is not sufficient, and the deeper reasoning over path constraints plays a significant role in this improvement.

5) *Contribution of Hybrid Architecture of LLM-based Concolic Execution:* Concolic execution adopts a hybrid architecture that combines concrete execution with symbolic execution. Our LLM-based concolic execution approach follows this

TABLE VI: Average number of branches covered by both HYLLFUZZ and its ablation tool HF0 without only concrete execution.

Subject	HYLLFUZZ	HF0	Improv
php	17268.7	12656.2	36.44%
jq	2076.0	2023.1	2.61%
bc	1203.7	1031.4	16.71%
calc	7194.5	5087.6	41.41%
Average	-	-	24.29%

same paradigm. However, given that LLMs have the potential of reasoning over concrete execution traces and generating inputs, it is natural to question whether this hybrid architecture remains necessary. In other words, could an LLM combined solely with concrete execution already achieve comparable results?

To investigate this question, we developed an ablation baseline HF0, which uses the LLM directly to reason over concrete execution and then generate new inputs, while keeping all other strategies (e.g., roadblock selection) the same. We evaluated HF0 on the same set of subjects php, jq, bc, and calc as that used in our previous analysis. Each subject was run for 24 hours and repeated 10 times. The average code coverage achieved by both HYLLFUZZ and its ablation HF0 is shown in Table VI. From this result, we could see that the hybrid architecture is necessary, as it significantly improves the code coverage. In addition, HYLLFUZZ also reduces the LLM token usage by approximately $3.5\times$ compared to HF0. Therefore, this hybrid architecture not only improves coverage but also lowers monetary costs.

The code improvement of HYLLFUZZ stems from its ability to solve a larger number of roadblocks compared to the baselines. The efficiency of LLM-based concolic execution, the LLM-based reasoning and constraint-solving capabilities, and the hybrid architecture of LLM-based concolic execution all contribute to this improvement.

E. Unsolved Roadblock Analysis (RQ.4)

In this analysis, we evaluated how HYLLFUZZ failed to solve certain roadblocks, which aims to provide insights about how to effectively use LLM-based concolic execution. Specifically, we analyzed how many roadblocks are selected for solving but remain unsolved by HYLLFUZZ. We further investigated whether the number of solving attempts affects the results. In addition, we evaluated whether there are roadblocks that are solved by the baselines but not solved by HYLLFUZZ. These analyses are conducted on two string-based subjects jq and php, and two algorithm-based subjects bc and calc.

1) *Number of Roadblocks Selected for Solving yet Unsolved by HYLLFUZZ:* During the fuzzing campaign, we tracked how many roadblocks were selected by HYLLFUZZ for solving, and how many roadblocks were unfortunately unsolved by HYLLFUZZ at the end. The corresponding statistics and unsolved rates are reported in Table VII. Overall, HYLLFUZZ

TABLE VII: Number of roadblocks selected for solving (*#Selected*), number of roadblocks finally unsolved by HYLLFUZZ (*#Unsolved*), and the unsolved rate (*UnsolvedRate*).

Tool	php	jq	bc	calc
#Selected	1131.2	72.9	238.2	1091.1
#Unsolved	294.1	28.9	38.2	211.0
UnsolvedRate	26.26%	39.64%	16.04%	19.34%

successfully solved more than 60% of the selected roadblocks, while the unsolved rates range from 16.04% to 39.64% across the evaluated subjects.

We then manually analyzed the reasons why HYLLFUZZ failed to solve these roadblocks. It is surprising to find that this is not due to the limitations of LLM-based concolic execution. Instead, the majority of the roadblocks (up to 80%) left unsolved by HYLLFUZZ are in fact unsatisfiable, as the corresponding branches are controlled by their configurations and program environments rather than by the fuzzed inputs. For example, a branch condition in `calc` checks whether logging information should be printed. Since printing is disabled in the configuration, the flipped branch cannot be satisfied. Similarly, for the branch conditions that check whether the files are successfully opened or whether memory allocation succeeds, their flipped branches are also unsatisfiable, as they depend on the execution environments. In the current design of HYLLFUZZ, roadblocks are selected uniformly for solving without performing any satisfiability analysis, and branches that are inherently unsatisfiable are not deprioritized. A more effective design would consider the feasibility of a roadblock before selection; we leave this direction for future work.

2) *Number of Solving Attempts Required to Solve a Roadblock*: Given the nondeterministic nature of the LLM responses, we further analyzed whether the number of solving attempts affects the success rate of solving. To this end, we counted the number of solved roadblocks under different numbers of solving attempts, and the results are illustrated in Fig. 7. We observe that more than 50% of the roadblocks are solved with a single solving attempt, and that using fewer than five solving attempts is sufficient to solve 94% of all solvable roadblocks. These results indicate that the LLM responses are relatively stable, and that increasing the number of solving attempts up to five can improve the success rate; however, additional attempts beyond five yield little further improvement.

3) *Number of Roadlocks Solved by Baselines but Unsolved by HYLLFUZZ*: We conducted an additional analysis to evaluate whether there are roadblocks that are solved by the baselines but remain unsolved by HYLLFUZZ. To this end, we counted the code branches uniquely covered by HYLLFUZZ and by the baselines. The results are shown in Fig. 8. As expected, HYLLFUZZ uniquely covered substantially more branches, reflecting its significant coverage improvement. In contrast, the three baselines COFUZZ, INTRIGUER, and QSYM also uniquely covered a small number of branches in the subjects: 8 branches in `bc`, 144 branches in `calc`, 0 branches in `jq`, and 556 branches in `php`.

We then manually analyzed why these branches were not

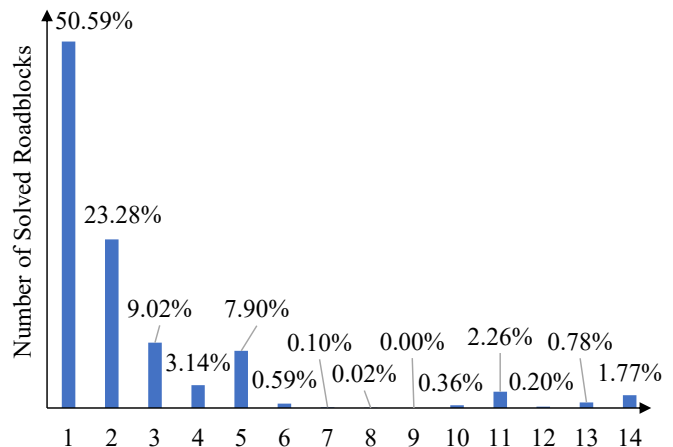


Fig. 7: Number of solved roadblocks (in percentage) under different numbers of solving attempts.

covered by HYLLFUZZ. While we initially expected to reveal limitations of LLM-based concolic execution used in HYLLFUZZ, we found that this issue does not arise from its inability to solve the relevant path constraints. Instead, most of the branches uncovered by HYLLFUZZ occur in `switch-case` conditions, where multiple branches coexist. In such cases, HYLLFUZZ represents the solving target in the `or` form after flipping the conditions, and then randomly selects one path conditions to satisfy. As a result, some branches are missed, because HYLLFUZZ chooses one constraint to satisfy while the baselines solve a different path constraint. This explains why HYLLFUZZ ultimately fails to cover some branches.

HYLLFUZZ failed to solve approximately 16.04% to 39.64% of the selected roadblocks. While increasing the number of solving attempts can slightly improve the success rate, this effect is limited to five attempts. Additionally, a small number of branches were solved by the baselines but missed by HYLLFUZZ. Our manual analysis revealed that these unsolved branches are not due to limitations of LLM-based concolic execution in HYLLFUZZ.

F. Case Study (RQ.5)

In the previous research questions, we evaluated the performance of HYLLFUZZ on C/C++ programs in combination with AFL. HYLLFUZZ has also been adapted for testing programs written in other programming languages and combined with other greybox fuzzers. We conducted a case study on the Jenkins subject [34] written in Java, by integrating the greybox fuzzer JAZZER [35] (designed to fuzz Java programs) into HYLLFUZZ. For this study, we selected the bug manually created by the 2024 DARPA and ARPA-H’s Artificial Intelligence Cyber Challenge (AIXCC) [36], and released after the gpt-4o snapshot. The details are shown in Fig. 9.

The bug occurs on line 32 in Fig. 9. To reach this location, the input must satisfy several conditions. In condition ①, the input data is split by the symbol “\0”, and the resulting parts must have a length of 3. Once this condition is satisfied, the

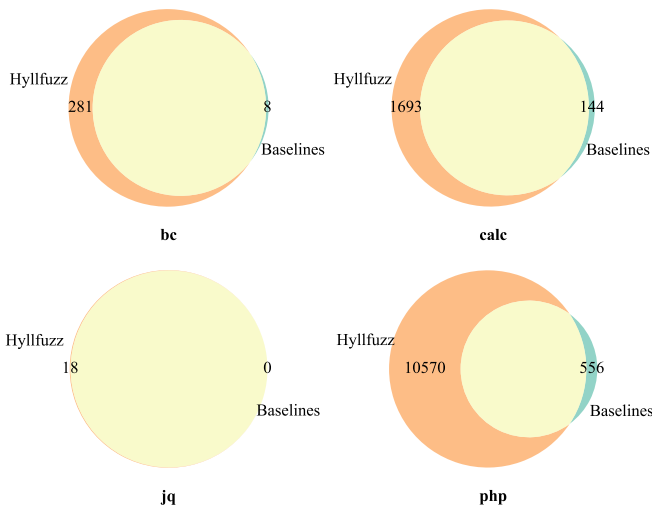


Fig. 8: Number of code branches uniquely covered by HYLLFUZZ (in orange) and the baselines (in blue).

input passes the first constraint and triggers the execution of code ②. Although no new conditions are introduced in ②, it mocks the behavior of a request, and any concolic execution engines require proper modeling of these behaviors. Following this, the function `doexecCommandUtils` is invoked, with the three input parts passed as its arguments: `parts[2]`, `parts[0]` for `req`, and `parts[1]` for `resp`. Upon entering this function, to satisfy the condition ③, the request value (i.e., `parts[0]`) must contain the string “x-evil-backdoor”. The return value is then assigned to the variable `backdoorValue` (line 29), which is derived from `parts[1]` (line 15). This value is hashed using the `sha256` function (line 30), and the resulting hash must match the hash of “breakin the law” to satisfy the final condition ④.

To expose the bug, the greybox fuzzer generated inputs that reached the condition at line 28; however, it failed to generate correct inputs to cover the branch. At this point, HYLLFUZZ was invoked to identify this roadblock, slice the relevant code, and prompt the LLM to generate inputs. As a result, HYLLFUZZ managed to overcome this roadblock. Subsequently, HYLLFUZZ was iteratively involved in reaching deeper and deeper code locations, and finally exposed the bug. Based on our evaluation, traditional concolic execution engines are incapable of exposing this bug. Condition ④ is related to a one-way function, which is unsolvable for traditional constraint solvers. Moreover, satisfying condition ③ requires precise reasoning over code ②, which poses another significant challenge for traditional concolic execution.

HYLLFUZZ can be effectively extended to test programs implemented in a different programming language.

G. Discussion

We evaluated HYLLFUZZ on widely-used real-world subjects with diverse input types and large codebases. Our results demonstrate that this approach is highly promising, achieving substantial improvements in code coverage compared to the

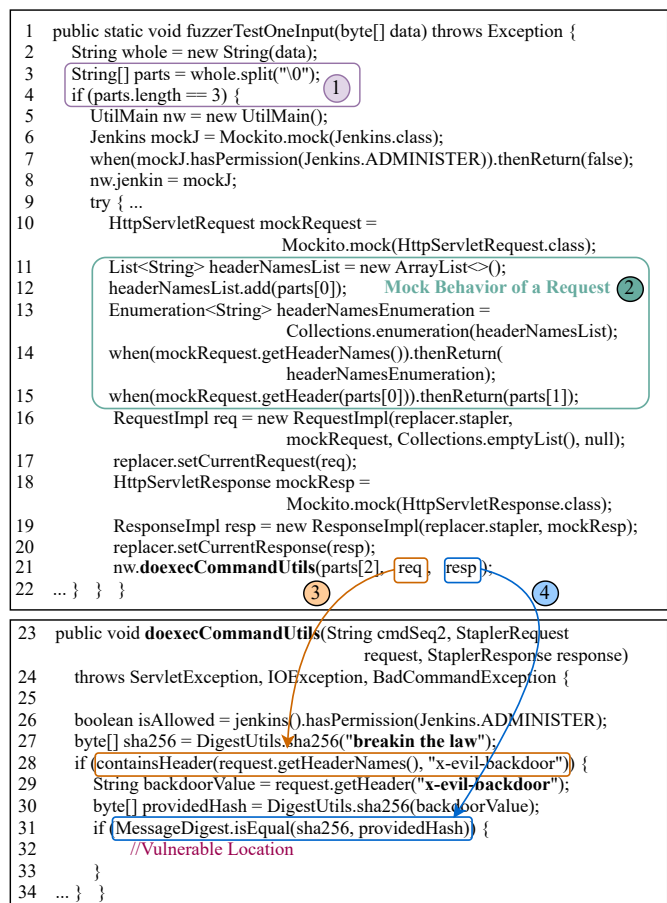


Fig. 9: Case study of the bug from the Jenkins subject in Java released in the 2024 DARPA and ARPA-H’s AIXCC.

baselines. Currently, our findings are limited to text-based inputs, including both string-based and arithmetic-based inputs. However, these findings may not be generalized to other types of binary-based inputs, as the current LLMs face challenges in generating non-text responses. Furthermore, the applicability of HYLLFUZZ is currently restricted to open-source programs, as it relies on code slices extracted from the source code.

H. Limitations

Existing hybrid fuzzers execute locally on CPU-based hardware, whereas HYLLFUZZ relies on a remotely hosted LLM service with asymmetric computational resources (e.g., GPUs). As a result, the reported wall-clock performance also reflects differences in the underlying execution environments. However, our primary goal is to evaluate the practical effectiveness of LLM-assisted hybrid fuzzing as an end-to-end workflow, rather than hardware-normalized solver performance.

Another limitation concerns the possibility of data leakage in LLM-based systems. Since modern LLMs are trained on massive corpora of public data, it is possible that some or all of the evaluated programs appeared in the training data, which may influence the LLM’s ability to reason about program behavior. At the same time, HYLLFUZZ also demonstrated effectiveness on the AIXCC Jenkins case study, which was

released *after* the gpt-4o snapshot used in our experiments. That said, evaluating the extent of this influence remains an open research question.

V. RELATED WORK

A. Hybrid Fuzzing

Hybrid fuzzing was first introduced through hybrid concolic testing [37], which combines random testing with concolic execution. DRILLER [8] was the first tool to integrate greybox fuzzing with concolic execution, demonstrating effectiveness in uncovering vulnerabilities in the 2016 DARPA’s Cyber Grand Challenge subjects. However, DRILLER struggles to scale to large real-world programs due to the performance bottlenecks inherent in concolic execution. To address this limitation, QSYM [9] introduces fast symbolic emulation, accelerated constraint solving, and efficient environment modeling. Building on this, INTRIGUER [10] further improves performance by adopting field-level constraint solving. Continuing these works, HYLLFUZZ also aims to improve the efficacy of concolic execution, but by leveraging recent advancements in large language models (LLMs) and eliminating the need for translation from a high-level programming language to logic formulas.

In contrast to improving the performance of concolic execution, several hybrid fuzzers [11], [26], [27], [7] aim to improve seed synchronization and scheduling. DIGFUZZ [11] employs a Monte Carlo-based probabilistic path prioritization model to guide concolic execution to solve the most challenging paths. MEUZZ [26] leverages machine learning to predict which seeds are likely to yield better fuzzing results, while PANGOLIN [27] preserves exploration states to guide mutation and constraint solving. Based on evaluating all previous hybrid fuzzers, CoFUZZ [7] introduced a coordinated strategy to better integrate greybox fuzzing and concolic execution. These general strategies are applicable to various hybrid fuzzers, including HYLLFUZZ, to enhance overall performance. However, HYLLFUZZ itself does not target improving seed synchronization and scheduling.

B. Concolic Execution

DART [3] is the first concolic testing tool, pioneering the combination of concrete and symbolic execution to incrementally generate test inputs. CUTE [38] extends DART to handle multithreaded programs. EXE [39] performs mixed concrete and symbolic execution, while KLEE [4] is a redesign of EXE that can check a large number of different software systems. S2E [12] applies concolic execution to binary code. These foundational works lay the groundwork for concolic execution, from the core concept to real-world use. HYLLFUZZ follows the principles of traditional concolic execution, but we explore the potential to develop smart concolic execution, with knowledge of the program semantics during input generation.

With the increase of real-world programs in size and complexity, concolic execution shows limitations in checking large programs, such as significant time overhead from symbolic emulation and constraint solving, as well as the manual effort

required in environment modeling. To scale to large real-world programs, several optimizations are proposed to address these limitations. QSYM [9] uses concrete execution to model external environments and collects an incomplete set of constraints to improve efficiency. INTRIGUER [10] uses solvers for complicated constraints only and lowers the priority of instructions that use a wide range of input offsets. While these optimizations reduce time overhead and modeling efforts, they may also compromise the performance of concolic execution. Our tool seeks to address these limitations at their core by leveraging the knowledge of the LLMs. The recent study [18] shows that LLMs can reason over constraints comparable to traditional tools such as KLEE in toy programs and even outperform them in programs with implicit data flow or out-of-code constraints. These findings further validate the promise of our approach.

C. LLM-based Unit Test Generation

Recent studies [40], [41] have investigated the capabilities of LLMs in assisting in automatic unit test generation. Existing approaches use the LLMs to generate unit tests for programs in different programming languages. For example, TESTPILOT [42] prompts LLMs to generate JavaScript unit tests, while CHATUNITEST [43] and HITS [44] focus on Java unit tests. Similarly, CODAMOSA [45], COVERUP [46], and SYMPROMPT [20] guide the LLM to generate unit tests for Python programs. These prompts include natural language descriptions of the intended code functionality, the expected properties, and the source code of the methods under test. These works primarily aim to improve the quality and effectiveness of unit tests, while HYLLFUZZ focuses on improving system testing through fuzzing.

HYLLFUZZ also shares some similarities with these works. Like CODAMOSA and CHATUNITEST, HYLLFUZZ incorporates source code into LLM prompts. However, rather than including entire modules, HYLLFUZZ extracts only the relevant path constraints for a specific execution path using program analysis, and then guides the LLM to generate new inputs by reasoning over these constraints. In contrast, CODAMOSA and CHATUNITEST rely on full source code, which can hinder coverage when methods involve complex or numerous path conditions. This is a limitation noted by prior work such as HITS [44]. To mitigate this, HITS proposes decomposing focal methods into smaller slices and generating tests for each slice. However, these slices are LLM-generated fragments that may span multiple or partial execution paths and are not grounded in formal static or dynamic slicing techniques. Moreover, HITS incurs additional LLM costs for code slicing, reportedly exceeding two dollars per large file. In comparison, HYLLFUZZ employs dynamic slicing to extract path constraints aligned with specific execution goals, thereby avoiding unnecessary LLM overhead and eliminating additional slicing costs.

D. LLM-based Fuzzing

LLMs have also been employed in various fuzzers to enhance test generation during fuzz testing. TITANFUZZ [47]

uses the LLMs to generate test cases for deep learning software libraries. CHATAFL [16] uses LLMs to enhance network fuzzing. LLM4FUZZ [17] uses the LLMs to improve test case generation while fuzzing smart contracts. FUZZ4ALL [48] advances compiler testing by leveraging LLMs. In these fuzzers, the interleaving between LLMs and greybox fuzzing is similar to that of hybrid fuzzers. The LLMs are usually queried only when greybox fuzzing reaches a coverage plateau, due to the expensive cost of the LLM calls. Similarly, hybrid fuzzers, including HYLLFUZZ, invoke the heavy concolic execution whenever facing a fuzzing plateau.

However, unlike existing fuzzers, HYLLFUZZ assists greybox fuzzing in a fundamentally different way. Existing approaches exploit the extensive domain knowledge of LLMs—such as knowledge of deep-learning libraries, network protocols, and user specifications—to generate valid inputs during fuzzing. For example, CHATAFL generates new inputs based on the LLM’s understanding of network protocols, including input formats and state machines. These approaches do not dive into the codebases or reason over code paths. In contrast, our fuzzer HYLLFUZZ uses LLMs to generate new inputs by its reasoning over the code path constraints in the form of code slices, and the new inputs aim to satisfy these path constraints. This makes HYLLFUZZ analogous to traditional concolic execution integrated into hybrid fuzzing, while HYLLFUZZ enhances traditional concolic execution through LLM-based reasoning. To the best of our knowledge, no previous works leverage LLMs to improve the efficacy of hybrid fuzzing.

E. LLM-based Path Reasoning

Recent studies [49], [50] have also leveraged LLMs to reason about path constraints, though in different contexts and with different research goals from HYLLFUZZ. LLMDFA focuses on analyzing dataflow information, where LLMs are used to extract source and sink locations, and then summarize their dataflows based on path constraints. On the other hand, LLIFT aims to improve the performance of detecting Use-Before-Initialization bugs; here, LLMs are used to identify variable initializers and their post conditions, and subsequently determine whether variables are properly initialized. Both LLMDFA and LLIFT are designed to advance static analysis. In contrast, HYLLFUZZ leverages LLMs to reason over path constraints to enable smart concolic execution, thereby improving the fuzzing performance.

F. LLM-based Symbolic Execution

A closely related line of work is *LLM-based symbolic execution*, as implemented by AUTOBUG [51], which uses LLMs for path-based program analysis. While HYLLFUZZ and AUTOBUG share the common idea of leveraging LLMs to reason about program behavior, they represent two distinct design points in the space of LLM-assisted program analysis. The key differences are summarized in Table VIII.

Conceptually, HYLLFUZZ optimizes for *practical input-space exploration* under fuzzing, whereas AUTOBUG targets *systematic path reasoning* for program analysis. As such, HYLLFUZZ emphasizes *efficiency* and *scalability* in testing

TABLE VIII: HYLLFUZZ and AUTOBUG [51] comparison.

Aspect	HYLLFUZZ	AUTOBUG
<i>Execution model</i>	Concolic	Symbolic
<i>Goal</i>	Fuzzing	Analysis / verification
<i>Constraints</i>	Dynamic slicing	Symbolic slicing
<i>Guarantees</i>	None (fuzzing)	Formal foundation
<i>Coverage</i>	Edge	Path
<i>Cost</i>	Lower	Higher

scenarios, whereas AUTOBUG emphasizes principled reasoning over program paths. To highlight the differences, we consider the loop in lines 7–17 from Fig. 1. *AutoBug* systematically explores path summaries via path decomposition. In contrast, HYLLFUZZ only considers the subset of paths *driven by coverage-guided fuzzing*, and will invoke LLM-based reasoning only when a roadblock is encountered.

Considering only the loop, AUTOBUG explores up to 7 distinct path summaries derived from the code structure, each potentially triggering an LLM invocation. In contrast, HYLLFUZZ explores at most 3 paths to achieve edge coverage in this example, and invokes the LLM only if a roadblock is encountered. If the other 20+ conditionals are considered, the number of unique path summaries can increase combinatorially. This difference reflects a fundamental trade-off: HYLLFUZZ reduces analysis cost by focusing on *reachable and coverage-relevant paths*, whereas AUTOBUG aims for systematic path exploration. HYLLFUZZ is therefore more suitable for large-scale fuzzing scenarios where exhaustive path reasoning is infeasible.

VI. CONCLUSION

In this paper, we present HYLLFUZZ, a hybrid fuzzer that combines greybox fuzzing with smart concolic execution powered by large language models (LLMs). Taking advantage of recent advances in LLMs, HYLLFUZZ explores the potential to develop smart concolic execution, with knowledge of the code semantics during input generation. In our evaluation, HYLLFUZZ shows strong promise in terms of effectiveness, efficiency, and usability. Our evaluation results show that HYLLFUZZ can cover 31%–59% more code branches than the state-of-the-art hybrid fuzzers QSYM, INTRIGUER, and COFUZZ, leading to more bugs found. In addition, HYLLFUZZ reduces the time usage of each concolic execution run by 3–19 times. Moreover, HYLLFUZZ avoids the combination of a heavy symbolic execution infrastructure and reduces the manual effort involved in environment modeling and system call support, which naturally enhances the usability of hybrid fuzzing and lowers the barrier to its adoption.

ACKNOWLEDGMENTS

This research is supported by the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Fuzz Testing NRF-NCR25-Fuzz-0001). Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, and Cyber Security Agency of Singapore.

REFERENCES

- [1] M. Böhme, C. Cadar, and A. Roychoudhury, “Fuzzing: Challenges and reflections,” *IEEE Software*, vol. 38, no. 3, 2021.
- [2] M. Böhme, V.-T. Pham, and A. Roychoudhury, “Coverage-based grey-box fuzzing as markov chain,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, pp. 1032–1043.
- [3] P. Godefroid, N. Klarlund, and K. Sen, “Dart: Directed automated random testing,” in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, 2005, pp. 213–223.
- [4] C. Cadar, D. Dunbar, D. R. Engler *et al.*, “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)*, 2008, pp. 209–224.
- [5] S. Poeplau and A. Francillon, “Symbolic execution with {SymCC}: Don’t interpret, compile!” in *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 181–198.
- [6] M. Böhme, C. Cadar, and A. Roychoudhury, “Fuzzing: Challenges and reflections,” *IEEE Software*, vol. 38, no. 3, pp. 79–86, 2020.
- [7] L. Jiang, H. Yuan, M. Wu, L. Zhang, and Y. Zhang, “Evaluating and improving hybrid fuzzing,” in *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2023, pp. 410–422.
- [8] 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 2016 Network and Distributed System Security Symposium (NDSS)*, vol. 16, 2016, pp. 1–16.
- [9] 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)*, 2018, pp. 745–761.
- [10] M. Cho, S. Kim, and T. Kwon, “Intriguer: Field-level constraint solving for hybrid fuzzing,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, pp. 515–530.
- [11] L. Zhao, Y. Duan, and J. XUAN, “Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing,” in *Proceedings of the 2019 Network and Distributed System Security Symposium (NDSS)*, 2019.
- [12] V. Chipounov, V. Kuznetsov, and G. Candea, “S2e: A platform for in-vivo multi-path analysis of software systems,” in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011, pp. 265–278.
- [13] R. Meng, G. J. Duck, and A. Roychoudhury, “Program environment fuzzing,” in *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2024, pp. 720–734.
- [14] A. Pandey, P. R. G. Kotcharlakota, and S. Roy, “Deferred concretization in symbolic execution via fuzzing,” in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2019, pp. 228–238.
- [15] L. De Moura and N. Björner, “Z3: An efficient smt solver,” in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337–340.
- [16] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, “Large language model guided protocol fuzzing,” in *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024.
- [17] C. Shou, J. Liu, D. Lu, and K. Sen, “Llm4fuzz: Guided fuzzing of smart contracts with large language models,” *arXiv preprint arXiv:2401.11108*, 2024.
- [18] Z. Jiang, M. Wen, J. Cao, X. Shi, and H. Jin, “Towards understanding the effectiveness of large language models on directed test input generation,” in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2024, pp. 1408–1420.
- [19] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, “Large language models for software engineering: A systematic literature review,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2023.
- [20] 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.
- [21] GNOME, “libxml2,” <https://github.com/GNOME/libxml2>.
- [22] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, “Fuzzbench: an open fuzzer benchmarking platform and service,” in *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering (ESEC/FSE)*, 2021, pp. 1393–1403.
- [23] A. Hazimeh, A. Herrera, and M. Payer, “Magma: A ground-truth fuzzing benchmark,” *Proc. ACM Meas. Anal. Comput. Syst.*, 2020.
- [24] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng *et al.*, “{UNIFUZZ}: A holistic and pragmatic {Metrics-Driven} platform for evaluating fuzzers,” in *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*, 2021, pp. 2777–2794.
- [25] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, “Collafi: Path sensitive fuzzing,” in *Proceedings of the 39th IEEE Symposium on Security and Privacy (IEEE S&P)*, 2018, pp. 679–696.
- [26] Y. Chen, M. Ahmadi, R. M. Farkhani, B. Wang, and L. Lu, “{MEUZZ}: Smart seed scheduling for hybrid fuzzing,” in *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020, pp. 77–92.
- [27] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang, “Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction,” in *Proceedings of the 41st IEEE Symposium on Security and Privacy (IEEE S&P)*, 2020, pp. 1613–1627.
- [28] 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 (CCS)*, 2018, pp. 2123–2138.
- [29] G. Neumann, M. Harman, and S. Poulding, “Transformed varghadelaney effect size,” in *Proceedings of Search-Based Software Engineering: 7th International Symposium*, 2015, pp. 318–324.
- [30] M. Wu, J. Xiang, K. Chen, P. Di, S. H. Tan, H. Cui, and Y. Zhang, “Tumbling down the rabbit hole: How do assisting exploration strategies facilitate grey-box fuzzing?” in *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2025, pp. 2036–2048.
- [31] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, “Nautilus: Fishing for deep bugs with grammars,” in *Proceedings of the 2019 Network and Distributed System Security Symposium (NDSS)*, vol. 19, 2019, p. 337.
- [32] J. Wang, B. Chen, L. Wei, and Y. Liu, “Superior: Grammar-aware grey-box fuzzing,” in *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 724–735.
- [33] C. Holler, K. Herzig, and A. Zeller, “Fuzzing with code fragments,” in *Proceedings of the 21st USENIX Security Symposium (USENIX Security)*, 2012, pp. 445–458.
- [34] “Jenkins,” <https://www.jenkins.io/>.
- [35] Code-Intelligence, “Coverage-guided, in-process fuzzing for the jvm,” <https://github.com/CodeIntelligenceTesting/jazzer>.
- [36] “A1xCC,” <https://aicyberchallenge.com/>.
- [37] R. Majumdar and K. Sen, “Hybrid concolic testing,” in *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, 2007, pp. 416–426.
- [38] K. Sen, D. Marinov, and G. Agha, “Cute: A concolic unit testing engine for c,” *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, 2005.
- [39] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, “Exe: Automatically generating inputs of death,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, pp. 1–38, 2008.
- [40] 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*, 2023.
- [41] L. Yang, C. Yang, S. Gao, W. Wang, B. Wang, Q. Zhu, X. Chu, J. Zhou, G. Liang, Q. Wang *et al.*, “On the evaluation of large language models in unit test generation,” in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2024, pp. 1607–1619.
- [42] 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*, 2023.
- [43] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, “Chatunitest: A framework for llm-based test generation,” in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE)*, 2024, pp. 572–576.
- [44] Z. Wang, K. Liu, G. Li, and Z. Jin, “Hits: High-coverage llm-based unit test generation via method slicing,” in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2024, pp. 1258–1268.

- [45] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, “Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models,” in *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2023, pp. 919–931.
- [46] J. A. Pizzorno and E. D. Berger, “Coverup: Coverage-guided llm-based test generation,” *arXiv preprint arXiv:2403.16218*, 2024.
- [47] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, “Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models,” in *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis (ISSTA)*, 2023, pp. 423–435.
- [48] 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 (ICSE)*, 2024, pp. 1–13.
- [49] C. Wang, W. Zhang, Z. Su, X. Xu, X. Xie, and X. Zhang, “Llmdfa: analyzing dataflow in code with large language models,” *Advances in Neural Information Processing Systems*, vol. 37, pp. 131 545–131 574, 2024.
- [50] H. Li, Y. Hao, Y. Zhai, and Z. Qian, “Enhancing static analysis for practical bug detection: An llm-integrated approach,” *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 474–499, 2024.
- [51] Y. Li, R. Meng, and G. J. Duck, “Large language model powered symbolic execution,” *Proceedings of the ACM on Programming Languages*, vol. 9, no. OOPSLA2, pp. 3148–3176, 2025.