

# R1-FUZZ: Specializing Language Models for Textual Fuzzing via Reinforcement Learning

Jiayi Lin  
 The University of Hong Kong  
 Hong Kong SAR, China  
 linjy01@connect.hku.hk

Liangcai Su  
 The University of Hong Kong  
 Hong Kong SAR, China  
 liangcaisu@connect.hku.hk

Junzhe Li  
 The University of Hong Kong  
 Hong Kong SAR, China  
 jzzzli@connect.hku.hk

Chenxiong Qian\*  
 The University of Hong Kong  
 Hong Kong SAR, China  
 cqian@cs.hku.hk

## Abstract

Fuzzing is effective for vulnerability discovery but struggles with complex targets such as compilers, interpreters, and database engines, which accept textual input that must satisfy intricate syntactic and semantic constraints. Although language models (LMs) have attracted interest for this task due to their vast latent knowledge and reasoning potential, their practical adoption has been limited. The major challenges stem from insufficient exploration of deep program logic among real-world codebases, and the high cost of leveraging larger models. To overcome these challenges, we propose R1-FUZZ, the first framework that leverages reinforcement learning (RL) to specialize cost-efficient LMs and integrate them for complex textual fuzzing input generation. R1-FUZZ introduces two key designs: *coverage-slicing-based* question construction and a *distance-based* reward calculation. Through RL-based post-training of a model with our constructed dataset, R1-FUZZ designs a fuzzing workflow that tightly integrates LMs to reason deep program semantics during fuzzing. Evaluations on diverse real-world targets show that our design enables a small model, named R1-FUZZ-7B, to rival or even outperform much larger models in real-world fuzzing. Notably, R1-FUZZ achieves up to 75% higher coverage than state-of-the-art fuzzers and discovers 29 previously unknown vulnerabilities, demonstrating its practicality.

## 1 Introduction

Software fuzz testing, or *fuzzing*, is an effective technique for uncovering hidden security vulnerabilities by generating diverse inputs to execute the programs under testing, aiming at exploring deep semantic logic [4, 24, 28]. One of the most persistent challenges in fuzzing lies in the input generation for target programs that handle complex structured inputs, such as language compilers, interpreters, databases, etc. These targets expect inputs that conform to intricate grammars, embody rich semantic rules, and engage deep program logic. Existing approaches [5, 9, 14, 30] usually suffer from scalability and adaptability issues. Besides requiring

non-trivial manual labor for each target, any evolution of the target program, such as new language features, might render their handcrafted specifications, mutators, or ad-hoc rules incomplete, leaving program states unexplored. In this paper, we refer to fuzzing such targets as *textual fuzzing*, where the input complexity arises from structured printable formats, and focuses on the central challenge of *generating high-quality fuzzing inputs for complex textual targets*.

For textual fuzzing, large language models (LLMs) offer transformative potential for overcoming these limitations. First, they possess vast latent knowledge from pretraining on large programming and natural language datasets, equipping LLMs with a "universal corpus" that encodes syntax, semantics, and idioms of the targets under testing. Second, LLMs exhibit strong semantic understanding ability. They can reason about program intent from rich semantic cues, not only from structural code, but also natural language semantics inside code symbol names or comments, which are extremely challenging for conventional methods to harness. This allows LLMs to infer how to construct meaningful, syntactically correct, and semantically relevant inputs that reach deep program branches. For instance, Listing 1 is an example we observed during our experiments. In the target program *SQLite*, the conditional branch after line 9 is consistently unreached by traditional fuzzers, while LLM can easily generate relevant inputs (line 13) to satisfy the condition, improving fuzzing code coverage. It is because the code comments in lines 4 to 8 provide functionality explanations of the following branch (SQL queries with `RANGE BETWEEN a FOLLOWING AND b FOLLOWING` where `a` equals `b`) that LLM can understand and harness.

However, the reasoning potential of LLMs has not been fully leveraged in practice. A primary obstacle is the challenge of applying them effectively to large-scale, real-world codebases. Although such codebases contain rich semantic information crafted for human comprehension, their scale exacerbates well-known limitations of LLMs: the trade-off between hallucination and context window sizes [18]. Moreover, the semantic cues required to reach specific program

\*Corresponding author.

---

```

1 // sqlite/src/window.c
2 static int windowCodeOp(WindowCodeArg *p, ...
3 ...
4 /* If this is a (RANGE BETWEEN a FOLLOWING AND b
5 ** FOLLOWING) or (RANGE BETWEEN b PRECEDING AND
6 ** a PRECEDING) frame, ensure the start cursor
7 ** does not advance past the end cursor within the
8 ** temporary table. ... */
9 if( pMWin->eStart==pMWin->eEnd && regCountdown
10 && pMWin->eFrmType==TK_RANGE ...
11
12 // A part of test input successfully generated by LLMs
13 // (by understanding the comments above) to satisfy
14 // and test the conditional branch:
15 SELECT b OVER w FROM t WINDOW w AS (ORDER BY b RANGE
16 BETWEEN 10 AND 10);

```

---

**Listing 1.** Example of pervasive, underutilized semantic cues (e.g., code comments, symbol names) that are suitable for language models to harness during fuzzing input generation, from *SQLite*.

branches are often fragmented throughout deep or nested call chains, making them difficult to isolate and present meaningfully to LLMs. Therefore, constructing targeted prompts that reliably guide LLMs to explore deep program logic for fuzzing remains unaddressed. As a result, existing approaches often circumvent this issue by leveraging LLMs on curated, targeted documentation [23, 34] or isolated fuzzing inputs [12, 13] that have a manageable size. To the best of our knowledge, there is no existing work that harnesses LLMs for fuzzing by directly reasoning over the target codebases. To summarize, the crucial challenge lies in: *constructing reasonable questions from large real-world code bases to explore deep program logic (C1)*.

Furthermore, applying LLMs inherently faces the *cost and scalability concerns (C2)*. Existing strategies generally rely on querying larger models for better performance. However, large models are expensive to query repeatedly, especially within intensive tasks like fuzzing input generation. As a result, some works choose to offload LLMs outside the fuzzing loop, such as generating mutator code [26, 37] or driver code [10, 22] offline.

Fortunately, recent research has introduced a promising paradigm that employs RL-based post-training [15] to align the reasoning capabilities of smaller models, enabling them to specialize in domain-specific tasks [17, 21, 27, 33, 36]. Such an RL framework for LMs requires both well-structured questions to elicit high-quality answers and a goal-aligned reward mechanism to evaluate those answers and guide the models' parameter updates. However, existing reward strategies for LMs rely on textual pattern matching [35] or simplistic execution output comparisons [20], which are inaccurate or too sparse to effectively guide fuzzing input generation, presenting a key challenge of *designing a precise and informative reward signal (C3)*.

To systematically overcome these challenges, we present R1-Fuzz, a novel framework for textual fuzzing powered

by LLMs with reinforcement learning-based post-training. R1-Fuzz first trains a cost-efficient model to specialize at the task of textual input generation (addressing **C2**) with a novel reward signal, and then uses it to generate high-quality inputs in real-world fuzzing campaigns. Specifically, R1-Fuzz introduces two key designs: **coverage-slicing-based question construction** and **distance-based reward calculation**. To address **C1**, R1-Fuzz decomposes the program by executing a seed input to identify nearby uncovered branches (*i.e.*, conditional paths not taken). For each target branch, it constructs a focused question by slicing the source code along the execution path from the program entry to that branch. This question construction method is used in both generating datasets for training and generating questions on the fly during fuzzing. Next, to address **C3**, our distance-based reward mechanism provides a finer-grained signal by computing the distance between the execution path of an LM-generated input and the target branch. This yields a more accurate and less sparse training signal, inspired by research on directed fuzzing [8].

Based on these two designs, R1-Fuzz consists of three stages. ① Dataset Construction: using our coverage-slicing-based question construction, given a target program and an input seed, R1-Fuzz can generate a set of questions. For a given set of seeds (*i.e.*, corpus), R1-Fuzz can generate a variety of questions leading to diverse deep code paths of the target program. Furthermore, by incorporating more target programs, R1-Fuzz can construct a comprehensive dataset consisting of diverse real-world source code that captures a wide range of program semantics. The second stage is: ② RL-based Post-training. R1-Fuzz uses the Group Relative Policy Optimization (GRPO) algorithm [15] to train a language model to excel at our task. By integrating our reward mechanism, R1-Fuzz stimulates the potential of a cost-efficient small model, named **R1-Fuzz-7B**, to effectively generate targeted fuzzing inputs.

The third stage in R1-Fuzz is a: ③ LLM-powered Fuzzing Loop. We design a fuzzing workflow that can tightly integrate LMs into a coverage-guided fuzzing loop. Specifically, the loop maintains an evolving fuzzing corpus of input seeds that trigger new code coverage, and R1-Fuzz uses these seeds to continuously construct questions, which express uncovered branches by the fuzzer. During fuzzing, the model responds to the questions with new seeds that it predicts will reach a specified uncovered branch. The new seeds then enrich the fuzzing corpus for further mutation by the fuzzer, forming a loop where our model exerts its potential to reason deep program logic guided by code coverage, eventually facilitating vulnerability discovery. Moreover, beyond the generated static dataset, our fuzzing loop establishes a benchmark, enabling the integration and practical evaluation of various LMs to assess their real-world fuzzing capabilities.

To evaluate our design, we implemented R1-Fuzz and conducted comprehensive experiments to assess both the training and real-world fuzzing effectiveness. We selected a diverse set of *textual* targets, including language interpreters (PHP, Lua, Ruby, QuickJS, NJS), compilers (CPython, Solidity), and database engines (SQLite, DuckDB). We then performed all three stages in R1-Fuzz on these targets, including generating a dataset consisting of 16338 questions, training a low-cost model (R1-Fuzz-7B) based on Qwen2.5-7B-Instruct [31], and comparing the performance across different models and state-of-the-art fuzzers. Moreover, we also conducted an ablation study to demonstrate the effectiveness of the key components in R1-Fuzz. Our experiment results showed that R1-Fuzz-7B can excel at the task of fuzzing input generation, comparable to or even outperforming larger and costly models (DeepSeek-V3 [15], GPT-o4mini [25]) in real-world fuzzing. Furthermore, our LLM-powered fuzzing loop showed impressive performance on both code coverage and vulnerability discovery. On our selected targets, R1-Fuzz achieved 75% higher coverage than state-of-the-art fuzzers [5, 9, 16, 28, 30] and discovered 29 previously unknown vulnerabilities, with 24 fixed or confirmed by the developers. Lastly, the ablation study showed that our question format and scheduling outperforms naive strategies by 8% and 3.8% on correct answer ratio and code coverage.

In conclusion, we made the following contributions:

- We designed a **coverage-slicing-based question construction** technique to decompose large codebases into targeted prompts, forming a dataset for LMs to reason about deep program semantics for fuzzing.
- We designed a **distance-based reward mechanism** that provides a fine-grained training signal, enabling efficient RL-based post-training of a low-cost model (R1-Fuzz-7B) to specialize in fuzzing input generation.
- We implemented a novel coverage-guided **LLM-powered fuzzing loop**. It achieves a 75% higher code coverage and discovers 5x more previously unknown bugs compared to state-of-the-art fuzzers.
- We open-sourced our model and implementation, named R1-Fuzz (<https://github.com/HKU-System-Security-Lab/R1-Fuzz>), as a comprehensive framework to post-train, deploy, and evaluate different LMs on real-world fuzzing tasks.

## 2 BackGround and Motivation

### 2.1 Fuzzing *Textual* Targets

Many software systems process complex structured text, such as compilers, interpreters, and database engines. These systems employ multi-phase pipelines where inputs (e.g., source code, SQL queries) must first be syntactically valid and then semantically meaningful to exercise deep execution logic. Fuzzing these textual targets is uniquely challenging: to

trigger deep behaviors, inputs must satisfy intricate grammar rules and semantic constraints (e.g., type systems, variable scoping). Furthermore, these targets evolve rapidly with new language features and dialects, rendering static grammar-based testing setups obsolete. We define such systems, whose input complexity stems from printable text, as *textual* targets.

Traditional approaches to test textual targets include: *Grammar-aware mutators* (e.g., Nautilus [5], Gramatron [30]) preserve syntactic validity but might neglect semantic correctness, such as producing undefined variables or type errors that cause early rejection during execution. *Handcrafted specifications* encode both syntax and semantic rules (e.g., Polyglot [9] uses ANTLR/Flex grammars with additional ad-hoc rules). However, writing and maintaining such specifications is labor-intensive and often fails to fully capture all behaviors of complex targets. *Manually curated seed corpora* (e.g., [1, 9]) provide domain-specific seeds to guide mutation, but their effectiveness is limited by the scope and quality of available seeds, leaving many feature-specific code paths untested.

To summarize, the crucial challenge for fuzzing complex textual targets persists: *testing textual targets involves deep semantic logic layered on top of complex syntactic rules, making high-quality input generation difficult and fragile*. This challenge motivates our exploration of adopting LLMs, which are inherently equipped with latent knowledge of programming languages and structures, offering the potential to bridge both syntax and semantics in a generalizable way. We discuss this potential in the following subsection.

### 2.2 Motivation Examples

To illustrate the unique capability of LLMs in textual fuzzing, consider the example shown in Listing 2. It comes from the compiler for Solidity, a smart contract programming language. The function `isImplicitlyConvertibleTo` started from line 4 checks if a `struct` type can be implicitly cast to another `_converTo` type. The conditional branch (line 10) during fuzzing in our experiments was consistently left unexplored in our experiment with state-of-the-art fuzzers, while our LLM-powered fuzzing loop successfully generated an input (lines 14 to 21) to satisfy it and improved the code coverage. The crucial part is constructing a new variable `foo` resided in the public storage location (line 15, similar to a global variable) and assigning to it with a variable (`bar`) resided in the `Memory` location (lines 18 to 19, similar to a local variable), triggering an implicit conversion. This is challenging for traditional mutation-based fuzzers because it requires multiple coordinated steps (defining variables, ensuring type consistency, performing the assignment). Moreover, this function invocation is deeply nested in the execution call stack (with a depth of 21), making testing techniques like symbolic or concolic execution easily fail due to path or state explosion [7].

---

```

1 // solidity/libsolidity/ast/Types.cpp
2 BoolResult StructType::isImplicitlyConvertible(Type
3   const& _convertTo) const
4 {
5   if (_convertTo.category() != category())
6     return false;
7   auto& convertTo = dynamic_cast<StructType const&>(
8     _convertTo);
9   // memory/calldata to storage can be converted, but
10  // only to a direct storage reference
11  if (convertTo.location() == DataLocation::Storage &&
12    location() != DataLocation::Storage && convertTo
13    .isPointer())
14  ...
15
16 // Correctly generated input from our language model
17 // to satisfy this conditional branch
18 contract test {
19   struct S { uint x; }
20   S public foo;
21   function test() public {
22     S memory bar = S({x: 1});
23     foo = bar; // convert data type located in memory
24     to storage
25   }
26 }
```

---

**Listing 2.** An example of LLM utilizing semantic cues from code comments and symbol names to infer relevant input, reaching a deep program branch that hinders traditional testing tools.

In contrast, this example is intuitive for humans with a background knowledge of Solidity, who only need to take the hints from the comment in line 9. Similarly, LLMs can exploit such explicit cues, leveraging their pretrained knowledge of programming languages. We also performed an experiment that removes the comment in line 9. The result showed that LLMs can still generate the desired input by inferring the intended behavior from *symbol names* in the source code (e.g., `Convertible`, `Storage`, `isPointer`, etc). Notably, such semantic cues are pervasive across complex software systems because they are written for human readability. Therefore, equipped with human-like capabilities of semantic interpretation, LLMs are uniquely positioned to harness these signals for the task of fuzzing input generation.

However, despite its potential, directly leveraging an LLM in practical fuzzing faces the following challenges: **C1: Constructing reasonable questions from large real-world codebases to explore deep program logic.** Large programs contain thousands of branches, and it is non-trivial to extract relevant prompts that highlight specific code logic for the LLM to reason about. To the best of our knowledge, there is no existing work that harnesses LLMs for fuzzing by reasoning over the target codebases to uncover deep program logic. Existing methods often employ LLMs on carefully selected documentation [23, 34] or isolated fuzzing inputs [12, 13], which are easier to manage in size. **C2: Managing inference cost and scalability.** Larger LMs generally perform better, but are inevitably expensive to use in the intensive fuzzing loop. Some existing works [26, 37] choose to offload LLMs outside

the fuzzing loop, such as generating mutator code offline, to avoid querying costly LLMs intensively.

In this paper, we address **C1** by designing a coverage-slicing technique that decomposes programs into compact, branch-specific prompts. This technique is lightweight enough to be conducted during fuzzing and, more importantly, can be used to construct a training dataset as illustrated in the next subsection. To address **C2**, we design a reinforcement learning-based post-training framework for smaller LMs, which can make them achieve comparable or even better performance than larger ones, significantly reducing deployment cost. Next, we discuss our motivations for RL-based post-training and the associated challenges.

### 2.3 Reinforcement Learning-based Post-training

In typical reinforcement learning (RL) for LLM setups, the model receives a prompt (*i.e.*, *question*), generates a response, and receives a reward that reflects how well the response satisfies the objective. The reward is then used to adjust the model’s parameters via policy gradient methods such as PPO or GRPO [15]. Previous efforts [17, 21, 27, 33, 35, 36] have shown that RL-based post-training smaller models can significantly improve performance, making them competitive with much larger models at a fraction of the cost. Inspired by them, we aim to use RL to train LLMs for fuzzing input generation.

However, existing works depend heavily on their domain-specific dataset and reward signal, which can not be directly applied to our task. Therefore, the first critical component is *question dataset construction for textual fuzzing*. To train the model effectively, we must construct a dataset of questions that aligns with the fuzzing objective, *i.e.*, generating inputs that explore designated uncovered code regions. To the best of our knowledge, there is no existing dataset for this purpose. This challenge aligns with **C1** described above, because it is natural to standardize the question format *for both model training and its application in actual fuzzing*, ensuring consistency.

The second critical component in RL is the reward mechanism. Most existing LLM RL work uses reward signals such as string pattern matching [15, 35] or execution output comparisons [20]. These are inadequate in fuzzing, where the objective is not to match a fixed output but to maximize the exploration of diverse program states. Therefore, we need a reward that reflects how “close” an input is to reaching a desired program location, which leads to the challenge **C3: designing a precise, non-sparse reward function that guides the model training**.

To summarize, fuzzing textual targets presents unique challenges due to deep syntax and semantic constraints. Using LLMs further faces the need for semantically relevant prompts (**C1**) and the practical cost limitations of using large models during fuzzing (**C2**). While reinforcement learning offers a promising path to address **C2**, it introduces a

new challenge in designing an effective reward mechanism (**C3**). These challenges motivate our design of R1-Fuzz in this paper, a framework that addresses these issues through coverage-slicing-based question construction (§3.2) and RL-based post-training (§3.3). Finally, R1-Fuzz harnesses our specialized, low-cost model in an LLM-powered fuzzing loop (§3.4) to facilitate real-world testing and vulnerability discovery.

### 3 R1-Fuzz Design

#### 3.1 Overview

Figure 1 illustrates the overview of our R1-Fuzz framework. It trains language models (LMs) to excel at textual input generation for fuzzing and vulnerability discovery.

The first stage of R1-Fuzz is *Dataset Construction* (stage ①, §3.2). Reinforcement learning (RL)-based post-training typically requires a high-quality dataset that is task-specific and informative. To address this, we design an automatic method to construct a dataset from scratch. The dataset consists of a set of *questions*, which embody relevant source code for the model to reason about. To construct them, R1-Fuzz begins by collecting an initial input corpus for each target program. By executing the target with inputs, stage ① performs code slicing based on run-time coverage to extract multiple *code traces*, *i.e.*, the source code alongside the execution paths. Each code trace starts from the program entry and ends at one branch condition (e.g., `if` or `case` inside a `switch`) that an input encountered during execution. We extract branches that are *uncovered* by the given input, specifically those where the condition is evaluated to one outcome (e.g., `False`) while the alternative outcome (e.g., `True`) remains unexplored. We then form a question by presenting the code trace and the original input to the LM with the instruction to *generate a new input that inverts the evaluation outcome of this branch condition (False to True)*". The LM's objective is thus to generate inputs that flip the branch decision, guiding the exploration into uncovered code regions.

Following dataset construction, stage ② is post-training via RL, in which the LM repeatedly answers our questions and receives reward feedback (§3.3). An effective reward mechanism is critical for producing goal-aligned responses. To address this, we design a new mechanism named *distance-based reward* (§3.3), which executes LM-generated inputs and computes the reward based on how closely their execution paths approach the specified target branch.

In stage ③, a post-trained LM is integrated into a fuzzing loop for practical evaluation (§3.4). As shown in the figure, the right half of the loop follows a traditional workflow: the fuzzer mutates seeds from a corpus, executes the target, and retains inputs that discover new coverage to evolve the corpus. To integrate the LM, the left half augments this process. R1-Fuzz analyzes the corpus to identify branches that remain uncovered by the fuzzer, and then uses coverage

slicing to generate questions from these branches to query the model. By responding to these questions during fuzzing, the model generates high-quality inputs that target specific unexplored regions and also enrich the corpus for further mutation.

#### 3.2 Dataset Construction

The dataset of R1-Fuzz consists of a set of questions for RL-based post-training. In textual fuzzing, the targets usually feature large codebases with numerous branch conditions to process complex textual inputs. Therefore, it is crucial to determine a concise and effective question format to exert LMs' reasoning ability. In R1-Fuzz, we mimic how humans debug and reason about program semantics: using a concrete input to trace execution paths and reason about specific program locations. Inspired by this, we develop a **coverage-slicing-based question construction** method. Each constructed question contains a sliced source code trace, a specific branch position, the content of the original input, and instruction prompts. The targeted branch is guarded by a conditional statement whose opposite outcome was left unexplored by the original input. The model's objective is to generate new inputs that satisfy the unexplored condition (e.g., invert `False` to `True`), thereby aligning its reasoning with the fuzzing objective of covering untested code.

---

#### Algorithm 1: Dataset Construction

---

```

1 Function ConstructQuestion(Branch):
2   functions  $\leftarrow$  ExtractSlice(Branch.function);
3   question.prompt  $\leftarrow$  SYSTEM_PROMPT
    $\quad\parallel$  Concat(functions)  $\parallel$  SUFFIX_PROMPT;
4   question.branch  $\leftarrow$  Branch;
5   return question;
6
6 Function ConstructDataset(Program, Seeds):
7   dataset, Coverage, BranchSet  $\leftarrow$   $\emptyset$ ;
8   foreach seed  $\in$  Seeds do
9     feedback  $\leftarrow$  Execute(Program, seed);
10    Coverage  $\leftarrow$ 
11      Coverage  $\cup$  feedback.CoveredBranches;
12    foreach
13      branch  $\in$  feedback.UncoveredBranches do
14        BranchSet.Add(branch);
15
16    foreach branch  $\in$  BranchSet do
17      if branch  $\in$  Coverage then
18        question  $\leftarrow$ 
19          ConstructQuestion(branch);
20        dataset.append(question);
21
22 return dataset;

```

---

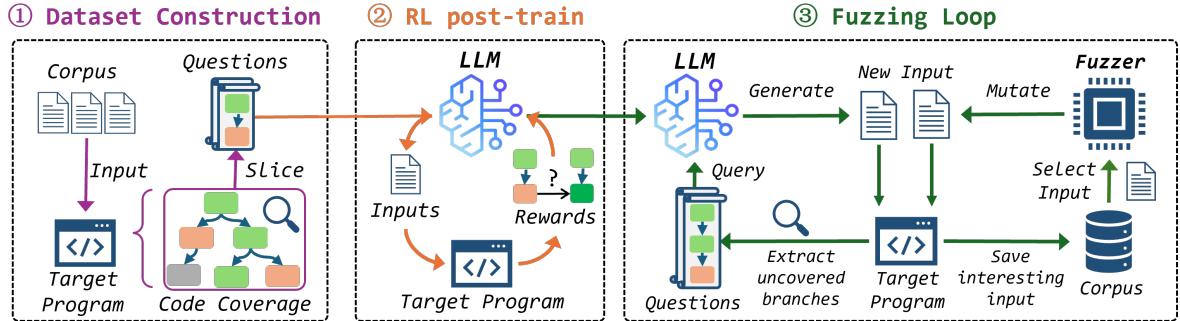


Figure 1. Overview of R1-Fuzz.

Algorithm 1 describes the process of dataset construction. The `ConstructDataset` function first receives the target program and an initial set of input seeds (*i.e.*, corpus) as arguments. Then, the construction begins by executing the program with each seed and collecting the accumulated code coverage (lines 8 to 12). For each seed, its uncovered branches are uniquely identified by its source code location and stored in the `BranchSet`, meaning the execution of this seed never reaches them. Next, after the execution of all seeds, the algorithm examines each uncovered branch to check if it belongs to the coverage set, which means that *there exist other seeds that reach this branch* (line 14). This ensures that all constructed questions have answers, *i.e.*, exists a feasible input to satisfy the condition (not dead code). While most branches in real-world targets are live, our evaluation uses targets with human-written fuzzing drivers, which may apply option settings that render certain code regions unreachable. Therefore, we apply this filtering strategy, which is also configurable to allow a trade-off between question validity and dataset size. The seeds that cover a branch serve as ground-truth answers to its question. Although they are not used in RL (stage ②), which relies on reward signals rather than ground truth labels, the answer seeds are supplied into the initial fuzzing corpus to prevent data leakage and guarantee fair evaluation of generalization (§5.2).

Next, the `ConstructQuestion` function generates one question given one uncovered branch (lines 1 to 5). It first extracts the runtime call stack to identify functions leading to the branch. To provide accurate context, we statically parse and analyze the program to maintain context information, including call graphs, function bodies, call site locations, etc. Then it forms the question body by concatenating instruction prompts with extracted code. The system prompt instructs the LMs with the objective description and format requirement for extracting generated seeds. The suffix prompt, or user prompt, specifies the target branch condition and the original and desired outcome: *Generate a new input to invert the branch condition COND's outcome from False/True to True/False*. Besides the prompts and code, we also provide the original input

content in the question, which is omitted in the algorithm for brevity.

**Question Validity.** Our design of the question format aims to balance reasonable lengths and context validity (or completeness), *i.e.*, *not all* necessary context is provided in the question body. For example, executed and returned functions not in the current call stack might also be necessary or even more helpful to reason about the target branch condition. Such a trade-off is necessitated by LMs' token limits, the scale of real-world codebases, and the efficiency demands of RL training. Nevertheless, given that LLMs are pretrained on vast and diverse corpora, their latent knowledge can also compensate for the missing context and enable effective reasoning. Our experiments (§5.3) explored the impact of question formats and demonstrated that our design can provide effective context for LMs to facilitate fuzzing performance in real-world testing. We left the exploration of more effective question formats as future work.

### 3.3 Policy Model Training

In stage ②, R1-Fuzz uses the dataset from stage ① to perform RL-based post-training via *Generative Reinforcement Policy Optimization* (GRPO) [15]. The effectiveness of this training heavily depends on the design of the reward function, which must guide the model to generate inputs that explore specific branch conditions.

Existing reward mechanisms—such as human feedback, rule-based pattern matching, or strict execution output comparison—are not ideal for this task. Pattern-matching rewards with collected ground-truth answers (§3.2) are restrictive and harm generalization, as multiple valid inputs can invert a branch. Binary execution feedback (*e.g.*, rewarding only on branch reachability) is too sparse for effective training. To address this, we introduce a coverage distance reward that provides dense, incremental feedback. Inspired by directed fuzzing [8], our reward function  $r(x, y)$  quantifies how closely a generated input  $y$ 's execution path approaches the target branch, relative to the original input  $x$ .

Formally, let  $T(x)$  and  $T(y)$  denote the function-level execution traces of the original and generated inputs, respectively,  $F$  as the function containing the target branch, and  $C$  ( $\neg C$ ) as the original (inverted) outcome of the branch condition. We define their function level *coverage distance* as

$$d(x, y) = \frac{|\text{LongestCommonPrefix}(T(x), T(y))|}{|T(x)|}, \quad (1)$$

and the reward function  $r(x, y)$  as

$$r(x, y) = \begin{cases} d(x, y) & \text{if } F \text{ not reached by } y, \\ 1 & \text{if branch reached with } C \text{ by } y, \\ 2 & \text{if branch reached with } \neg C \text{ by } y, \\ 0.1 & \text{penalty if } y = x. \end{cases} \quad (2)$$

For example, if the original input's executed function trace towards a branch condition is `main->f1->f2->f3:condition1:False`, and the new input's trace is `main->f1->f2->f4->f5`, their distance is measured by the number of same functions they executed, resulting in  $3/4=0.75$ . When the new trace can reach the function containing the target branch but does not invert the condition outcome (*i.e.*, has the same effect as the original input), we reward the model by 1. When the new input successfully inverts the branch outcome, we reward the model by 2. Notably, since the original input is given and would be rewarded positively by Formula 1, we instruct the model not to respond with the same input and penalize such behavior to avoid *reward hacking* by deducting the reward to 0.1. As a result, the total reward  $r(x, y)$  lies in the range [0, 2] before being normalized into GRPO [15].

### 3.4 LLM-powered Fuzzing Loop

In stage ③ of R1-Fuzz, we design an LLM-powered fuzzing loop to harness LMs for real-world fuzzing, aiming at enhancing code coverage and vulnerability discovery. This stage enables a practical and comprehensive evaluation of LMs on the task of fuzzing input generation. Because it allows R1-Fuzz to not only assess accuracy on the generated static dataset but also measure model performance in practical fuzzing campaigns.

As shown in Figure 1, the fuzzing loop surrounds feeding generated and mutated test inputs into the target programs to reveal potential vulnerabilities through monitoring if the execution crashes. Algorithm 2 describes the process of our fuzzing loop. It maintains an accumulated *coverage* set of reached branches during fuzzing. In each iteration, the fuzzer selects a seed from the fuzzing corpus to mutate and execute (lines 3 to 5). For each mutated seed, the fuzzer examines if it covers previously uncovered code; if it does, the corpus is updated with the new seed and the *coverage* set is updated accordingly; otherwise, the loop continues to the next iteration (lines 7 to 10).

---

### Algorithm 2: LLM-powered Fuzzing Loop

---

```

1 Coverage  $\leftarrow \emptyset$ ;
2 Queue  $\leftarrow \emptyset$ ;
3 while Fuzzer.NotStopped() do
4   seed  $\leftarrow$  Corpus.ScheduleNext();
5   newSeed  $\leftarrow$  Fuzzer.Mutate(seed);
6   feedback  $\leftarrow$  Execute(Program, newSeed);
7   hasNewCov  $\leftarrow$ 
8     Corpus.CheckSave(newSeed, feedback);
9   if  $\neg$  hasNewCov then
10    | continue;
11   Coverage  $\leftarrow$ 
12     Coverage  $\cup$  feedback.CoveredBranches;
13   foreach ub  $\in$  feedback.UncoveredBranches do
14     if ub  $\notin$  Coverage then
15       | question, priority  $\leftarrow$ 
16         ConstructQuestion(ub);
17       | Queue.Enqueue(question, priority);
18 ...

```

---

Next, to integrate LMs, R1-Fuzz extracts uncovered branches for each seed (line 11) and checks if they are *not reached* by previous seeds (line 12). This check differs from the check in Algorithm 1 line 14, since in real-world fuzzing, R1-Fuzz aims at exploring uncovered branches. Afterwards, it performs the same `ConstructQuestion` process as in Algorithm 1. Notably, instead of querying LMs immediately after constructing each question, we maintain a priority queue (`queue`) to perform better *question scheduling* (line 14). If an uncovered branch is repeatedly encountered without being reached, it suggests that both LMs and the fuzzing mutator are stuck on that particular branch. In such cases, R1-Fuzz uses the negation of the queried count as the *priority*, ensuring that LMs explore newly encountered branches instead of repeatedly querying the same question constructed from a challenging branch. We omit certain details in the algorithm, including a consumer thread to retrieve questions from the `queue`, querying LMs to execute and save seeds, monitoring crashes, etc.

## 4 Implementation

We implemented Algorithm 1 and Algorithm 2 in R1-Fuzz in nearly 5000 lines of Python code. The training stage is implemented based on the *Verl* [29] reinforcement learning framework, in which we integrated our coverage distance-based reward calculation. The reward calculation is based on the source-based code coverage in *LLVM* [2], which we modified to collect covered and uncovered branches more efficiently. The fuzzing loop (Algorithm 2) is implemented on top of the fuzzer integration in *FuzzBench* [24].

## 5 Evaluation

In the evaluation of R1-Fuzz, we aim to answer the following research questions:

- **RQ1:** Can RL-based training enhance the performance of LMs on our fuzzing input generation dataset (§5.1)?
- **RQ2:** Can trained LMs excel in real-world fuzzing (§5.2)?
- **RQ3:** How effective are the design choices of R1-Fuzz (§5.3)?

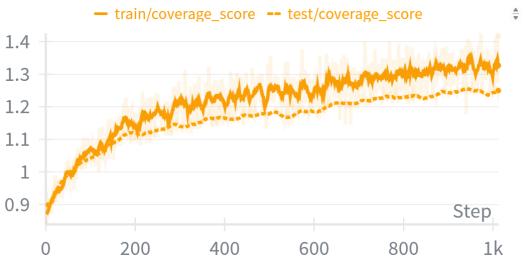
**Datasets.** To construct our dataset of textual fuzzing input generation (*i.e.*, stage ①), we select 10 real-world targets in total. They consist of language compilers, interpreters, and database engines, including PHP, CPython, Lua, mruby, NJS, QuickJS, Solidity, SQLite, Sql-parser, and DuckDB. All of them have manually crafted fuzzing drivers by experts maintained by OSS-Fuzz [4], which offers unified entry points for fair experiments.

To prepare the dataset construction corpus required by Algorithm 1, we directly use the initial fuzzing corpus provided by OSS-Fuzz. For targets that have no provided corpus, we ran a baseline fuzzer (*AFL++* [16]) for 24 hours and collected the fuzzing corpus as our dataset construction corpus. To limit the dataset size, we randomly sampled up to 1,000 seeds from the initial corpus when its size exceeded this threshold. As a result, we constructed a dataset with 16338 questions in total for the 10 targets.

### 5.1 RL-based Post-training Evaluation

In this section, we present the results that demonstrate the effectiveness of RL-based post-training on our dataset.

In the training stage (*i.e.*, stage ②), we used *Qwen2.5-7B-Instruct* [31] as our base model. We randomly partitioned the prepared dataset into training and testing sets in a 9:1 ratio and used the training set to train the model using GRPO (§3.3) with a batch size of 128. To balance stability and exploration, we enabled KL regularization with a low-variance estimator and a coefficient of 0.001. Rollouts were performed with eight candidate generations per question under a temperature of 1.0. We calculated reward scores on the testing set every two training steps. Figure 2 shows the statistics during training, including the mean of reward scores on the training and testing datasets across 1000 training steps. The reward scores have a range between 0 to 2 based on Formula 2. As shown by the trends, both training and testing rewards steadily increased and gradually converged to stable values, indicating that the model successfully learned to align its output with the objective. Moreover, the testing reward scores closely followed the training scores without significant divergence, suggesting that the model did not overfit to the training set and generalized well to unseen data. We named the trained model after 1000 steps of RL training as **R1-Fuzz-7B**.



**Figure 2.** RL-based post-training statistics (reward scores on the training and testing dataset).

**Table 1.** Static performance of different models on the testing dataset (1640 questions in total)

	Qwen2.5-7B	Qwen2.5-32B	Deepseek-V3	GPT-o4mini	R1-Fuzz-7B
Pass@1	145 (8%)	264 (16%)	391 (23%)	583 (35%)	<b>830 (50%)</b>
Pass@5	342 (20%)	543 (33%)	651 (39%)	811 (49%)	<b>904 (55%)</b>

Table 1 shows performance statistics on our dataset from different models. In addition to *Qwen2.5-7B* and *R1-Fuzz-7B*, we included both open-source models (*Qwen2.5-32B* [31], *Deepseek-V3* [15]) and a closed-source model (*GPT-o4mini* [25]). Each model was evaluated by answering all questions in the testing dataset, and we measured their accuracy using *Pass@1* and *Pass@5* scores, which capture the fraction of questions correctly answered on the first attempt and within five attempts, respectively. The results show that *R1-Fuzz-7B* outperforms both its base model *Qwen2.5-7B* (with nearly 5x higher scores) and larger models (with nearly 60% higher scores), demonstrating the effectiveness of our RL-based training in boosting reasoning ability on our dataset.

### 5.2 Fuzzing Evaluation

In this section, we present the results of evaluating R1-Fuzz-7B in practical fuzzing campaigns on our selected targets. We conducted two sets of experiments: the first set compares the performance between different language models in fuzzing (§5.2.1), and the second set compares our LLM-powered fuzzing loop (stage ③ in Figure 1) with different state-of-the-art fuzzers (§5.2.2). Lastly, we presented the results of newly discovered vulnerabilities in §5.2.3. All the fuzzing experiments lasted for 24 hours for each target, and we ran five trials to calculate the average of code coverage. All fuzzing experiments were conducted on a server equipped with an Intel Xeon Gold 5418Y CPU (128 cores) and 512 GB of RAM.

**Data Leakage Prevention.** Besides using the initial corpus provided by OSS-Fuzz [4] for all fuzzers when starting fuzzing, we augmented this corpus with the “answered seeds” extracted from our constructed training dataset. As described in §3.2, each constructed question can be associated with one or more correct answer seeds, which can serve as a form of ground truth. By incorporating these answered seeds into the initial corpus, we ensure that the corresponding branches are already covered at the beginning of fuzzing.

**Table 2.** Fuzzing performance (code region coverage, ratio of correct answers, number of total questions) from different models integrated in the LLM-powered fuzzing loop in R1-Fuzz (the darker the cell color, the better).

		PHP	CPython	Lua	mruby	NJS	QuickJS	Solidity	Sqlite	Sql-parser	DuckDB
Baseline	Cov	82526	34470	7645	7923	8775	14377	21032	26942	1491	18107
Qwen2.5-7B	Cov	82140	34870	7808	14036	8393	14653	21562	27707	1951	36352
	Ratio	0.2% (7048)	0.1% (7435)	0.1% (7242)	0.1% (7043)	0.9% (7352)	0.5% (8145)	7% (7968)	1.7% (7142)	1.1% (6506)	10% (6417)
Qwen2.5-32B	Cov	82298	36364	7778	14486	10840	14847	21563	27715	1961	38180
	Ratio	0.3% (4658)	0.2% (6929)	0.2% (6917)	0.9% (6771)	2.2% (6866)	2.9% (6160)	18% (6001)	5.1% (6542)	0.9% (5870)	13% (6288)
Deepseek-V3	Cov	83066	36452	7880	14929	11662	15942	21602	27814	1965	43769
	Ratio	6.8% (5113)	0.5% (6210)	3.4% (7394)	5.2% (5518)	7.1% (5580)	4.9% (5709)	19% (5572)	10% (6409)	0.9% (6388)	18% (4928)
GPT-o4mini	Cov	88554	34793	8108	15619	13158	17507	21816	28791	2013	51070
	Ratio	12% (5662)	0.2% (6700)	4% (6953)	9.1% (6162)	7.9% (6577)	6% (6800)	44% (6451)	25% (6615)	1.1% (6643)	28% (5505)
R1-Fuzz-7B	Cov	83967	36966	7923	15088	14624	16941	21585	28441	1994	41363
	Ratio	8.2% (5772)	0.8% (7736)	2.9% (7547)	6.9% (6808)	13.5% (7659)	5.1% (6872)	23% (6906)	12% (6975)	1.2% (6980)	20% (5776)

Consequently, questions constructed during training will not reappear during fuzzing, since their associated branches no longer represent uncovered code. In other words, all questions encountered by R1-Fuzz-7B during fuzzing are outside its training set, preventing data leakage and ensuring a fair evaluation of generalization.

**5.2.1 Comparison of Models.** We used one of the most widely-used and well-maintained fuzzing engines, *AFL++* [16], as the base fuzzing engine of stage ③. Then, we created different instances of LLM-powered fuzzers named based on different language models they used, such as *AFL++Qwen2.5-7B*, or *AFL++R1-FUZZ-7B*, etc. In this subsection, we omit the "*AFL++*" prefix for brevity.

Table 2 shows the results, where the *Baseline* is the original *AFL++* without the LLM-powered loop design. The *Cov* row indicates the number of covered code regions calculated by the source-based coverage in LLVM [2], and the *Ratio* row contains the ratio of correctly answered questions and the total number of asked questions during 24 hours' fuzzing. For example, for *PHP*, R1-Fuzz-7B constructed 5772 questions in total in 24 hours, with 473 (8.2%) input seeds generated by the model that correctly answered the question, *i.e.*, reached previously uncovered branches.

The results in the table demonstrate several key findings. First, all fuzzers under our LLM-powered fuzzing loops outperform the baseline, confirming the effectiveness of our stage ③ design in R1-Fuzz. On average, the coverage achieved by R1-Fuzz exceeds the baseline by nearly 40%, highlighting the substantial gain from incorporating LMs. Second, the observed code coverage generally grows in proportion to the ratio of correctly answered questions, *i.e.*, models that are more effective at generating correct inputs tend to achieve higher coverage, providing explicit evidence that model reasoning ability translates into tangible fuzzing improvements. In a few cases, however, such as target *Sql-parser*, the ratio is higher but the coverage is not, which we attribute to the inherent randomness of fuzzing; notably, the differences remain relatively small.

Comparing across models, we observe a general hierarchy: *Qwen2.5-7B* < *Qwen2.5-32B* < *DeepSeek-V3* < *R1-Fuzz-7B* <

*GPT-o4mini*. The legends in Figure 3 are arranged from top to bottom in the order of each model's coverage performance. It shows that our RL-based training substantially enhances the base 7B model, enabling it to unlock its latent potential and rival or even surpass much larger models with more parameters. Moreover, large closed-source models incur costs with 40 USD per target for 24 hours, while our *R1-Fuzz-7B* runs efficiently on local hardware, demonstrating a favorable balance between performance and costs.

For certain special cases, for example, on target *Lua*, *Qwen2.5-7B* slightly outperforms *Qwen2.5-32B* despite their similar ratios, because the smaller model answered more total questions, underscoring the importance of model efficiency in the intensive fuzzing scenarios. For the targets *CPython* and *NJS*, *R1-Fuzz-7B* even outperforms *GPT-o4mini*, providing further evidence of the necessity of stimulating latent capabilities in low-cost LMs for fuzzing.

Comparing across targets, performance varies for all models: some have higher ratios while others remain less satisfactory. We attribute this to three possible factors: (1) the training dataset can be further improved to better capture real-world fuzzing distributions; (2) the question construction may be refined to improve the provided context; and (3) limitations may stem from the fundamental ability of the underlying base model, as they exhibit similar weaknesses. We leave these directions to future work.

In summary, the LLM-powered fuzzing loop in R1-Fuzz proves effective in real-world fuzzing, achieving significant gains over the baseline. Furthermore, our RL-based post-training enables a cost-efficient 7B model to reach or exceed the performance of larger models. These findings demonstrate both the practical effectiveness of our approach and the broader potential of RL training for unlocking the potential of LMs in fuzzing.

**5.2.2 Comparison of State-Of-The-Arts.** Next, we compared the performance of *AFL++R1-FUZZ-7B* against a general-purpose fuzzer *libFuzzer* [28] and several state-of-the-art fuzzers, including grammar-aware fuzzers (*Polyglot* [9], *Nautilus* [5], and *Gramatron* [30]) on an available subset of our selected targets.

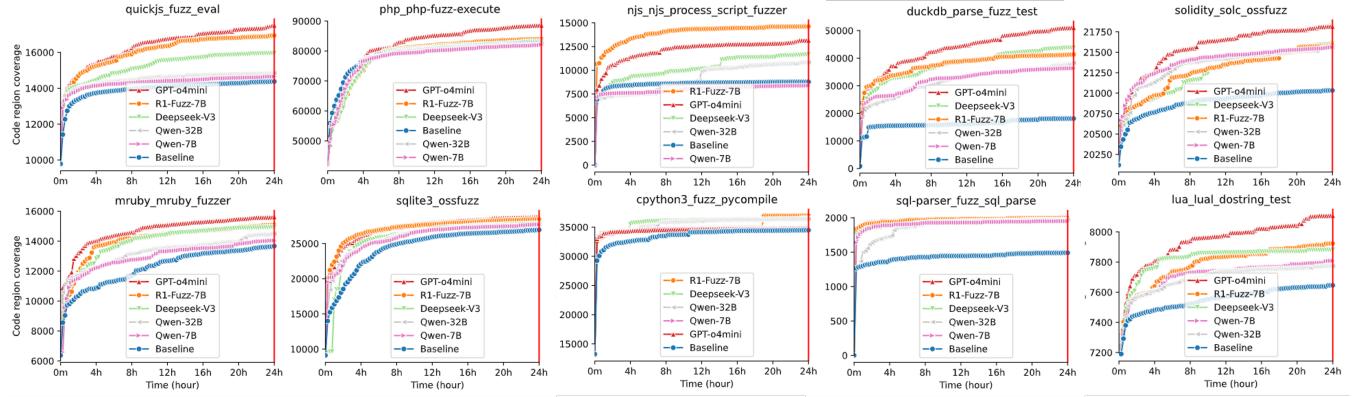


Figure 3. Code Coverage from Different Models

Table 3. Code region coverage of R1-Fuzz and other state-of-the-art fuzzers.

	PHP	mruby	NJS	QuickJS	Solidity
libFuzzer	58274	9738	4382	12865	20673
Polyglot	46768	N/A	N/A	10913	20406
Gramatron	61943	9230	7633	11550	N/A
Nautilus	57020	10102	10835	13145	N/A
AFL+++R1-Fuzz-7B	<b>83967</b>	<b>15088</b>	<b>14624</b>	<b>16941</b>	<b>21585</b>

Table 3 presents the number of covered code regions achieved by each approach. Targets not in Table 3 and entries marked *N/A* indicate cases where the open-source implementation of the grammar-based fuzzer does not provide the grammar or specification file required for the corresponding programming language. In contrast, our approach builds on the general fuzzer *AFL++* [16] without relying on handcrafted specifications, highlighting the generalization capability of our LLM-powered fuzzing loop. The results show that *AFL++R1-Fuzz-7B* consistently outperforms all compared fuzzers. The average code coverage improvement achieves nearly 75%, demonstrating the significant effectiveness of integrating LMs into the fuzzing loop (stage ③).

Table 4. Bug numbers

	PHP	Lua	mruby	NJS	QuickJS	Solidity	DuckDB	Total
AFL++	0	0	1	0	0	0	0	1
libFuzzer	0	0	1	0	0	0	0	1
Nautilus	0	N/A	1	0	1	N/A	N/A	2
Gramatron	0	N/A	0	0	0	N/A	N/A	0
Polyglot	3	N/A	N/A	N/A	0	0	N/A	3
AFL++	Qwen2.5-7B	0	1	0	0	0	0	1
	Qwen2.5-32B	2(2)	1	0	0	0	2	5
R1-Fuzz	Deepseek-V3	2	1	2	0	0	1	9
	GPT-o4mini	4	1	0	2	1(1)	0	7(3)
	R1-Fuzz-7B	6	3	2	2	3	4	23

**5.2.3 Discovery of Vulnerabilities.** We further evaluate the effectiveness of R1-Fuzz fuzzers (different models integrated with *AFL++* [16]) in vulnerability discovery. We ran all fuzzers on the latest version of our selected targets to discover previously unknown bugs. The results are shown

in Table 4. The numbers in parentheses indicate unique vulnerabilities discovered by other fuzzers or models but not by *R1-Fuzz-7B*. Traditional general fuzzers, including *AFL++* and *libFuzzer*, discovered two bugs, and state-of-the-art grammar-aware fuzzers (*Nautilus*, *Gramatron*, *Polyglot*) found five bugs in total after manual analysis. In contrast, our LLM-powered fuzzing loop in R1-Fuzz outperformed existing methods, uncovering 29 unique vulnerabilities that are previously unknown. These vulnerabilities span diverse projects, including PHP, Lua, mruby, NJS, QuickJS, Solidity, and DuckDB, with 24 confirmed or fixed by developers shown in Table 6. Compared with the non-LLM fuzzer, R1-Fuzz revealed over 5× more vulnerabilities, highlighting the practical advantage of integrating language models into fuzzing.

Among different models in our LLM-powered fuzzing loop, our post-trained model, *R1-Fuzz-7B*, outperforms both smaller and larger non-specialized models. While a few bugs remain exclusive to other models, the majority of vulnerabilities are revealed by our model. The comparison with untuned smaller models emphasizes this observation: *Qwen2.5-7B* and *Qwen2.5-32B* identified only 1 and 5 bugs, respectively, whereas *R1-Fuzz-7B* uncovered a total of 23 bugs. It demonstrates that our RL-based training effectively equips a small model with specialized ability on the target project. When combined with the efficiency of the small model, this approach proves highly effective for intensive fuzzing tasks.

In summary, the vulnerability discovery results validate the effectiveness of our LLM-powered fuzzing loop and the necessity of RL-based post-training.

### 5.3 Ablation Study

In this section, we conducted ablation experiments to assess the effectiveness of key design choices in R1-Fuzz, including the *coverage-slicing-based question construction* and the *question scheduling* design in our LLM-powered fuzzing loop.

**5.3.1 Question Format.** We first investigated the impact of function-trace extraction in the question construction

**Table 5.** Ablation study (relative difference of correctly answered ratio and code coverage)

		PHP	CPython	Lua	mruby	NJS	QuickJS	Solidity	Sqlite	Sql-parser	DuckDB
w/o Trace	Ratio	-0.5%	-12.43%	-4.17%	-2.2%	-11.92%	0.83%	-14.77%	-18.06%	-14.48%	-11.67%
w/o Prio	Cov	-3.1%	-2.4%	-3.1%	-6.0%	-6.3%	-1.6%	-4.3%	-1.7%	-4.1%	-6.2%
	Ratio	-2.7%	-0.2%	-5.1%	-2.6%	-1.1%	-1.4%	-10.3%	-1.3%	-7.4%	-5.3%

**Table 6.** Bug details

Target	Status	Link
1 PHP	Fixed	<a href="https://github.com/php/php-src/issues/18845">https://github.com/php/php-src/issues/18845</a>
2 PHP	Duplicated	<a href="https://github.com/php/php-src/issues/18844">https://github.com/php/php-src/issues/18844</a>
3 PHP	Fixed	<a href="https://github.com/php/php-src/issues/18838">https://github.com/php/php-src/issues/18838</a>
4 PHP	Fixed	<a href="https://github.com/php/php-src/issues/19303">https://github.com/php/php-src/issues/19303</a>
5 PHP	Fixed	<a href="https://github.com/php/php-src/issues/19304">https://github.com/php/php-src/issues/19304</a>
6 PHP	Fixed	<a href="https://github.com/php/php-src/issues/19305">https://github.com/php/php-src/issues/19305</a>
7 PHP	Fixed	<a href="https://github.com/php/php-src/issues/19306">https://github.com/php/php-src/issues/19306</a>
8 PHP	Fixed	<a href="https://github.com/php/php-src/issues/19844">https://github.com/php/php-src/issues/19844</a>
9 NJS	Fixed	<a href="https://github.com/nginx/njs/issues/918">https://github.com/nginx/njs/issues/918</a>
10 NJS	Fixed	<a href="https://github.com/nginx/njs/issues/921">https://github.com/nginx/njs/issues/921</a>
11 QuickJS	Fixed	<a href="https://github.com/bellard/quickjs/issues/412">https://github.com/bellard/quickjs/issues/412</a>
12 QuickJS	Fixed	<a href="https://github.com/bellard/quickjs/issues/441">https://github.com/bellard/quickjs/issues/441</a>
13 QuickJS	Duplicated	<a href="https://github.com/bellard/quickjs/issues/413">https://github.com/bellard/quickjs/issues/413</a>
14 mruby	Fixed	N/A
15 mruby	Confirmed	<a href="https://github.com/mruby/mruby/issues/6584">https://github.com/mruby/mruby/issues/6584</a>
16 Lua	Fixed	<a href="https://groups.google.com/g/lua-l/c/IJ30td8P9EY">https://groups.google.com/g/lua-l/c/IJ30td8P9EY</a>
17 Lua	Confirmed	<a href="https://github.com/ligurio/lua-c-api-tests/issues/132">https://github.com/ligurio/lua-c-api-tests/issues/132</a>
18 Lua	Confirmed	<a href="https://github.com/ligurio/lua-c-api-tests/issues/155">https://github.com/ligurio/lua-c-api-tests/issues/155</a>
19 DuckDB	Fixed	<a href="https://github.com/duckdb/duckdb/issues/17781">https://github.com/duckdb/duckdb/issues/17781</a>
20 DuckDB	Confirmed	<a href="https://github.com/duckdb/duckdb/issues/17780">https://github.com/duckdb/duckdb/issues/17780</a>
21 DuckDB	Confirmed	<a href="https://github.com/duckdb/duckdb/issues/17734">https://github.com/duckdb/duckdb/issues/17734</a>
22 DuckDB	Confirmed	<a href="https://github.com/duckdb/duckdb/issues/18448">https://github.com/duckdb/duckdb/issues/18448</a>
23 DuckDB	Confirmed	<a href="https://github.com/duckdb/duckdb/issues/18449">https://github.com/duckdb/duckdb/issues/18449</a>
24 DuckDB	Confirmed	<a href="https://github.com/duckdb/duckdb/issues/18450">https://github.com/duckdb/duckdb/issues/18450</a>
25 DuckDB	Confirmed	<a href="https://github.com/duckdb/duckdb/issues/18451">https://github.com/duckdb/duckdb/issues/18451</a>
26 DuckDB	Confirmed	<a href="https://github.com/duckdb/duckdb/issues/18452">https://github.com/duckdb/duckdb/issues/18452</a>
27 Solidity	Reported	<a href="https://github.com/ethereum/solidity/issues/16071">https://github.com/ethereum/solidity/issues/16071</a>
28 Solidity	Reported	<a href="https://github.com/ethereum/solidity/issues/16069">https://github.com/ethereum/solidity/issues/16069</a>
29 Solidity	Reported	<a href="https://github.com/argotorg/solidity/issues/16202">https://github.com/argotorg/solidity/issues/16202</a>

process. Specifically, we considered an ablation variant (*w/o Trace*) in which the `ConstructQuestion` function (see Algorithm 1 and Algorithm 2) was modified to exclude the call-stack traces of the target branch. In this setting, the constructed question body contained only the code of the immediate function associated with the uncovered branch, without additional contextual information from caller functions. To obtain an accurate comparison, we did not implement this variant as a separate fuzzing instance. Instead, within the same fuzzer, both the original question and the *w/o Trace* question were constructed for each branch and issued to the model. This design ensures that both formats are evaluated on the identical set of questions under realistic fuzzing conditions. We then calculated the *answer ratio*, *i.e.*, the fraction of correctly answered questions over the total number of questions, and reported the relative difference between the two variants. The results are presented in Table 5, which demonstrates that the absence of function-trace information generally leads to degraded performance by 8% on correct

answer ratios. Across most targets, the reduction in the answer rate confirms that providing the model with execution context—beyond the local function code—is essential for enabling effective reasoning about how to reach uncovered branches.

Besides supplying a broader execution context, we regard further exploration of more question construction strategies, which adaptively balance question validity with contextual completeness, as a valuable future direction.

**5.3.2 Question Scheduling.** We further investigated the effect of our question scheduling mechanism, which determines how constructed questions are prioritized for LM queries during fuzzing. As outlined in §3.4, R1-Fuzz continuously extracts uncovered branches to construct questions during fuzzing. Instead of immediately querying the LMs, R1-Fuzz enqueues the question into a priority queue for scheduling. We compared two variants: a naive strategy that queries each constructed question immediately (*w/o Prio*), and the default scheduling mechanism in R1-Fuzz, which assigns priorities based on the negation of the queried count. In the latter, branches that remain unreached despite repeated attempts are gradually deprioritized, thereby encouraging exploration of newly encountered branches. We implemented the variants into two fuzzing instances. Table 5 presents their relative differences in code coverage and answer ratio. The result shows that priority-based scheduling consistently achieves higher answer ratios and successfully leads to higher code coverage by the differences of 3.5% and 3.8%, respectively. This indicates that our scheduling mechanism can improve query efficiency by avoiding redundant attempts to further enhance the diversity of explored paths.

## 6 Related Work

### 6.1 LLM for Fuzzing

Since the prevalence of LLMs, there are many related works applying LLMs to the fuzzing area, including fuzzing input generation [6, 12, 13, 19, 23, 32, 34], fuzzing driver or unit test code generation [3, 10, 11, 22], and generator or mutator code generation [26, 37]. For example, PromptFuzz [22] uses LLMs to analyze library source code for generating and evolving fuzzing driver code that harnesses library APIs to test them. MetaMut [26] harnesses LLMs to generate grammar-aware mutator code to test compilers for the C language. Similarly, G2Fuzz [37] harnesses LLMs to write input generator code

for certain formats, such as PDF, ELF, and PNG, to test corresponding processors. For fuzzing input generation, Fuzz4All [34] uses LLMs to read documentation to generate feature-guided inputs. Asmita et al. [6] leverages LLMs to generate inputs for BusyBox with crash-reuse heuristics. Clozemaster [13] and CovRL-Fuzz [12] use LLMs to infill or mutate partial fuzzing input for Rust and JavaScript, respectively. Compared to them, which depend on extra documentation or the input seed itself, R1-Fuzz is the first framework to post-train small LMs to effectively reason about deep program source code for fuzzing input generation.

## 6.2 LLM Post-training

Recent works have increasingly applied reinforcement learning (RL) as a post-training paradigm to enhance the reasoning, alignment, and domain specialization of large language models. DeepSeek-R1 [15] highlights the potential of pure RL (by GRPO) in enabling LLMs to reason and benefit from chains of thought, revealing both strengths in problem-solving and challenges in controllability and safety. Compiler-R1 [27] uses RL for compiler auto-tuning, achieving reductions in intermediate representation instruction counts. Memory-R1 [36] equips LLMs with RL-trained agents for adaptive memory management and utilization. SWE-RL [33] applies RL-based reasoning to real-world software engineering by training on massive software evolution data, enabling a medium-scale model to achieve state-of-the-art performance on SWE-bench while generalizing to out-of-domain reasoning tasks. RLSF [17] leverages symbolic feedback from solvers and provers as fine-grained RL signals, enabling smaller models to surpass larger ones on program synthesis, chemistry, and logical reasoning benchmarks. Compared to them, R1-Fuzz is the first framework that demonstrates RL-based post-training is effective on practical fuzzing input generation beyond static benchmark and finds real-world vulnerabilities that are previously unknown.

## 7 Conclusion

In this paper, we introduced R1-Fuzz, a reinforcement learning-based framework for post-training language models to enhance complex textual fuzzing. R1-Fuzz addresses the key challenges of leveraging LMs for testing complex, constraint-rich software by introducing two core techniques: coverage-slicing-based question construction, which systematically decomposes deep program logic into reasonable questions, and a distance-based reward mechanism that provides fine-grained feedback during RL-based post-training. By specializing a cost-efficient 7B-parameter model rather than relying on expensive larger models, R1-Fuzz significantly improves accessibility and scalability while maintaining competitive fuzzing performance. Our evaluation demonstrates that R1-Fuzz not only achieves higher coverage—up to 75% more than state-of-the-art fuzzers—but also proves highly

practical by uncovering 29 previously unknown vulnerabilities in real-world projects. These results highlight the potential of lightweight, specialized LMs to reason about program semantics and advance the state of the art in fuzzing sophisticated textual targets.

## References

- [1] 2024. Fuzzilli in Google. <https://github.com/v8/v8/tree/main/test/fuzzilli>.
- [2] 2025. Source based code coverage. <https://clang.llvm.org/docs/SourceBasedCodeCoverage.html>.
- [3] Juan Altmayer Pizzorno and Emery D. Berger. 2025. CoverUp: Effective High Coverage Test Generation for Python. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE128 (June 2025), 23 pages. <https://doi.org/10.1145/3729398>
- [4] Abhishek Arya, Oliver Chang, Jonathan Metzman, Kostya Serebryany, and Dongge Liu. [n. d.]. OSS-Fuzz. <https://github.com/google/oss-fuzz>
- [5] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for deep bugs with grammars.. In *NDSS*, Vol. 19. 337.
- [6] Asmita, Yaroslav Oliinyk, Michael Scott, Ryan Tsang, Chongzhou Fang, and Houman Homayoun. 2024. Fuzzing BusyBox: Leveraging LLM and Crash Reuse for Embedded Bug Unearthing. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 883–900. <https://www.usenix.org/conference/usenixsecurity24/presentation/asmita>
- [7] Roberto Baldoni, Emilio Coppa, Daniele Cono D’elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.
- [8] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2329–2344.
- [9] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. 2021. One engine to fuzz’em all: Generic language processor testing with semantic validation. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 642–658.
- [10] Xiang Cheng, Fan Sang, Yizhuo Zhai, Xiaokuan Zhang, and Taesoon Kim. 2025. Rug: Turbo Llm for Rust Unit Test Generation. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 2983–2995. <https://doi.org/10.1109/ICSE55347.2025.00097>
- [11] Yinlin Deng, Chunqin Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. 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* (Seattle, WA, USA) (ISSTA 2023). Association for Computing Machinery, New York, NY, USA, 423–435. <https://doi.org/10.1145/3597926.3598067>
- [12] Jueon Eom, Seyeon Jeong, and Taekyoung Kwon. 2024. Fuzzing JavaScript Interpreters with Coverage-Guided Reinforcement Learning for LLM-Based Mutation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) (ISSTA 2024). Association for Computing Machinery, New York, NY, USA, 1656–1668. <https://doi.org/10.1145/3650212.3680389>
- [13] Hongyan Gao, Yibiao Yang, Maolin Sun, Jiangchang Wu, Yuming Zhou, and Baowen Xu. 2025. Clozemaster: Fuzzing Rust Compiler by Harnessing Llms for Infilling Masked Real Programs. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 1422–1435. <https://doi.org/10.1109/ICSE55347.2025.00175>
- [14] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*. 206–215.

- [15] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, and et al. 2025. Deepseek-R1 incentivizes reasoning in llms through reinforcement learning. *Nature* 645, 8081 (Sep 2025), 633–638. <https://doi.org/10.1038/s41586-025-09422-z>
- [16] Marc Heuse, Heiko Eifeldt, Andrea Fioraldi, and Dominik Maier. 2022. AFL++. <https://github.com/AFLplusplus/AFLplusplus>
- [17] Piyush Jha, Prithwish Jana, Pranavkrishna Suresh, Arnav Arora, and Vijay Ganesh. 2024. RLSF: Fine-tuning LLMs via Symbolic Feedback. *arXiv preprint arXiv:2405.16661* (2024).
- [18] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of hallucination in natural language generation. *ACM computing surveys* 55, 12 (2023), 1–38.
- [19] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 919–931. <https://doi.org/10.1109/ICSE48619.2023.00085>
- [20] Junlong Li, Daya Guo, Dejian Yang, Runxin Xu, Yu Wu, and Junxian He. 2025. Codei/o: Condensing reasoning patterns via code input-output prediction. *arXiv preprint arXiv:2502.07316* (2025).
- [21] Zhaowei Liu, Xin Guo, Fangqi Lou, Lingfeng Zeng, Jinyi Niu, Zixuan Wang, Jiajie Xu, Weige Cai, Ziwei Yang, Xueqian Zhao, et al. 2025. Fin-r1: A large language model for financial reasoning through reinforcement learning. *arXiv preprint arXiv:2503.16252* (2025).
- [22] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. 2024. Prompt Fuzzing for Fuzz Driver Generation. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) (CCS '24). Association for Computing Machinery, New York, NY, USA, 3793–3807. <https://doi.org/10.1145/3658644.3670396>
- [23] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, Vol. 2024.
- [24] Jonathan Metzman, László Szekeres, Laurent Maurice Romain Simon, Read Trevelin Spraberry, and Abhishek Arya. 2021. 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)*. Association for Computing Machinery, New York, NY, USA, 1393–1403. <https://doi.org/10.1145/3468264.3473932>
- [25] OpenAI. [n. d.]. Introducing OpenAI o3 and o4-mini – openai.com. <https://openai.com/index/introducing-o3-and-o4-mini/>.
- [26] Xianfei Ou, Cong Li, Yanyan Jiang, and Chang Xu. 2025. The Mutators Reloaded: Fuzzing Compilers with Large Language Model Generated Mutation Operators. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4* (Hilton La Jolla Torrey Pines, La Jolla, CA, USA) (ASPLOS '24). Association for Computing Machinery, New York, NY, USA, 298–312. <https://doi.org/10.1145/3622781.3674171>
- [27] Haolin Pan, Hongyu Lin, Haoran Luo, Yang Liu, Kaichun Yao, Libo Zhang, Mingjie Xing, and Yanjun Wu. 2025. Compiler-R1: Towards Agentic Compiler Auto-tuning with Reinforcement Learning. *arXiv preprint arXiv:2506.15701* (2025).
- [28] Kosta Serebryany. 2016. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 157–157.
- [29] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2025. Hybrid-flow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*. 1279–1297.
- [30] Prashast Srivastava and Mathias Payer. 2021. Gramatron: Effective grammar-aware fuzzing. In *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*. 244–256.
- [31] Qwen Team. 2024. Qwen2.5: A Party of Foundation Models. <https://qwenlm.github.io/blog/qwen2.5/>
- [32] Jincheng Wang, Le Yu, and Xiapu Luo. 2024. LLMIF: Augmented Large Language Model for Fuzzing IoT Devices. In *2024 IEEE Symposium on Security and Privacy (SP)*. 881–896. <https://doi.org/10.1109/SP54263.2024.00211>
- [33] Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I Wang. 2025. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449* (2025).
- [34] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (Lisbon, Portugal) (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 126, 13 pages. <https://doi.org/10.1145/3597503.3639121>
- [35] Tian Xie, Zitian Gao, Qingnan Ren, Haoming Luo, Yuqian Hong, Bryan Dai, Joey Zhou, Kai Qiu, Zhirong Wu, and Chong Luo. 2025. Logic-rl: Unleashing llm reasoning with rule-based reinforcement learning. *arXiv preprint arXiv:2502.14768* (2025).
- [36] Sikuan Yan, Xiufeng Yang, Zuchao Huang, Ercong Nie, Zifeng Ding, Zonggen Li, Xiaowen Ma, Hinrich Schütze, Volker Tresp, and Yumpu Ma. 2025. Memory-R1: Enhancing Large Language Model Agents to Manage and Utilize Memories via Reinforcement Learning. *arXiv preprint arXiv:2508.19828* (2025).
- [37] Kunpeng Zhang, Zongjie Li, Daoyuan Wu, Shuai Wang, and Xin Xia. 2025. Low-Cost and Comprehensive Non-textual Input Fuzzing with LLM-Synthesized Input Generators. *arXiv preprint arXiv:2501.19282* (2025).