ELFUZZ: Efficient Input Generation via LLM-driven Synthesis Over Fuzzer Space

Chuyang Chen
The Ohio State University
chen.13875@osu.edu

Brendan Dolan-Gavitt
New York University
brendandg@nyu.edu

Zhiqiang Lin
The Ohio State University
zlin@cse.ohio-state.edu

Abstract

Generation-based fuzzing produces appropriate testing cases according to specifications of input grammars and semantic constraints to test systems and software. However, these specifications require significant manual efforts to construct. This paper proposes a new approach, ELFUZZ (Evolution Through Large Language Models for Fuzzing), that automatically synthesizes generation-based fuzzers tailored to a system under test (SUT) via LLM-driven synthesis over fuzzer space. At a high level, it starts with minimal seed fuzzers and propels the synthesis by fully automated LLM-driven evolution with coverage guidance. Compared to previous techniques, ELFUZZ can 1) seamlessly scale to SUTs of real-world sizes (up to 1,791,104 lines of code in our evaluation) and 2) synthesize efficient fuzzers that catch interesting grammatical structures and semantic constraints in a human-understandable way. Our evaluation shows that ELFUZZ achieves up to 418.5% more coverage and triggers up to 216.7% more artificially injected bugs, compared to the state-of-the-art. We also used ELFUZZ to conduct a real-world fuzzing campaign on the newest version of `cvc5` for 14 days, and encouragingly, it found five 0-day bugs (three are exploitable). Moreover, our ablation study shows that fuzzer space, the key component of ELFUZZ, contributes the most (up to 62.5%) to the effectiveness of ELFUZZ. Further analysis of the fuzzers synthesized by ELFUZZ confirms that they correctly express the grammatical structures and semantic constraints of a SUT. The results present a promising potential of ELFUZZ for more automated, efficient, and extensible input generation for fuzzing.

1 Introduction

Over the past decade, fuzzing has proven to be a highly effective technique for finding vulnerabilities in critical software and systems such as the Linux kernel, OpenSSL, FFmpeg, and SQLite [1, 2, 54]. However, fuzzing systems under test (SUTs) [10] that consume complex text formats is particularly challenging, as they impose rigorous grammatical and semantic checks on the inputs. A specific fuzzing technique that targets these SUTs is generation-based fuzzing, where test cases are generated according to the specifications of the SUT to ensure that they pass the early-stage checks [21, 28, 54]. Then, the generated test cases can be fed to a mutation-based fuzzer (like AFL++ [34]) as seeds to reach deeper into the codebase. Unfortunately, constructing such specifications is extremely expensive. For example, CSMITH [73], a notable generation-based fuzzer targeting C compilers, consists of 80k lines of code (LoC) to meet the requirements of the C99 standard [30, 71]. Putting similar labor for every SUT in different domains would be impractical.

The problem. Automatic synthesis of these specifications, usually expressed as grammars with semantic constraints, is a long-standing difficult problem [44, 63]. Various approaches, including data-flow and control-flow analyses, have been proposed for synthesizing grammars [38, 41], as well as template-based techniques for synthesizing semantic constraints [62]. However, these approaches are far from being practical for real-world SUTs due to the following two reasons:

- **Scalability.** Existing techniques rely on complex program analyses to reconstruct input grammars, but these analyses often lack engineering and algorithmic scalability.
- **Efficiency.** Even provided with a pre-existing grammar, a fuzzer has to instantiate the grammatical rules to concrete test cases, which causes significant overhead that reduces fuzzing efficiency.

Our solution. To advance the state of the art, this paper introduces ELFUZZ (Evolution Through Large Language Models for Fuzzing), which employs a large language model (LLM) to conduct LLM-driven synthesis over fuzzer space (explained later) to synthesize generation-based fuzzers tailored to a SUT. ELFUZZ starts with a naïve seed fuzzer that generates purely random texts and propels the synthesis by fully automated LLM-driven evolution with fuzzer space guidance. Compared to previous techniques, ELFUZZ can 1) seamlessly scale to SUTs of a real-world size (up to 1,791,104 LoC in our evaluation) and 2) synthesize input generators that directly embed grammars with semantic constraints, eliminating the overhead associated with grammar rule instantiation.

LLM-driven evolution. ELFUZZ synthesizes the fuzzers via an LLM-driven evolution loop. The loop starts from a naïve seed fuzzer and gradually improves towards superior fuzzers. In each evolution iteration, ELFUZZ queries the LLM to mutate the current survivors. The knowledge about the SUT learned in training enables the LLM to conduct natural, human-like modifications (e.g., reworking several lines while maintaining consistency with the context). Later, the mutants will be assessed according to their coverage information, and the best ones will be chosen for the next iteration. By gradually improving, the challenging task of synthesizing a whole fuzzer is divided into much simpler, small steps. Therefore, ELFUZZ can leverage the coding capability of the LLM to "conquer" these tractable steps.

Fuzzer space guidance. The evolution loop is guided by coverage information consisting of the exact range of the covered code. Not like coverage represented by a single value that coarsely "equalizes" fuzzers covering the same *size* of code, ELFUZZ compares the candidate fuzzers (produced by mutation during the evolution) in a fine-grained way by the *ranges* of code they cover. The fuzzer space structures the candidate fuzzers accordingly into a lattice, namely, the *fuzzer space*, which is formed by the partial order relation between the range of the covered code of each fuzzer. Fuzzers that cover the same code are essentially equivalent, while those that cover more code are superior; incomparable ones (possibly with the same coverage value), on the contrary, test different parts of the SUT and cannot be substituted by one another. Fuzzer space enables fine-grained comparison of the fuzzers, which is impossible if using a single coverage value, where two fuzzers incomparable in the fuzzer space can have the same coverage value and be wrongly regarded as equivalent. Through such evaluation and comparison, the fuzzer space guides the evolution loop toward better fuzzers.

Encouraging results. We have implemented and evaluated ELFUZZ. Our evaluation shows that fuzzers synthesized by ELFUZZ achieve 418.5% more coverage than state-of-the-art techniques and trigger up to 216.7.0% more artificially injected bugs. In a real-world bug-finding experiment, it found five 0-day bugs of `cvc5`, three of which are exploitable vulnerabilities. Note that two of the bugs have been fixed before our responsible disclosure, and we, therefore, only reported the other three bugs to the developers. An ablation study shows that the fuzzer space model contributes the most (up to 62.5%) to the effectiveness of ELFUZZ. Two case studies further demonstrate that the fuzzers synthesized by ELFUZZ catch interesting grammatical structures and semantic constraints in an interpretable way and can be easily extended with other generation-based fuzzing techniques.

Contributions. We make the following contributions:

- **Novel notion (§3).** We present the notion of fuzzer space to model the task of synthesizing generation-based fuzzers. The notion of fuzzer space provides a formal way to analyze and guide the synthesis process.
- **Innovative approach (§4).** We propose LLM-driven synthesis over fuzzer space to synthesize effective and efficient generation-based fuzzers that produce initial seeds for a mutation-based fuzzer, which further mutates them to reach deeper code. The synthesis approach is based on an LLM-driven evolution loop, which decomposes the target of synthesizing a complex fuzzer into tractable, small steps and leverages an LLM to make gradual improvements guided by the fuzzer space. We have implemented our approach as ELFUZZ and made it publicly available online (download link in the open science statement).
- **Empirical evaluation (§6).** We have evaluated ELFUZZ on seven widely used benchmarks. It shows that ELFUZZ outperforms the state-of-the-art techniques. The initial seeds provided by ELFUZZ significantly boost later mutation-based fuzzing. Future research may investigate how to extend the LLM-driven evolution to the entire fuzzing cycle to maintain corpus diversity throughout the whole process.

2 Background and Related Work

Grammar-based fuzzing. Fuzzing (or fuzz testing) triggers bugs and vulnerabilities in SUTs via generating numerous random test cases [21, 28, 53, 54, 74, 78, 79]. While mutation-based fuzzing derives new test cases from existing ones, generation-based fuzzing synthesizes them according to specifications that define the acceptable structures [21, 36, 39, 60, 62] of the inputs to a SUT. Hardcoding such specifications in a fuzzer results in a staggering workload: CSMITH, a notable generation-based fuzzer, consists of 80k LoC to meet the requirements of the C99 standard [30, 49, 71].

Grammar-based fuzzing reduces the workload by developing a general fuzzing framework that can produce test cases for an arbitrary SUT with the specification of its inputs expressed in a domain-specific language (DSL), typically using Backus-Naur form grammars. Thus, the user only needs to provide the specification, rather than writing a complete fuzzer. However, challenges exist even in these specifications themselves:

- **Obtaining the grammar.** While DSLs mitigate the coding burden, using them to express grammars still demands considerable expertise and domain knowledge.
- **Semantic constraints.** Some input formats require semantic constraints beyond the expressive power of a grammar, such as the "define-before-use" rule in C/C++. The absence of a simple and efficient way to impose such semantic constraints impairs the quality of the generated test cases.

Grammar synthesis techniques are proposed to resolve the first challenge [38, 41]. However, the complex static and dy-

Marker	Challenge
❶	Considering input grammars
❷	Considering semantic constraints
❸	Scaling to real-world SUTs
❹	Preventing grammar rule instantiation
❺	Human-understandable fuzzing process
❻	Easily generalizable to different domains
❼	No need for domain expertise

Table 1: Challenges for effective input generation

Approach	Year	❶	❷	❸	❹	❺	❻	❼
AUTOGRAM [41]	2016	✓				✓	✓	✓
LEARN&FUZZ [37]	2017	✓	✓	✓	✓			✓
GLADE [24]	2017	✓		✓		✓	✓	✓
MIMID [38]	2020	✓				✓	✓	✓
ISLEARN [62]	2022		✓			✓		
FUZZNG [27]	2023	✓	✓	✓	✓	✓		
TITANFUZZ [31]	2023	✓	✓	✓	✓			
FUZZ4ALL [71]	2024	✓	✓	✓	✓			✓
METAMUT [57]	2024	✓	✓	✓	✓	✓		
DY FUZZING [23]	2024	✓	✓	✓	✓	✓		
SYZGEN++ [29]	2024	✓	✓	✓	✓	✓		
COVRL-FUZZ [33]	2024	✓	✓	✓	✓			
FUZZINMEM [51]	2024	✓	✓	✓	✓			
ELFUZZ		✓	✓	✓	✓	✓	✓	✓

Table 2: Comparison with works targeting the challenges

namic analyses they used limit their engineering and algorithmic scalability. GLADE [24] claims to overcome these problems. Yet, a replication study [26] raises doubts about the reliability of the performance reported in the paper. The challenge of semantic constraints gains less attention. ISLA [62] introduces an SMT-based DSL to express and solve the constraints, but the DSL is of restricted expressive power. Complementing ISLA, ISLEARN proposed in the same paper mines constraints in the DSL if the templates apply.

Fuzz driver synthesis. Fuzz drivers take test cases generated by a fuzzer and feed them into APIs of a library SUT. They can be crucial for the fuzzing effectiveness for libraries, and many works aim to synthesize fuzz drivers automatically [42, 48, 75, 76]. This line of research is parallel to the target of ELFUZZ: A fuzz driver aims to feed given test cases into a SUT, while fuzzers synthesized by ELFUZZ aim to generate inputs to explore the SUT through a given fuzz driver. Thus, the techniques used in these works cannot be adapted to our purpose.

Fuzzing by LLM. The emergence of LLMs has opened up unprecedented opportunities and challenges for cybersecurity research [40, 50, 56, 65, 72]. LLMs overcome the shortcomings of previous deep learning-based fuzzing techniques (which require re-training the model on massive samples for every new SUT [22, 37, 67]) with their state-of-the-art zero-shot and few-shot learning abilities. However, fuzzing directly via LLMs poses high requirements for GPU usage during the entire fuzzing process and lacks interpretability and extensibility.

Challenges. Table 1 summarizes the mentioned challenges for efficient input generation, and Table 2 lists recent works

```python
def balanced_parenthesis(parens: str) -> bool:
    stack = []
    for c in parans:
        if c == "(":
            stack.insert(0, c)
        elif c == ")":
            if stack and stack[0] == "(":
                stack.pop(0)
            else:
                return False
        else:
            raise ValueError("Invalid character")
    return not stack
```

Listing 1: Running example

that target these challenges. However, none of these techniques resolves them all. Therefore, these techniques cannot fulfill automated, efficient, and interpretable input generation.

Our insights. ELFUZZ overcomes all the challenges through several novel designs:

- **High efficiency by no grammar instantiation**. Instead of grammars, ELFUZZ synthesizes fuzzers tailored for each SUT, which skip parsing and instantiating grammar rules and generate test cases directly. The related overhead is thus avoided.
- **Input generation by code instead of LLMs.** ELFUZZ provides interpretable and extensible fuzzers written in Python instead of relying on an LLM to generate test cases in a black-box manner.
- **Decomposing the synthesis via evolution.** Considering the complexity of the task, synthesizing fuzzers could be challenging for LLMs despite their good performance on coding benchmarks [66, 75]. Thus, ELFUZZ adopts an evolution loop to decompose the synthesis into tractable, small steps. Such a strategy reduces the need for complex prompt engineering and powerful yet resource-intensive "big models" with a massive number of parameters.
- **Fuzzer space guidance.** The evolution loop is guided by a lattice structure, namely, the fuzzer space, which is derived from the coverage information consisting of the covered code range of the candidate fuzzers. The fuzzer space guidance enables fine-grained evaluation of fuzzers while maintaining scalability.

3 The Notion of Fuzzer Space

Fuzzer space structures fuzzers into a lattice that characterizes their effectiveness (or *strength*). The process of synthesizing good fuzzers is modeled as exploration from the bottom (less effective generators) to the top (more effective generators) of the fuzzer space. We illustrate these concepts using the example SUT in Listing 1.

Cover set. For a given SUT, one important metric for characterizing fuzzers is the range of the code they cover, referred to as the *cover set*. Two fuzzers with the same cover set are

```python
def fuzzer_fa(random: Random) -> str:
    random_sequence = ""
    while len(random_sequence) < MAX_LEN:
        choice = random.randint(0, CHOICE_RANGE)
        if choice == 0:
            random_sequence += "("
        elif choice == 1:
            random_sequence += ")"
        elif choice == 2:
            random_sequence += "*"
        else:
            break
    return random_sequence
```

Listing 2: Fuzzer $\mathrm{F_A}$ covering all code of Listing 1

```python
def fuzzer_fa_prime(random: Random) -> str:
    random_len = random.randint(0, MAX_LEN)
    random_sequence = ""
    for _ in range(random_len):
        choice = random.randint(0, CHOICE_RANGE)
        if choice == 0:
            random_sequence += "("
        elif choice == 1:
            random_sequence += ")"
        else:
            random_sequence += "*"
    return random_sequence
```

Listing 3: Fuzzer $\mathrm{F'_A}$ equivalent to $\mathrm{F_A}$

```python
def fuzzer_fb(random: Random) -> str:
    random_sequence = ""
    while len(random_sequence) < MAX_LEN:
        choice = random.randint(0, CHOICE_RANGE)
        if choice == 0:
            random_sequence += "("
        else:
            break
    return random_sequence
```

Listing 4: Fuzzer $\mathrm{F_B}$ weaker than $\mathrm{F_A}$

```python
def fuzzer_fc(random: Random) -> str:
    random_sequence = ""
    while len(random_sequence) < MAX_LEN:
        choice = random.randint(0, CHOICE_RANGE)
        if choice == 0:
            random_sequence += ")"
        elif choice == 1:
            random_sequence += "*"
        else:
            break
    return random_sequence
```

Listing 5: Fuzzer $\mathrm{F_C}$ also weaker than $\mathrm{F_A}$

considered equivalent. For example, fuzzer $\mathrm{F_A}$ in Listing 2 and fuzzer $\mathrm{F'_A}$ both generate random sequences of left and right parentheses and asterisks and cover lines 1–13 of the SUT. Thus, their fuzzing effectiveness is the same. Note that the illustration uses line coverage; other forms of coverage are possible in implementation, such as edge coverage [34, 68]. We use $\mathcal{C}(\cdot)$ to denote the cover set of a fuzzer.

The strength of fuzzers. The subset relation between cover sets defines the relative strength of fuzzers, as formalized in Definition 1. Intuitively, a proper superset relation of one cover set over another indicates strictly better fuzzing effectiveness. The strength of fuzzers forms a partial order, where two fuzzers can sometimes be incomparable. This occurs when the two fuzzers have distinct strengths and their cover sets cannot fully encompass each other. Listing 2 and Listing 4 illustrate different strengths of fuzzers. Fuzzer $\mathrm{F_B}$ generates sequences of only left parentheses, covering only lines 1–5 and 13, wich is a subset of the cover set of $\mathrm{F_A}$ (lines 1–13). Thus, $\mathrm{F_B}$ is weaker than $\mathrm{F_A}$.

Definition 1 (Strength of fuzzers)**.** Fuzzer F_1 is stronger than fuzzer F_2 if $\mathcal{C}(F_2) \subset \mathcal{C}(F_1)$, denoted as

$$\mathcal{S}(F_2) \sqsubset \mathcal{S}(F_1)$$

where $\mathcal{S}(\cdot)$ is the strength of a fuzzer.

Fuzzer space. Fuzzers with the partial order $\mathcal{S}(\cdot) \sqsubset \mathcal{S}(\cdot)$ forms a lattice, i.e., the fuzzer space. Fuzzer space enables fine-grained comparison between fuzzers. Such a comparison is impossible if using a single coverage value, which may coarsely equalize two fuzzers with distinct strengths. Intuitive correspondence exists between the lattice structure and the original fuzzers: the top element $\top$ represents the theoretically most powerful fuzzer, capable of reaching every line of code in the SUT, while the bottom element $\bot$ represents the weakest one. Figure 1 presents the fuzzer space consisting of six fuzzers, $\mathrm{F_A}$ (Listing 2), $\mathrm{F'_A}$ (Listing 3), $\mathrm{F_B}$ (Listing 4), $\mathrm{F_C}$ (Listing 5), $\mathrm{F_D}$ (Listing 6), and $\mathrm{F_E}$ (Listing 7). The lattice structure clearly shows the effectiveness of each fuzzer. For example, $\mathrm{F_C}$ is strictly stronger than $\mathrm{F_D}$, as it can cover all code covered by $\mathrm{F_D}$ (the grey and red boxes) plus code that $\mathrm{F_D}$ cannot cover (the yellow box). However, $\mathrm{F_B}$ and $\mathrm{F_C}$ cannot be thus compared, as they test different parts of the SUT. Note that if we use a single coverage value to assess the fuzzers, $\mathrm{F_B}$ may be considered as replaceable by $\mathrm{F_C}$, as its line coverage (which is 6) is smaller than that of $\mathrm{F_C}$ (which is 11).

In real-world scenarios, generation-based fuzzers are only run for a finite time, and it is impractical and unnecessary to obtain the theoretically precise cover sets. Thus, the code covered by a fuzzer in a limited time span is used as an approximation of its cover set.

4 LLM-driven Synthesis Over Fuzzer Space

At a high level, ELFUZZ leverages an LLM to motivate an evolution loop to improve the seed fuzzer iteratively, and the fuzzer space steers the evolution loop towards superior fuzzers. Compared to evolutionary algorithms traditionally relying on random tree mutations [43], the domain knowledge embedded in the LLM enables more reasonable and human-like modifications on the code of the candidate fuzzers [47]. Then, the mutants are evaluated and placed into the fuzzer space according to their strength. Superior fuzzers closer to the top are chosen as survivors, while degenerate ones are discarded. Thus, the evolution loop "climbs" the fuzzer space

```python
def fuzzer_fd(random: Random) -> str:
    random_sequence = ""
    while len(random_sequence) < MAX_LEN:
        choice = random.randint(0, CHOICE_RANGE)
        elif choice == 0:
            random_sequence += "*"
        else:
            break
    return random_sequence
```

Listing 6: Fuzzer F_D weaker than F_C

```python
def fuzzer_fe(random: Random) -> str:
    return ''
```

Listing 7: Fuzzer F_E generating only empty sequences

step by step, iteratively advancing toward the optimal solution. Figure 2 illustrates this overall process.

Specifically, every iteration of the evolution loop comprises three steps as shown by Figure 3, each detailed by a subsection:

- **LLM-driven mutation (§4.1).** At this step, the candidate fuzzers selected in the previous iteration as seed fuzzers are mutated by an LLM to produce mutants that may or may not augment the exploration of the fuzzer space.
- **Fuzzer space exploration (§4.2).** At this step, the mutants are added to the already explored part of the fuzzer space according to their relative strength compared to the seed fuzzers. Invalid mutants (e.g., containing type errors) and mutants weaker than the seed fuzzers are discarded.
- **Max-cover selection (§4.3).** At this step, mutants with the maximum unioned cover set are selected from the fuzzer space as seed fuzzers for the next iteration. The selection prevents the population of candidate fuzzers from growing explosively.

4.1 LLM-driven Mutation

At the start of each iteration, the seed fuzzers are mutated by an LLM to produce mutants. The process is depicted in the leftmost part of Figure 3, where green boxes are the seed fuzzers and white boxes are the mutants. ELFUZZ employs three LLM-driven mutators [47] to conduct the mutation. These mutators leverage the advanced capabilities of the LLM in code generation, modification, and comprehension to apply human-like, syntactically correct, and semantically meaningful modifications. The LLM increases the likelihood of valid and diverse mutants and reduces the time spent on invalid or irrelevant ones. The details of the three mutators are provided below:

- **Splicing.** The splicing mutator involves combining the prefix of one candidate fuzzer and the suffix of another with glue code produced via the LLM's fill-in-the-middle ability [25]. Splicing is crucial since it allows the fusion of different candidate fuzzers to join their strengths. It serves the role of crossover mutators in traditional evolutionary algorithms [46, 69]. By gluing two fuzzers F_1 and F_2, the LLM may discover a more powerful mutant (like $F_1 \sqcup F_2$) while keeping the population small.
- **Completion.** The completion mutator involves truncating a candidate fuzzer at a random point and having the LLM complete the remaining part. Completion exploits LLM's predictive capabilities to continue an incomplete program. Note that truncation is needed because the LLM sometimes sloppily ends an incomplete program. The completion mutator enables the LLM to improve the candidate fuzzer based on previous code snippets. It mimics human programmers who implement a program line by line.
- **Infilling.** The infilling mutator involves removing random lines from a candidate fuzzer and utilizing the LLM's fill-in-the-middle capability to rewrite them. Infilling allows the LLM to rework parts of the code and potentially redirect the fuzzer to previously undiscovered code. Moreover, the code before and after the removed lines serves as the context for the LLM to maintain the consistency of the program [70]. Infilling mimics human programmers who revise or refactor their programs.

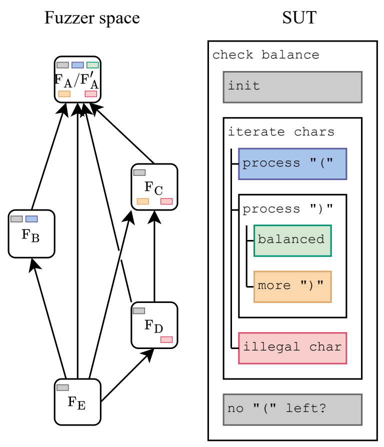


Figure 1: An illustration of Listing 1's fuzzer space

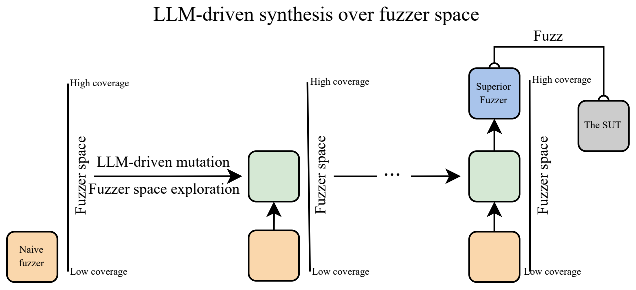


Figure 2: The workflow of ELFUZZ

We proceed to illustrate the three mutators by examples. Suppose that the previous fuzzers F_B (Listing 4) and F_D (List-

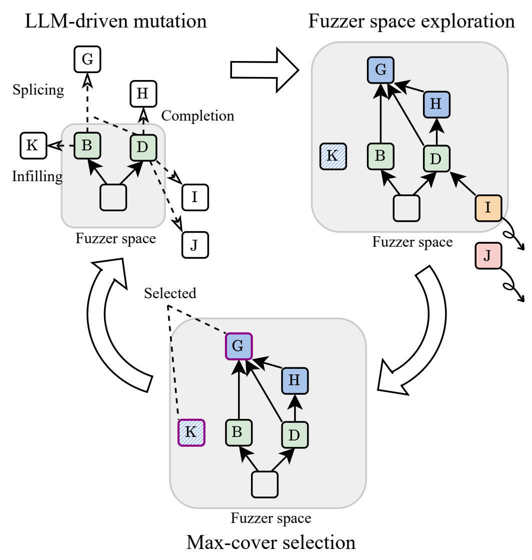
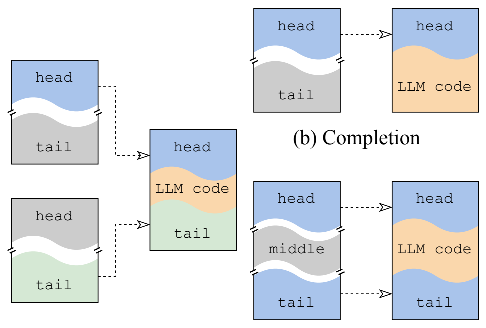


Figure 3: An evolution iteration of ELFUZZ



(b) Completion

(a) Splicing

(c) Infilling

Figure 4: LLM-driven mutators

ing 6) are the seed fuzzers. The splicing mutator is applied to glue the code before line 7 of F_B and the code after line 5 of F_D. The completion mutator is used to continue writing F_D after removing the code after line 8. The infilling mutator is used to rewrite line 6 of F_B. The splicing mutator will cut F_B at line 7, keeping the head and discarding the tail. The same happens to F_D at line 5, but discarding the head and keeping the tail. Then, the head of F_B and the tail of F_D will be glued by the LLM to get the fuzzer F_G in Listing 8 where line 7 is the junction. The completion mutator will remove line 9 of F_D and re-continue the code starting from this line. The fuzzer F_K in Listing 9 is the resultant mutant, while line 10 is the re-completed code by LLM. The infilling mutator will remove line 6 of F_B, resulting in a "hole" in the program, and the LLM will fill the hole with new code and produce the fuzzer F_H in Listing 10, where line 7 is the code that replaces the original line 6.

The LLM-driven mutators produce vast amounts of mutants, but not all of them are helpful for our synthesis. For example, a mutant can be weaker than the seed fuzzers before mutation like F_I in Listing 11 or contain type errors like F_J

```
1  def fuzzer_fg(random: Random) -> str:
2      random_sequence = ""
3      while len(random_sequence) < MAX_LEN:
4          choice = random.randint(0, CHOICE_RANGE)
5          if choice == 0:
6              random_sequence += "("
7          elif choice == 1: # LLM glue code
8              random_sequence += "*"
9          else:
10             break
11     return random_sequence
```

Listing 8: Fuzzer F_G stronger than the seed fuzzers

```
1  def fuzzer_fk(random: Random) -> str:
2      random_sequence = ""
3      while len(random_sequence) < MAX_LEN:
4          choice = random.randint(0, CHOICE_RANGE)
5          if choice == 0:
6              random_sequence += "("
7          else:
8              break
9      # Code re-completed by the LLM
10     return ")" + random_sequence
```

Listing 9: Fuzzer F_K discovering new code

in Listing 12. Fuzzer space exploration, detailed in the next subsection, will distinguish the mutants that improve the seed fuzzers and those that degenerate.

4.2 Fuzzer Space Exploration

Mutants generated by LLM-driven mutation are then collected to augment the exploration of the fuzzer space, as depicted in the central part of Figure 3. The strength of the mutants and the seed fuzzers is compared to determine the lattice structure between them. Mutants weaker than the seed fuzzers (the orange box) and those containing errors (the red box) are discarded, while those stronger than the seed fuzzers (the blue boxes) or those covering new code (the hatched blue box) survive. This way, only mutants contributing to the "climb" toward the top are kept.

Algorithm 1 approximates the cover set of a mutant and use the mutant to augment the explored part of the fuzzer space accordingly. A detailed explanation of the algorithm is as follows:

- Lines 2 and 3 select a mutant fuzzer m and invoke the APPROXCOV subroutine to approximate its cover set;
- Lines 4–9 compare the cover set of m with existing fuzzers. It is added to the fuzzer space if being stronger than an existing fuzzer f and the relation $\mathcal{S}(f) \sqsubset \mathcal{S}(m)$ is recorded by an arrow from f to m;
- Lines 13 and 14 in the APPROXCOV subroutine approximate the cover set of a fuzzer by using it to fuzz the SUT in a limited time span. As mentioned in §3, obtaining the precise cover sets in real-world scenarios is impractical and unnecessary.

Take the seed fuzzers F_B, F_D and the mutants F_H–F_K in the previous subsection as an example. After running for

```python
def fuzzer_fh(random: Random) -> str:
    random_sequence = ""
    while len(random_sequence) < MAX_LEN:
        choice = random.randint(0, CHOICE_RANGE)
        elif choice == 0:
            # Code rewritten by the LLM
            random_sequence += "("
        else:
            break
    return random_sequence
```

Listing 10: Fuzzer $\mathrm{F_H}$ also stronger than the seed fuzzers

```python
def fuzzer_fi(random: Random) -> str:
    random_sequence = ""
    while len(random_sequence) < MAX_LEN:
        choice = random.randint(0, CHOICE_RANGE)
        elif choice == 0:
            # Code weakened by the LLM
            random_sequence += ""
        else:
            break
    return random_sequence
```

Listing 11: Fuzzer $\mathrm{F_I}$ weaker than the seed fuzzers

T time, $\mathrm{F_J}$ triggers type errors and is thus discarded. The cover sets of the other fuzzers are: 1) lines 1–5 and 13 for $\mathrm{F_B}$ producing `"("`, `"(("` and `""`, 2) lines 1–3 and 11–13 for $\mathrm{F_D}$ producing `"*"`, `"**"` and `""`, 3) lines 1–6 and 11–13 for $\mathrm{F_G}$ producing `"("`, `"*"` and `""`, 4) lines 1–3 and 13 for $\mathrm{F_I}$ producing `""`, and 5) lines 1–7 and 9–10 for $\mathrm{F_K}$ producing `"("`, `")"` and `")("`.

According to the subset relations between these cover sets, $\mathrm{F_G}$ and $\mathrm{F_H}$ (the blue boxes in Figure 3) are added to the explored part of the fuzzer space where edges from $\mathrm{F_B}$ and $\mathrm{F_D}$ indicate that the two new fuzzers ($\mathrm{F_G}$ and $\mathrm{F_H}$) are stronger than the seed fuzzers ($\mathrm{F_B}$ and $\mathrm{F_D}$). $\mathrm{F_K}$ (the hatched blue box) is also added, as it has distinct strengths, but no edge connects to it since its cover set cannot fully encompass the cover set of any other fuzzer, and vice versa. $\mathrm{F_I}$ is a fuzzer even weaker than the seed fuzzers and cannot contribute to our target of synthesizing stronger fuzzers. It is thus discarded.

4.3 Max-cover Selection

As more and more parts of the fuzzer space are discovered, the seed fuzzers to be mutated will increase exponentially if not filtered. Therefore, we select a fixed-size group of elites from the mutants retained after fuzzer space exploration as the seed fuzzers for the next iteration to avoid such an explosion.

The strategy is to maximize the union of the cover sets of the elites, as illustrated by the rightmost part of Figure 3. Specifically, when selecting N elites to advance to the next iteration from a pool V of $M > N$ mutants, we choose a subset $E \subset V$ according to the following formula:

$$E = \underset{S \in \mathcal{P}_N(V)}{\arg\max} \left| \bigcup \{ C(F) | F \in S \} \right|$$

where $\mathcal{P}_N(V)$ denotes all N-element subsets of V.

```python
def fuzzer_fj(random: Random) -> str:
    random_sequence = ""
    while len(random_sequence) < MAX_LEN:
        choice = random.randint(0, CHOICE_RANGE)
        elif choice == 0:
            # Type error introduced by the LLM
            random_sequence += 1
        else:
            break
    return random_sequence
```

Listing 12: Fuzzer $\mathrm{F_J}$ containing a type error

```
procedure EXPLORE_{SUT,T}(fuzzerSpace, mutants) begin
    for $m \leftarrow$ mutants do
        $\mathrm{cov}_m \leftarrow \mathrm{APPROXCOV}_{SUT,T}(m)$
        for $f \leftarrow$ fuzzerSpace do
            if $\mathrm{cov}_f \subset \mathrm{cov}_m$ then
                fuzzerSpace.add_fuzzer($f, m$)
                fuzzerSpace.add_arrow($f, m$)
            end
        end
    end
end
procedure APPROXCOV_{SUT,T}(fuzzer) begin
    fuzzer.fuzz($SUT, T$)
    return fuzzer.coveredCode
end
```

Algorithm 1: Exploring the fuzzer space

The optimization problem described above is the well-known set cover problem [17], which is $\mathcal{NP}$-complete. Instead of computing an exact solution, we use an approximation algorithm, detailed in Algorithm 2, to find a near-optimal solution. The algorithm repeats greedy optimization from random start points for T times (lines 2–6) and chooses the best solution (line 7). The greedy optimization (lines 9–21) keeps substituting elements in the candidate solution with one that can increase the unioned cover set until no such substitution exists. The greedy optimization could be trapped in local optima, but repeating it multiple times and choosing the best mitigates the probability of being trapped in local optima. The time complexity of this approximation algorithm is $O\left(TM^2\right)$.

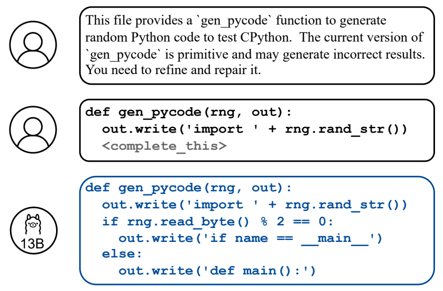
5 Implementation

We have implemented ELFuzz in Python and made the code publicly available online. The core logic for the evolutionary algorithm and fuzzer space exploration consists of 3,602 lines of code, which is lightweight compared to traditional generation-based fuzzers like Csmith [73] (80k LoC), the state-of-the-art grammar synthesizer Mimid [38] (13k LoC), and the semantic constraints synthesizer ISLearn [62] (10k LoC). Details of our implementation are described below.

Selection of the LLM. ELFuzz is LLM-agnostic. The LLM-driven evolution loop keeps the same, whatever LLM is used to conduct the mutation. Due to resource limits, our implementation uses a local CodeLLama model with 13 billion parameters. It is a rather small model compared to those well-

```
procedure ApproxMax$_{M,N,T}$(covSets) begin
    attempts $\leftarrow \emptyset$
    for $i \leftarrow 1,\dots,T$ do
        attempt$_i$ $\leftarrow$GreedyMax$_{M,N}$(covSets)
        attempts $\leftarrow$ attempts $\cup$ {attempt$_i$}
    end
    return BestOf(attempts)
end
procedure GreedyMax$_{M,N}$(covSets) begin
    attempt $\leftarrow$RandPick$_N$(covSets)
    while attempt.changed do
        for s $\in$ attempt do
            for s$'$ $\in$ covSets do
                if $|\bigcup$attempt[s$'$/s]$| > |\bigcup$attempt$|$ then
                    attempt $\leftarrow$ attempt[s$'$/s]
                end
            end
        end
    end
    return attempt
end
```

Algorithm 2: Selecting the max-cover fuzzers



This file provides a `gen_pycode` function to generate random Python code to test CPython. The current version of `gen_pycode` is primitive and may generate incorrect results. You need to refine and repair it.

```
def gen_pycode(rng, out):
  out.write('import ' + rng.rand_str())
  <complete_this>
```

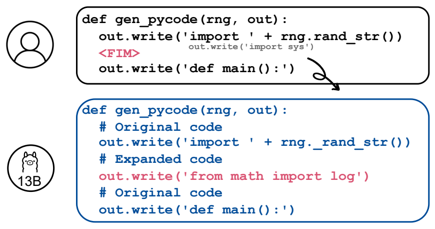
```
def gen_pycode(rng, out):
  out.write('import ' + rng.rand_str())
  if rng.read_byte() % 2 == 0:
    out.write('if name == __main__')
  else:
    out.write('def main():')
```

Figure 5: Prompts for the LLM-driven mutation

known "big models" such as GPT-4. However, even with such a small model, ELFuzz has shown significant advantages over existing techniques. We attribute this to the evolution loop that decomposes the fuzzer synthesis task into small steps, which is tractable even for a small model. Bigger models can be seamlessly plugged into our implementation for better performance.

Prompt engineering. At each evolution iteration, ELFuzz feeds a candidate fuzzer with a prompt that states the purpose of the code (i.e., generating inputs to test a SUT) and the task for the LLM (i.e., mutating the code) prepended to the fuzzer code as comments with a simple hint for the format (e.g., XML documents always start with `<?xml ...?>`). Since the task for the LLM (viz., completing, replacing, or splicing some code, as explained in §4.1) in each iteration is quite simple, the prompt consists of only several sentences. This minimizes the manual labor needed for prompt engineering. Figure 5 presents a candidate fuzzer and the prompt (simplified for illustration).

Implementation of the LLM-driven mutators. The completion mutator is implemented by a text completion query to the LLM, where the LLM simply continues the queried text (i.e., the code to complete) until a terminal token. The infilling



```
def gen_pycode(rng, out):
  out.write('import ' + rng.rand_str())
  <FIM>            out.write('import sys')
  out.write('def main():')
```

```
def gen_pycode(rng, out):
  # Original code
  out.write('import ' + rng._rand_str())
  # Expanded code
  out.write('from math import log')
  # Original code
  out.write('def main():')
```

Figure 6: Fill-in-the-middle query

and splicing mutators are implemented via fill-in-the-middle queries [25] supported by the most recently updated LLMs. Fill-in-the-middle queries insert a special `FIM` token into the queried text, and the LLM will return possible expansions that fit the token's location. Figure 6 shows how the infilling mutator is implemented via fill-in-the-middle queries. Implementation of the splicing mutator is similar: the `FIM` token is inserted between the head of the first candidate fuzzer and the tail of the second candidate fuzzer and expanded to glue code after the fill-in-the-middle query.

Seed fuzzers. ELFuzz improves candidate fuzzers step by step via the evolution loop. In each iteration, the LLM makes only small modifications. However, the improvements will accumulate as the process advances and eventually become a fully functional fuzzer. Thus, the initial seed fuzzers just naïvely generate purely random texts via the provided I/O utilities. The template of the seed fuzzers is included in the Appx. A. The seed fuzzers and the synthesized fuzzers are written in Python.

6 Evaluation

We have conducted a comprehensive evaluation, which aims to investigate the following research questions on ELFuzz's effectiveness and characteristics:

- **RQ1: Effectiveness in fuzzing.** How effective is ELFuzz in fuzzing? Specifically, how much coverage can ELFuzz achieve compared to other input generation techniques?
- **RQ2: Effectiveness in bug finding.** How effective is ELFuzz in finding bugs? How does it perform in finding bugs compared to other input generation techniques? Also, can it find new bugs in real-world software?
- **RQ3: Ablation study.** How do different components of ELFuzz, viz., the fuzzer space guidance and the three LLM-driven mutators, contribute to its effectiveness? Which is the most important one?
- **RQ4: Interpretability and extensibility.** Do the synthesized fuzzers catch interesting characteristics of the SUTs in an interpretable way? Can the synthesized fuzzers be extended with other techniques that enhance grammar-based fuzzing?

6.1 Benchmarks and Baselines

Benchmarks. Our evaluation leverages seven widely used SUTs from FuzzBench [55], OSS-Fuzz [9] and the open-source community as the benchmarks. These SUTs span diverse fields, require structural inputs with complex grammars and/or semantic constraints, and have codebases at real-world scales (up to 1.8 million LoC).

Baselines. We revisited the publications listed in Table 2 to select baselines. Among them, AUTOGRAM [41], GLADE [24], MIMID [38], and ISLEARN [62] share the same niche as ELFUZZ, focusing on input generation. AUTOGRAM, GLADE, and MIMID synthesize grammars, while ISLEARN synthesizes semantic constraints. Techniques in the other publications are domain-specific and/or fuse the input generation and mutation functionalities cohesively. For example, METAMUT [57] also improves fuzzing via code written by LLMs, but the application domain is strictly restricted to C compilers. They are unsuitable to serve as baselines for ELFUZZ .

As discussed in §2, AUTOGRAM [38] and MIMID [38] face scalability issues. While AUTOGRAM is built for Java, its successor, MIMID, can be applied to C projects. However, the implementation of MIMID is tailored to the nine benchmarks in the paper, eight of which are of less than 1,000 LoC, and the remaining one is of 9,000 LoC. Our attempts showed that adapting it to the real-world SUTs of up to 1.8 million LoC as our benchmarks requires an unaffordable engineering effort. For example, MIMID handles only 25 kinds of AST nodes parsed by `libclang` when instrumenting C programs, but there are 248 in total [4].

We include grammars synthesized by GLADE as a baseline, despite the doubts about the reliability of its evaluation results [26]. The comparison between GLADE and ELFUZZ is not entirely fair due to a key implementation difference. While the ELFUZZ fuzzers synthesize test cases without references to any existing test cases, the GLADE fuzzer produces new test cases by mutating existing ones under the guidance of a grammar. This means GLADE will take advantage of the domain knowledge contained in manually written seed test cases. Besides, the grammars synthesized by GLADE are stored in a special binary format and cannot be leveraged by other grammar-based fuzzers.

Therefore, another baseline is needed to manifest the effectiveness of ELFUZZ . GRAMMARINATOR [39], a widely used grammar-based fuzzer, with manually written grammars from the ANTLR4 project [3], serves the role. The combination has two advantages to facilitate the comparison: 1) GRAMMARINATOR produces test cases in the same way as ELFUZZ , i.e., solely according to the grammars without references to any existing test cases, and 2) the ANTLR4 grammars are manually written by domain experts and used as golden standards to verify the correctness of the synthesized grammars in previous work [26, 38], thus should reliably cover grammatical and semantic features of the SUTs.

SUT	Input format	Category	LoC
`jsoncpp`	JSON	Markup language	12,457
`libxml2`	XML	Markup language	267,880
`re2`	RegEx	Text processing	37,431
`SQLite`	SQL	Database	418,823
`CPython`	Python	Interpreter	1,791,104
`cvc5`	SMT-LIB 2	SMT solver	553,771
`librsvg`	SVG	Image rendering	52,125

Table 3: Benchmarks

Fuzzer	Short name
GRAMMARINATOR with ANTLR4 grammars	GRMR
ISLA with ANTLR4 grammars	ISLA
ISLA with ANTLR4 grammars and ISLEARN constraints	ISLEARN
GLADE	
ELFUZZ	

Table 4: Fuzzers to compare

Besides grammars, we also want to evaluate how existing techniques perform in synthesizing semantic constraints. Semantic constraints gain less attention than grammars, as mentioned in §2, and ISLA [62] is the only general-purpose grammar-based fuzzer that supports semantic constraints to our knowledge. It consumes semantic constraints mined by ISLEARN, which was proposed in the same paper. Thus, we combine these two techniques, along with the ANTLR4 grammars, as a baseline to evaluate their performance in synthesizing and leveraging semantic constraints. We also use ISLA [62] as a pure grammar-based fuzzer, i.e., with only the ANTLR4 grammars and without semantic constraints, to highlight the effectiveness of the semantic constraints. Together, we get five fuzzers to compare, listed in Table 4.

The format of grammars consumed by ISLA is different from ANTLR4, but can be translated from the latter. We implemented several Python scripts to conduct the translation.

6.2 Evaluation Design

Answering RQ1. To answer RQ1, we ran each fuzzer for 10 minutes to generate test cases. The generated test cases are then minimized (by `afl-cmin`) and fed to AFL++ as seeds for 24-hour mutation-based fuzzing. Such a setup aligns with the typical usage of generation-based fuzzers, i.e., as input generators for a mutation-based fuzzer. The mutation-based fuzzing is repeated 10 times to mitigate random disturbance. We use the edge coverage to evaluate the effectiveness of fuzzing. The 10-minute setting for input generation enables the fuzzers to cover various features of the SUTs while keeping the corpus small enough for AFL++ to explore well. The 24-hour and 10-repetition settings for mutation-based fuzzing inherit typical setups from previous work [23, 29, 61].

Answering RQ2. To answer RQ2, we conducted two-fold experiments, including controlled experiments and a real-world bug-finding experiment. In the controlled experiments, we fed the previously generated seeds to AFL++ to conduct 24-hour mutation-based fuzzing to find artificial bugs injected into `libxml2`, `SQLite`, and `CPython` by FIXREVERTER [77]. FIXREVERTER "reverses" fixes of existing CVEs and injects the reverted fixes into the software to introduce the original bugs. Such a setup provides a quantitative comparison between the fuzzers. We include only the three benchmarks written in C among all seven benchmarks because FIXREVERTER only applies to C code [77]. In total, 2,180 bugs are injected into `libxml2`, 1,092 are injected into `CPython`, and 780 are injected into `SQLite`. Each experiment is repeated 10 times to mitigate random disturbance. When a bug is triggered, the debug code attached by FIXREVERTER will print the ID of the bug. We use these IDs to distinguish and count unique bugs. For example, if a test case triggers three bugs, the debug code will print information like "Bugs 42, 43, 44 triggered" in the standard output, and our evaluation script will count these three bugs as triggered ones.

In the real-world bug-finding experiment, we fed the seeds generated by ELFUZZ to AFL++ to fuzz `cvc5` of the newest version as we started the experiment (version 1.1.2) for 14 days with 30 parallel instances to check whether ELFUZZ can find real-world bugs.

Answering RQ3. To answer RQ3, we construct four variants of ELFUZZ, each excluding a component of the approach:

- ELFUZZ-NOFS replaces the fuzzer space model with a simple top-*k* elite selection strategy [45] where *k* is set to 10. In each evolution cycle, it selects the top 10 high-coverage variants without the guidance of the fuzzer space.
- The other three variants respectively exclude one LLM-driven mutator, viz., ELFUZZ-NOSP excluding splicing, ELFUZZ-NOIN excluding infilling, and ELFUZZ-NOCP excluding completion.

We recorded the coverage trends of the candidate fuzzer during the evolution to show the impact of different components on the efficiency of the evolution loop. Then, we ran the synthesized fuzzers for 10 minutes and counted the edge coverage of generated test cases to inspect the impact of different components on the effectiveness of the synthesized fuzzers.

Answering RQ4. To answer RQ4, we conducted two case studies. The two case studies serve as evidence of the interpretability and extensibility of ELFUZZ . Although more rigorous experiments and analyses are needed to generalize the conclusions to a broader scope in the future, these case studies aim to provide initial insights within a reasonable workload and timeframe.

The first case study is to show the interpretability of ELFUZZ. It fed 31,200 test cases sampled from the total of 3,120,000 generated by the fuzzer synthesized by ELFUZZ to `SQLite` and counted the number of test cases that hit each source file. Fewer hits should indicate more interesting code (e.g., special `SQLite` features in contrast to commonly seen command-line option validation) in a source file. We present example test cases that hit these interesting source files and the fuzzer code that produces them.

The second case study is to show the extensibility of ELFUZZ . It comprises a proof-of-concept prototype that enhances the ELFUZZ fuzzer for `cvc5` with ZEST [58], a technique to enlarge the coverage of existing generation-based fuzzers. We will explain the crux of ZEST and how to implement it on fuzzers synthesized by ELFUZZ . We will present the amount of work spent on implementing such a prototype and validate the correctness of the implementation.

6.3 Experiment Settings and Costs

The LLM. We use a local full-precision (BF-16) version of CodeLlama-13B model, configured with a temperature of 0.2, a repetition penalty of 1.15, and a limit of 8192 input tokens in total. The LLM-driven evolution is run for 50 iterations for each benchmark, and each iteration produces 200 mutants, among which 10 are selected as survivors.

Synthesis costs. The experiments (including the synthesis processes and the fuzzing campaigns) occupy two AMD EPYC 7251 CPUs, one NVIDIA H100 Tensor Core GPU, and 16 GiB of memory. Each fuzzing campaign is assigned a single CPU core. There is no API invocation cost since all the computation happens locally. We use the NVIDIA H100 Tensor Core GPU as it is our only GPU for general-purpose computation. CodeLlama-13B is a relatively small model. Any GPU with more than 26 GiB VRAM (like NVIDIA A40 [8]) should be able to run it [11], and model quantization may enable it on GPUs with even less VRAM [59]. Similarly, the 16 GiB of memory is an upper bound, too. We set the memory limit to this fixed value to ensure that it is large enough for all experiments to avoid exceptional failure. During the actual synthesis process, this upper bound has never been reached. A smaller memory limit may still fit the requirements. The synthesis of grammars for GLADE and semantic constraints for ISLEARN is conducted on the same machine. They do not leverage the GPU, though.

The time for ELFUZZ to synthesize the fuzzers is listed in Table 5, together with the time for GLADE to synthesize the grammars and ISLEARN to mine the semantic constraints. There is no data point of ISLEARN for the `jsoncpp` benchmark, as the JSON format is simple and can be completely specified by a context-free grammar without semantic constraints. Thus, ISLEARN does not apply to it.

When using GLADE, we followed the replication study [26] to filter out seed test cases (required by GLADE as ground truths [24]) larger than 100 bytes and keep at most 30 ones randomly sampled (if there are more in total) to make the synthesis process finish in an acceptable time. Our first five attempts on `jsoncpp` without such filtration always failed

Fuzzer	`jsoncpp`	`libxml2`	`re2`	`CPython`	`SQLite`	`cvc5`	`librsvg`
ELFUZZ	21.4	25.6	29.5	55.5	40.6	54.9	19.9
ISLEARN	N/A	3.12	0.39	7.09	1.71	0.09	0.09
GLADE	0.20	2.53	0.11	1.46	0.29	5.72	1.12

Table 5: Time costs of synthesis (h)

due to a stack overflow error. Inspection showed that this was caused by very deep recursions on large seed test cases.

Generally, ELFUZZ takes the longest time as it requires multiple evolution iterations. This is justifiable since:

- The synthesis process in ELFUZZ happens only once before the fuzzing process. It will not harm the efficiency of fuzzing at runtime. Besides, it can be reasonably completed within 2.5 days for every benchmark.
- GLADE synthesizes only grammars. Additionally, it takes advantage of the filtration of the seed test cases.
- ISLEARN synthesizes only semantic constraints. Moreover, the synthesized semantic constraints make no contribution (compared to ISLA without these semantic constraints) on five out of the six benchmarks, as shown in §6.4.1 and §6.4.2 later. It brings some coverage promotion to the `librsvg` benchmark. Still, the promotion is significantly lower than ELFUZZ .

In later experiments, we exclude synthesis time from the recorded fuzzing time, as synthesis is not part of the AFL++ fuzzing process, and the evaluation is primarily concerned with the influence of the methods on the fuzzing effectiveness. One must trade off the promotion of the fuzzing effectiveness against the extra time spent on fuzzer synthesis when applying ELFUZZ in practice, though. For example, in later experiments of RQ2, ELFUZZ spent 30 minutes to trigger 75% of the total number of bugs it can trigger in the 24-hour fuzzing experiment on `libxml2`, as shown by Table 6, while the second-best method spent 78 minutes. However, the speed-up is less obvious when considering the synthesis time as well. This limitation could be a direction for future research, as we will discuss in §7.

Environment and miscellaneous settings. All the experiments are conducted in Docker containers. The Docker version is 24.0.7. The operating system is Ubuntu 24.04 LTS. During fuzzing, AFL++ uses dictionaries shipped with the SUTs [7, 14, 16] or provided by the OSS-Fuzz and FuzzBench projects [6, 15] for the six benchmarks other than `cvc5`. There are no pre-compiled dictionaries for `cvc5`. We collected tokens from the project's regression tests [12] and used them as the dictionary. Edge coverage [32] was used to compute the cover sets. Each cover set is approximated by feeding 1,000 inputs generated by a candidate fuzzer into the corresponding SUT.

GLADE requires manually written seed test cases as labeled ground truths for grammar synthesis. We extracted them from the test suite of each benchmark [7, 12–14, 16, 19, 64].

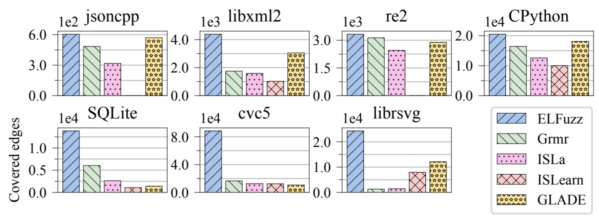


Figure 7: Coverage of the inputs generated within 10min

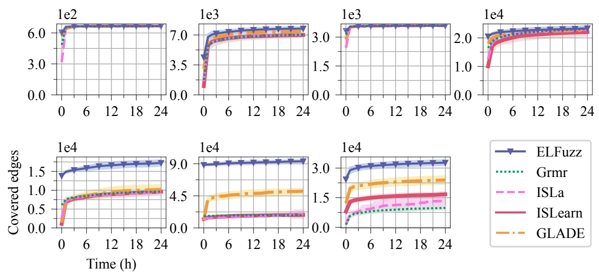


Figure 8: Coverage trends during mutation-based fuzzing

It also requires a binary for each SUT as an oracle that returns 0 for test cases conforming to the correct format and non-zero values for others. We create such binaries using either the grammar parsing APIs of the SUTs if provided [7, 12–14, 16, 19] or open-source parsers [20]. Grammars of benchmarks except `librsvg` are fetched from the official grammar repository of the ANTLR4 project [3]. While no formal grammar for SVG exists, SVG, based on XML, is essentially the XML grammar plus semantic constraints [18, 35]. Thus, we use the XML grammar for `librsvg`. ISLEARN is expected to synthesize the extra semantic constraints for the SVG format.

We use forked versions of GLADE, ISLA, and ISLEARN, which fix fatal bugs and add some convenient command-line interfaces. The modification does not affect any of the main functionalities presented in the papers. The code of the forked versions is released together with the code of ELFUZZ. For each benchmark, ISLEARN typically proposes hundreds to thousands of candidate semantic constraints, while ISLA can only handle one during fuzzing. Thus, we keep only the candidate with the best fitness scores.

6.4 The Results

6.4.1 RQ1: Effectiveness in fuzzing

Figure 7 presents the edge coverage of inputs generated by the four fuzzers within ten minutes. For every SUT, ELFUZZ achieves the best performance, with up to 434.8% more covered edges (on the `cvc5` benchmark). There are no results for ISLEARN on the `re2` benchmarks because ISLEARN failed to synthesize any semantic constraints for it (possibly because its predefined templates do not apply to the SUT). This shows the limited applicability of ISLEARN.

The coverage of seeds generated by GLADE is greater than that of other methods except ELFUZZ on six out of the seven benchmarks. This is not surprising, recalling that GLADE gets the test cases by mutating manually written seeds, while the other methods synthesize them without references to any existing ones. Notably, ELFUZZ significantly outperforms GLADE on all benchmarks, even with such unfairness.

Figure 8 presents the coverage trends of AFL++ on seeds generated by each fuzzer to compare. The solid lines are the averaged values over the 10 repetitions, and the narrow shadows show the standard deviations. Though the parts other than the seeds are identical in the fuzzing processes, the high coverage of seeds generated by ELFUZZ brings significant advantages to the later fuzzing processes. On `libxml2`, `CPython`, `SQLite`, `cvc5`, and `librsvg`, AFL++ instances that use other seeds can never catch up to the instance that uses the ELFUZZ seeds, and the gaps stay significant (up to 436.4% when comparing ELFUZZ with the second-best results) at the end of the experiments. On `jsoncpp` and `re2`, other techniques gradually reach similar coverage as time elapses. This is because the JSON format and regular expressions both have relatively simple grammars. The grammatical tokens of them are mostly single characters (e.g., "{" and "[" in JSON, and "?", "*" in regular expressions) and do not require complex rules (e.g., matching tags in XML) to be valid. Mutation-based fuzzing can work well by itself, whatever the seeds are; therefore, the gulf brought by the seeds is gradually bridged when given sufficient time. However, seeds generated by ELFUZZ reduce the time to reach high coverage.

While ELFUZZ initially brings huge advantages, and the advantages remain huge after 24-hour fuzzing on `SQLite`, `cvc5`, and `librsvg`, Figure 8 shows that the fuzzing process tends to level down the advantage on `libxml2` and `CPython`. This can be explained by the fact that ELFUZZ only provides initial seeds, and the mutation process is all the same across the methods. The longer the mutation-based fuzzing runs, the greater the proportion of the latter counts in the coverage. §7 will discuss this further.

RQ1 Answer. *The above results demonstrate that ELFuzz can generate test cases with high coverage (up to 434.8% more compared to other techniques). These test cases, as seeds, bring significant advantages for later mutation-based fuzzing compared to other techniques.*

6.4.2 RQ2: Effectiveness in bug finding

Figure 9 presents the number of bugs triggered by AFL++ within 24 hours using different seeds. The solid lines are the average values, and the shadows show the standard deviations. The results demonstrate that the seeds generated by ELFUZZ can not only find more bugs but also find the bugs faster. Table 6 presents the time (averaged over the 10 repetitions) required by the seeds generated by each fuzzer to reach 25%, 50%, and 75% of the number of all bugs triggered by

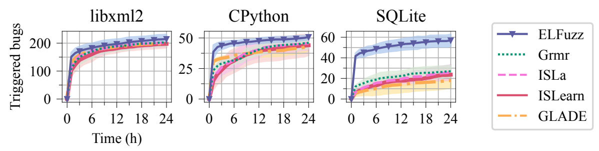


Figure 9: Number of the triggered bugs during 24-hour mutation-based fuzzing, showing that ELFUZZ finds more bugs in a shorter time

Fuzzer	libxml2			CPython			SQLite		
	25%	50%	75%	25%	50%	75%	25%	50%	75%
ELFUZZ	**10**	**20**	**30**	**10**	**22**	**39**	**10**	**20**	**30**
GRMR	**10**	41	145	**10**	52	484	69	769	1080
ISLA	11	41	119	12	166	374	250	805	∞
ISLEARN	11	57	173	78	264	434	332	1027	∞
GLADE	**10**	22	78	11	31	284	562	455	∞
Speed-up	1.0x	1.1x	2.6x	1.0x	1.4x	7.3x	6.9x	22.8x	36.0x

Table 6: Time (min) for each method to trigger 25%, 50%, and 75% of the total number of bugs triggered by ELFUZZ and the speed-up by ELFUZZ over the second best

the seeds generated by the ELFUZZ fuzzers as quantitative metrics to demonstrate the different speeds.

Figure 10 shows the number of unique bugs (i.e., bugs that can only be triggered by the seeds generated by one specific fuzzer) they find. On `libxml2` and `SQLite`, the seeds generated by ELFUZZ find the most unique bugs (5.7–13x the second best). On `CPython`, it gets the same best number as ISLA (3x the third best).

We analyzed the 16 bugs in the three benchmarks that ELFUZZ cannot trigger and found three kinds of causes. The details of the analysis are in Appx. B. In general, six out of the 16 bugs are triggered by SUT features, e.g., `match` statements in Python, that are missing in the ELFUZZ fuzzers. We will discuss possible solutions to avoid such failures in §7. The other ten bugs are either caused by randomness or implementation differences between ELFUZZ and other methods. The absence of these ten bugs is not attributable to ELFUZZ.

After fuzzing `cvc5` for 14 days by 30 AFL++ instances with the seeds generated by ELFUZZ, five new bugs were found in the latest version of `cvc5` at the time when we started our testing. Interestingly, we noticed two of these bugs were recently fixed in the latest released version. We, therefore, have just disclosed the three unfixed bugs. Note that the five bugs include a format string injection vulnerability and two denial-of-service vulnerabilities caused by dead loops. The

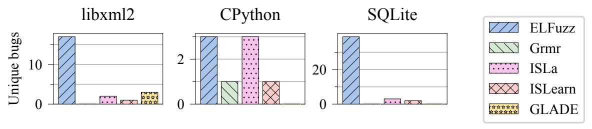


Figure 10: Unique bugs that only one fuzzer can find

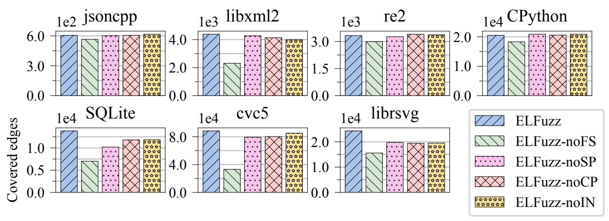


Figure 11: Coverage of the inputs generated by the variants within 10min

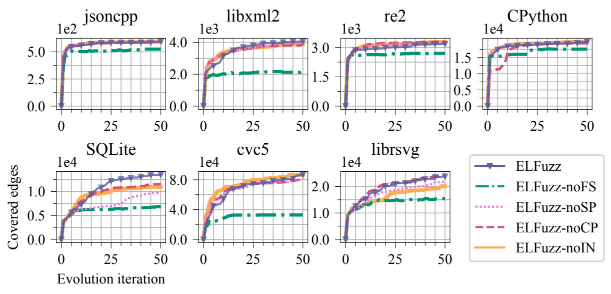


Figure 12: Edge coverage of survivors during the synthesis

other two bugs involve operation on invalid pointers, which may possibly be exploited for control flow hijacking (though we have not confirmed this yet). One of the denial-of-service vulnerabilities is triggered by the complex cyclic dependency of SMT expressions generated by the synthesized fuzzer. Compared with other fuzzing papers (typically fuzzing for more than six months with more than 50 parallel AFL++ instances [57]), the time budget we use is short, but ELFuzz can still find severe and complex real-world vulnerabilities. Details of the five bugs are in Appx. C.

RQ2 Answer. *The above results show that ELFuzz is not only effective in covering more code but also in finding more bugs. In the controlled experiments, it finds more bugs in a shorter time than other techniques. Moreover, it can find the most unique bugs. In the real-world experiment, ELFuzz finds five new bugs, including severe and complex vulnerabilities, even with a short time budget.*

6.4.3 RQ3: Ablation study

Figure 11 compares the coverage of test cases generated by ELFuzz, ELFuzz-noFS, ELFuzz-noSP, ELFuzz-noCP, and ELFuzz-noIN within 10 minutes. Among the four variants, ELFuzz-noFS has the largest gap (6.6–62.5%) compared to the original ELFuzz. This indicates that the fuzzer space model contributes the most to ELFuzz's performance. The three variants which exclude LLM-driven mutators (ELFuzz-noSP, ELFuzz-noCP, and ELFuzz-noIN) have almost the same coverage as ELFuzz on the first four benchmarks, `jsoncpp`, `libxml2`, `re2`, and `CPython`. The gaps become significant on the other three benchmarks `SQLite`, `cvc5`, and `librsvg`, especially for ELFuzz-noSP (up to 26.2% on the three benchmarks). As stated in §4.1, the splicing mutator is the only mutator that could combine the strengths of different candidate fuzzers. In contrast, the other two mutators can only improve a single fuzzer. Excluding the splicing mutator hinders the evolution process from joining fit features from multiple candidate fuzzers. However, none of the LLM-driven mutators are as important as the fuzzer space model.

Figure 12 shows the edge coverage of candidate fuzzers selected at each iteration during the LLM-driven synthesis processes, echoing the above observation that ELFuzz-noFS's performance downgrades the most. As the trends show, without the guidance of a fuzzer space, ELFuzz-noFS quickly reaches the ceiling, causing significant gaps between ELFuzz-noFS and other variants at the end of the synthesis processes. Besides, ELFuzz-noFS's coverage trends are not guaranteed to increase monotonically (while other variants are), as it cannot ensure that the candidate fuzzers get improved at every iteration due to the lack of guidance of the fuzzer space.

RQ3 Answer. *Fuzzer space contributes the most to the performance of ELFuzz. While excluding an LLM-driven mutator causes some performance downgrading (up to 26.2%), none of them have impacts as significant as that caused by excluding the fuzzer space model (up to 62.5%*

6.4.4 RQ4: Interpretability and extensibility

In the case study on interpretability, we analyzed 31,200 test cases generated by the ELFuzz fuzzers for SQLite and counted the number of test cases that hit each source file. Appx. D presents the details of the analyses. The results show a significant gap between the top-4 least-hit files and the other files: the hit numbers of the top-4 least-hit files are all less than 1,100, while those of the other files are larger than 14,000. This gap indicates interesting code that handles special SQL language features rather than trivial functionalities like command-line argument parsing in the four source files. These four files with their corresponding language features are: 1) `ctime.c` handling the `compile_option` pragma, 2) `vacuum.c` handling the `VACUUM` command, 3) `rowset.c` operating rows in a database, and 4) `vdbesort.c` sorting the results of a SQL query.

Among these language features, the `compile_option` pragma and the `VACUUM` command are simple but rarely used. In contrast, row operations and query sorting are commonly used features with complex grammatical and semantic constraints. The synthesized fuzzers can cover both common features and rarely used features, as well as both simple features and complex features.

Listing 13 presents part of the fuzzer code that targets `rowset.c`. The row operations in `rowset.c` requires non-empty tables, and the code satisfies this constraint by insert-

```
for value in VALUES:
    testcase.write('INSERT INTO ')
    testcase.write(TABLE_NAME)
    testcase.write(' VALUES (')
    for v in value[:-1]:
        testcase.write(v)
        testcase.write(', ')

    testcase.write(value[-1])
    testcase.write(');\n')
```

Listing 13: Synthesized code that inserts rows to a table

ing rows into the generated table before invoking any row operations. Similar code exists for the other three features. Such code shows that ELFUZZ handles the grammatical and semantic constraints of the SUTs in an interpretable way.

In the case study on extensibility, we implemented a proof-of-concept prototype that enhances the fuzzers synthesized for `cvc5` with ZEST [58], a technique that enlarges the coverage of existing generation-based fuzzers. The crux of ZEST is to "parameterize" generation-based fuzzers to let them generate test cases according to provided byte arrays instead of arbitrary random values, and then evolve the byte sequences using a genetic algorithm. Fuzzers synthesized by ELFUZZ are naturally easy to adapt to this technique, as they are basically Python functions. The adaptation can be achieved by replacing the random number generators passed to the functions as an argument with a byte array. In total, the adaptation takes five man-days. We also validated the correctness of the adaptation by manually tracing how the first 10 test cases are generated to ensure that the byte-array evolution algorithm is followed. Note that the implementation lacks various optimizations mentioned in the paper and thus cannot be a practical fuzzer due to low fuzzing efficiency. It can serve as an illustration of the extensibility of ELFUZZ , though. Appx. E details the implementation and validation of the prototype.

RQ4 Answer. *The results above confirm that fuzzers synthesized by ELFuzz catch diverse and correct grammatical and semantic requirements of the input format of the SUT, even those complex ones, in an interpretable way. Moreover, these fuzzers can be extended with existing techniques that enhance generation-based fuzzing.*

7 Limitations and Future Work

Better seed fuzzers. The current implementation of ELFUZZ uses the same seed fuzzer that outputs random bytes for all SUTs for simplicity. Better-designed seed fuzzers, possibly produced by an LLM [48], may further improve the performance of ELFUZZ.

Data contamination and input formats. Data contamination is a well-known problem for LLM benchmarks [52]. However, it is exactly what we need for ELFUZZ . Instead of trying to *test what an LLM knows*, ELFUZZ aims to *leverage what an LLM knows*. Knowledge about the SUTs in the training set is necessary for the LLM to guess the input format during the synthesis of the fuzzers. Uncommon SUTs, such as proprietary software with limited publicly available documentation, may lack such knowledge. Besides, binary input formats could also cause problems, as they are typically excluded from the training sets of LLMs. Approaches such as prompt engineering or fine-tuning could help inject the necessary information into the LLM in such situations.

Evolution throughout the entire fuzzing cycle. Currently, ELFUZZ acts as the input generator for initial seeds. Thus, it mainly boosts the early stage of the fuzzing cycle. However, maintaining corpus diversity is crucial during the entire fuzzing process. Extending the use of the ELFUZZ fuzzers to the entire fuzzing cycle could benefit fuzzing processes more, but it faces a main difficulty caused by the complex dynamics of fuzzing. As the fuzzer touches different parts of a SUT, the desired properties of the corpus keep changing. For example, the importance of grammatical structures may decrease after the fuzzer proceeds to the assembly optimization passes when testing a compiler. Moreover, rarely used features initially ignored by ELFUZZ may be discovered, and covering them would require adaptation of the ELFUZZ fuzzer to cover them. A possible solution is to continuously evolve the input generators throughout the entire fuzzing cycle to keep pace with such changing needs. The continuous evolution and input generation can also make the time spent on fuzzer synthesis before the fuzzing campaigns more cost-effective. Future research may investigate how to leverage runtime feedback to facilitate continuous evolution, enabling adaptive input generation and the incorporation of newly discovered features of the SUTs while maintaining reasonable resource costs (such as GPU hours).

8 Conclusion

In this paper, we introduce the novel notion of fuzzer space, which structures fuzzers into a lattice to assess their relative effectiveness. Based on the notion of fuzzer space, we develop ELFUZZ to perform LLM-driven evolution with the fuzzer space guidance to synthesize fuzzers for fully automated, efficient, and interpretable input generation. The test cases generated by the synthesized fuzzers can then serve as initial seeds for a mutation-based fuzzer to reach deeper into the codebase. We have implemented and evaluated ELFUZZ. The results show that ELFUZZ outperforms state-of-the-art techniques in fuzzing and bug-finding experiments. In a 14-day real-world fuzzing campaign on `cvc5`, ELFUZZ found five new bugs, including three severe and complex vulnerabilities, despite the short time budget. We also conducted ablation studies. They show that the fuzzer space model contributes the most to the effectiveness of ELFUZZ. Additionally, two case studies demonstrate that the synthesized fuzzers capture interesting features of a SUT in an interpretable way and can be extended with other generation-based fuzzing techniques.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd for providing their feedback that improved this work. This work was supported by NSF awards 2112471 and 2419722.

Ethics Considerations

All the experiments, including those comparing different techniques on the bug-injected benchmarks, are conducted locally. No ethical risk exists in our work. Also, we have responsibly disclosed the new vulnerabilities (and bugs) to relevant stakeholders, and at this time of writing, the corresponding patch is still under development.

Open Science

The open-source implementation of ELFuzz together with the source code of the baselines and the experiment data is available at https://doi.org/10.6084/m9.figshare.29177162.

References

[1] "The AFL++ fuzzing framework." [Online]. Available: https://aflplus.plus/

[2] "AFL trophy cases." [Online]. Available: https://lcamtuf.coredump.cx/afl/#bugs

[3] "antlr/grammars-v4: Grammars written for ANTLR v4." [Online]. Available: https://github.com/antlr/grammars-v4

[4] "clang.cindex — libclang 16.0.6 documentation." [Online]. Available: https://libclang.readthedocs.io/en/latest/_modules/clang/cindex.html#CursorKind

[5] "Format string attack | OWASP foundation." [Online]. Available: https://owasp.org/www-community/attacks/Format_string_attack

[6] "fuzzbench: FuzzBench - Fuzzer benchmarking as a service." [Online]. Available: https://github.com/google/fuzzbench/tree/master

[7] "jsoncpp: A C++ library for interacting with JSON." [Online]. Available: https://github.com/open-source-parsers/jsoncpp

[8] "NVIDIA A40 GPU for visual computing." [Online]. Available: https://www.nvidia.com/en-us/data-center/a40/

[9] "OSS-Fuzz." [Online]. Available: https://google.github.io/oss-fuzz/

[10] "system under test - ISTQB glossary." [Online]. Available: https://glossary.istqb.org/en_US/term/system-under-test

[11] "codellama/CodeLlama-13b-python-hf · [AUTOMATED] model memory requirements," Nov. 2023. [Online]. Available: https://huggingface.co/codellama/CodeLlama-13b-Python-hf/discussions/7

[12] "cvc5/cvc5," Oct. 2024. [Online]. Available: https://github.com/cvc5/cvc5

[13] "GNOME / librsvg · GitLab," Oct. 2024. [Online]. Available: https://gitlab.gnome.org/GNOME/librsvg

[14] "GNOME/libxml2," Oct. 2024. [Online]. Available: https://github.com/GNOME/libxml2

[15] "google/fuzzing," Oct. 2024. [Online]. Available: https://github.com/google/fuzzing

[16] "python/cpython," Oct. 2024. [Online]. Available: https://github.com/python/cpython

[17] "Set cover problem," Dec. 2024, page Version ID: 1264796057. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Set_cover_problem&oldid=1264796057

[18] "Cover pages: XML and semantic transparency," Feb. 2025. [Online]. Available: https://web.archive.org/web/20250203130605/http://xml.coverpages.org/xmlAndSemantics.html

[19] "google/re2," May 2025, original-date: 2014-08-18T21:21:26Z. [Online]. Available: https://github.com/google/re2

[20] "sqlparser - crates.io: Rust Package Registry," May 2025. [Online]. Available: https://crates.io/crates/sqlparser

[21] H. Al Salem and J. Song, "A review on grammar-based fuzzing techniques," *International Journal of Computer Science & Security (IJCSS)*, vol. 13, no. 3, pp. 114–123, 2019.

[22] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, "Software engineering for machine learning: a case study," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, May 2019, pp. 291–300. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8804457

[23] M. Ammann, L. Hirschi, and S. Kremer, "DY Fuzzing: Formal Dolev-Yao models meet cryptographic protocol fuzz testing," in *2024 IEEE Symposium on Security and Privacy (SP)*, May 2024, pp. 1481–1499, iSSN: 2375-1207. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10646658

[24] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 95–110, Jun. 2017. [Online]. Available: https://dl.acm.org/doi/10.1145/3140587.3062349

[25] M. Bavarian, H. Jun, N. Tezak, J. Schulman, C. McLeavey, J. Tworek, and M. Chen, "Efficient training of language models to fill in the middle," Jul. 2022, arXiv:2207.14255 [cs]. [Online]. Available: http://arxiv.org/abs/2207.14255

[26] B. Bendrissou, R. Gopinath, and A. Zeller, "'Synthesizing input grammars': a replication study," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2022. New York, NY, USA: Association for Computing Machinery, Jun. 2022, pp. 260–268. [Online]. Available: https://dl.acm.org/doi/10.1145/3519939.3523716

[27] A. Bulekov, B. Das, S. Hajnoczi, and M. Egele, "No grammar, no problem: Towards fuzzing the linux kernel without system-call descriptions," in *Proceedings of 2023 Network and Distributed System Security Symposium*, 2023. [Online]. Available: https://par.nsf.gov/servlets/purl/10438507

[28] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu, "A systematic review of fuzzing techniques," *Computers & Security*, vol. 75, pp. 118–137, Jun. 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404818300658

[29] W. Chen, Y. Hao, Z. Zhang, X. Zou, D. Kirat, S. Mishra, D. Schales, J. Jang, and Z. Qian, "SyzGen++: Dependency inference for augmenting kernel driver fuzzing," in *2024 IEEE Symposium on Security and Privacy (SP)*, May 2024. [Online]. Available: https://zhyfeng.github.io/files/2024-Oakland-SyzGen++.pdf

[30] Y. Chen, R. Zhong, H. Hu, H. Zhang, Y. Yang, D. Wu, and W. Lee, "One engine to fuzz 'em all: Generic language processor testing with semantic validation," in *2021 IEEE Symposium on Security and Privacy (SP)*, May 2021, pp. 642–658, iSSN: 2375-1207. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9519403

[31] 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*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, Jul. 2023, pp. 423–435. [Online]. Available: https://doi.org/10.1145/3597926.3598067

[32] V. H. S. Durelli, M. E. Delamaro, and J. Offutt, "An experimental comparison of edge, edge-pair, and prime path criteria," *Science of Computer Programming*, vol. 152, pp. 99–115, Jan. 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642317302150

[33] J. Eom, S. Jeong, and T. Kwon, "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*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, Sep. 2024, pp. 1656–1668. [Online]. Available: https://dl.acm.org/doi/10.1145/3650212.3680389

[34] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Conference on Offensive Technologies*, ser. WOOT'20. USA: USENIX Association, 2020, p. 10.

[35] V. Geroimenko, "SVG and X3D in the context of the XML family and the semantic web," in *Visualizing Information Using SVG and X3D*, V. Geroimenko and C. Chen, Eds. London: Springer-Verlag, 2005, pp. 3–20. [Online]. Available: http://link.springer.com/10.1007/1-84628-084-2_1

[36] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: Association for Computing Machinery, Jun. 2008, pp. 206–215. [Online]. Available: https://dl.acm.org/doi/10.1145/1375581.1375607

[37] P. Godefroid, H. Peleg, and R. Singh, "Learn&Fuzz: Machine learning for input fuzzing," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Urbana, IL: IEEE, Oct. 2017, pp. 50–59. [Online]. Available: http://ieeexplore.ieee.org/document/8115618/

[38] R. Gopinath, B. Mathis, and A. Zeller, "Mining input grammars from dynamic control flow," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 172–183. [Online]. Available: https://dl.acm.org/doi/10.1145/3368089.3409679

[39] R. Hodovan, A. Kiss, and T. Gyimothy, "Grammarinator: a grammar-based open source fuzzer," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, ser. A-TEST 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 45–48. [Online]. Available: https://doi.org/10.1145/3278186.3278193

[40] 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," Aug. 2023, arXiv:2308.10620 [cs]. [Online]. Available: http://arxiv.org/abs/2308.10620

[41] M. Höschele and A. Zeller, "Mining input grammars from dynamic taints," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '16. New York, NY, USA: Association for Computing Machinery, Aug. 2016, pp. 720–725. [Online]. Available: https://dl.acm.org/doi/10.1145/2970276.2970321

[42] B. Jeong, J. Jang, H. Yi, J. Moon, J. Kim, I. Jeon, T. Kim, W. Shim, and Y. H. Hwang, "UTopia: Automatic generation of fuzz driver using unit tests," in *2023 IEEE Symposium on Security and Privacy (SP)*, May 2023, pp. 2676–2692, iSSN: 2375-1207. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10179394

[43] J. R. Koza, "Genetic programming as a means for programming computers by natural selection," *Statistics and Computing*, vol. 4, no. 2, pp. 87–112, Jun. 1994. [Online]. Available: https://doi.org/10.1007/BF00175355

[44] N. Kulkarni, C. Lemieux, and K. Sen, "Learning highly recursive input grammars," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2021, pp. 456–467, iSSN: 2643-1572. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678879

[45] A. Lalejini, E. Dolson, A. E. Vostinar, and L. Zaman, "Artificial selection methods from evolutionary computing show promise for directed evolution of microbes," *eLife*, vol. 11, p. e79665, Aug. 2022, publisher: eLife Sciences Publications, Ltd. [Online]. Available: https://doi.org/10.7554/eLife.79665

[46] W. B. Langdon and R. Poli, *Foundations of genetic programming*. Springer Science & Business Media, Mar. 2013.

[47] J. Lehman, J. Gordon, S. Jain, K. Ndousse, C. Yeh, and K. O. Stanley, "Evolution through large models," Jun. 2022. [Online]. Available: https://arxiv.org/abs/2206.08896v1

[48] D. Liu, O. Chang, J. metzman, M. Sablotny, and M. Maruseac, "OSS-fuzz-gen: Automated fuzz target generation," May 2024. [Online]. Available: https://github.com/google/oss-fuzz-gen

[49] J. Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. Zhang, "NNSmith: Generating diverse and valid test cases for deep learning compilers," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 530–543. [Online]. Available: https://dl.acm.org/doi/10.1145/3575693.3575707

[50] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, pp. 21 558–21 572, Dec. 2023. [Online]. Available: https://dl.acm.org/doi/10.5555/3666122.3667065

[51] X. Liu, W. You, Y. Ye, Z. Zhang, J. Huang, and X. Zhang, "FuzzInMem: Fuzzing programs via in-memory structures," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. Lisbon Portugal: ACM, Apr. 2024, pp. 1–13. [Online]. Available: https://dl.acm.org/doi/10.1145/3597503.3639172

[52] I. Magar and R. Schwartz, "Data contamination: From memorization to exploitation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, S. Muresan, P. Nakov, and A. Villavicencio, Eds. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 157–165. [Online]. Available: https://aclanthology.org/2022.acl-short.18/

[53] S. Mallissery and Y.-S. Wu, "Demystify the fuzzing methods: a comprehensive survey," *ACM Computing Surveys*, vol. 56, no. 3, pp. 71:1–71:38, Oct. 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3623375

[54] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, 2019.

[55] 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*, ser. ESEC/FSE 2019. Athens, Greece: Association for Computing Machinery, Aug. 2021, pp. 1393–1403. [Online]. Available: https://dl.acm.org/doi/10.1145/3468264.3473932

[56] F. N. Motlagh, M. Hajizadeh, M. Majd, P. Najafi, F. Cheng, and C. Meinel, "Large language models in cybersecurity: State-of-the-art," Jan. 2024, arXiv:2402.00891 [cs]. [Online]. Available: http://arxiv.org/abs/2402.00891

[57] X. Ou, C. Li, Y. Jiang, and C. Xu, "The mutators reloaded: Fuzzing compilers with large language model generated mutation operators," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2024)*, San Diego, USA, 2024. [Online]. Available: https://connglli.github.io/pdfs/metamut_asplos24.pdf

[58] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon, "Semantic fuzzing with Zest," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 329–340. [Online]. Available: https://doi.org/10.1145/3293882.3330576

[59] A. Polino, R. Pascanu, and D. Alistarh, "Model compression via distillation and quantization," Feb. 2018, arXiv:1802.05668 [cs]. [Online]. Available: http://arxiv.org/abs/1802.05668

[60] S. Sargsyan, S. Kurmangaleev, M. Mehrabyan, M. Mishechkin, T. Ghukasyan, and S. Asryan, "Grammar-based fuzzing," in *2018 Ivannikov Memorial Workshop (IVMEM)*, May 2018, pp. 32–35. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8636353

[61] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale-Ebrahim, N. Bissantz, M. Muench, and T. Holz, "SoK: Prudent evaluation practices for fuzzing," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 1974–1993. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10646824/

[62] D. Steinhöfel and A. Zeller, "Input invariants," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, pp. 583–594. [Online]. Available: https://dl.acm.org/doi/10.1145/3540250.3549139

[63] ——, "Language-based software testing," *Communications of the ACM*, vol. 67, no. 4, pp. 80–84, Mar. 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/3631520

[64] T. S. D. Team, "sqlite/sqlite," May 2025, original-date: 2019-03-18T12:21:01Z. [Online]. Available: https://github.com/sqlite/sqlite

[65] P. Vaithilingam, T. Zhang, and E. L. Glassman, “Expectation vs. experience: evaluating the usability of code generation tools powered by large language models,” in *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, ser. CHI EA ’22. New York, NY, USA: Association for Computing Machinery, Apr. 2022, pp. 1–7. [Online]. Available: https://doi.org/10.1145/3491101.3519665

[66] V. Vikram, C. Lemieux, J. Sunshine, and R. Padhye, “Can large language models write good property-based tests?” Jul. 2024, arXiv:2307.04346 [cs]. [Online]. Available: http://arxiv.org/abs/2307.04346

[67] Y. Wang, P. Jia, L. Liu, C. Huang, and Z. Liu, “A systematic review of fuzzing based on machine learning techniques,” *PLOS ONE*, vol. 15, no. 8, p. e0237749, Aug. 2020, publisher: Public Library of Science. [Online]. Available: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0237749

[68] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su, “Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization.” in *Proceedings of 2020 Network and Distributed System Security Symposium*, 2020. [Online]. Available: https://wcventure.github.io/FuzzingPaper/Paper/NDSS20_Prioritization.pdf

[69] F. Wilhelmstötter, “Jenetics: Java genetic algorithm library.” [Online]. Available: https://jenetics.io/

[70] X. Wu, S.-h. Wu, J. Wu, L. Feng, and K. C. Tan, “Evolutionary computation in the era of large language model: Survey and roadmap,” May 2024, arXiv:2401.10034 [cs]. [Online]. Available: http://arxiv.org/abs/2401.10034

[71] 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*, ser. ICSE ’24. New York, NY, USA: Association for Computing Machinery, Apr. 2024, pp. 1–13. [Online]. Available: https://dl.acm.org/doi/10.1145/3597503.3639121

[72] H. Xu, S. Wang, N. Li, K. Wang, Y. Zhao, K. Chen, T. Yu, Y. Liu, and H. Wang, “Large language models for cyber security: a systematic literature review,” Jul. 2024, arXiv:2405.04760 [cs]. [Online]. Available: http://arxiv.org/abs/2405.04760

[73] X. Yang, Y. Chen, E. Eide, and J. Regehr, “Finding and understanding bugs in C compilers,” in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’11. New York, NY, USA: Association for Computing Machinery, Jun. 2011, pp. 283–294.

[74] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, *The fuzzing book*. CISPA Helmholtz Center for Information Security, 2024. [Online]. Available: https://www.fuzzingbook.org/

[75] C. Zhang, Y. Zheng, M. Bai, Y. Li, W. Ma, X. Xie, Y. Li, L. Sun, and Y. Liu, “How effective are they? Exploring large language model based fuzz driver generation,” in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, Sep. 2024, pp. 1223–1235. [Online]. Available: https://dl.acm.org/doi/10.1145/3650212.3680355

[76] M. Zhang, J. Liu, F. Ma, H. Zhang, and Y. Jiang, “IntelliGen: automatic driver synthesis for fuzz testing,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, May 2021, pp. 318–327.

[77] Z. Zhang, Z. Patterson, M. Hicks, and S. Wei, “FIXREVERTER: a realistic bug injection methodology for benchmarking fuzz testing,” 2022, pp. 3699–3715. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-zenong

[78] X. Zhao, H. Qu, J. Xu, X. Li, W. Lv, and G.-G. Wang, “A systematic review of fuzzing,” *Soft Computing*, vol. 28, no. 6, pp. 5493–5522, Mar. 2024. [Online]. Available: https://doi.org/10.1007/s00500-023-09306-2

[79] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, “Fuzzing: a survey for roadmap,” *ACM Computing Surveys*, vol. 54, no. 11s, pp. 230:1–230:36, 2022. [Online]. Available: https://doi.org/10.1145/3512345

Appendix A The Seed Fuzzers

Listing 14 presents the seed fuzzer we use (simplified for illustration), where `FORMAT` is a placeholder for the input format required by the SUT. It provides clues for the LLM about what kinds of inputs it has to generate.

```
class ByteOrTextIO:
    def __init__(self, binary_io: BinaryIO): ...
    def read_chars(self, char_count: int) -> str:...
    def read_byte(self, size: int) -> int: ...

def gen_<FORMAT>(rng: Random, output: TextIO):
    len = rng.read_byte()
    random_text = rng.read_chars(len)
    output.write(random_text)
```

Listing 14: Template of the seed fuzzers

Appendix B Failure Case Analysis

Table 7 lists the causes of the failure cases in RQ2, i.e., those injected bugs that cannot be triggered by the fuzzers synthesized by ELFUZZ . There are 16 cases in total, and one of the failure cases (Bug 1425) is purely an implementation issue. Among the other cases, six (Bugs 204, 2030, 3043, 5088, 534, and 538) happen because of missing SUT features in the synthesized fuzzers, and nine (the other bugs) are caused by randomness. §7 discusses how to resolve the failure cases caused by missing features.

Appendix C The Real-world Bugs in `cvc5`

Our real-world bug-finding experiment in RQ2 has found five possibly exploitable 0-day bugs in `cvc5`. These bugs are as follows:

- **Bug 1: Format string injection.** This bug will be triggered when passing an invalid argument to the `set-logic` command. When printing the error message, `cvc5` will mistakenly format it with values on the stack, possibly causing attacks that execute arbitrary commands [5].
- **Bug 2: Dead loop during parsing.** This bug traps `cvc5` into a dead loop if the left parenthesis of a `set-option` command is not closed properly. The dead loop could lead to a denial-of-service attack.
- **Bug 3: Dead loop when rewriting expressions.** This bug traps `cvc5` into a dead loop when rewriting integer arithmetic expressions with self-dependency. The dead loop could lead to a denial-of-service attack.
- **Bug 4: Crash when setting multiple outputs.** This bug triggers a NULL-pointer dereference when setting multiple outputs.
- **Bug 5: Crash in `nullable` checks without arguments.** This bug triggers a NULL-pointer dereference when calling `nullable` checks without arguments.

Appendix D The Interpretability Case Study

Table 8 presents the hit numbers of the top-10 least-hit source files of SQLite. A significant gap exists between `vdbesort.c` and `upsert.c`.

Benchmark	Bug IDs	Cause
`libxml2`	204, 2030, 3043	These bugs are in code that handles multi-layer DTD element declarations. The feature is missing in the ELFUZZ fuzzers.
	1425	This bug is triggered by UTF-16 codes. The implementation of ELFUZZ forces all test cases to be UTF-8 encoded, while the ANTLR4 grammars permit UTF-16.
	3726	This bug is in a utility function that caches used strings. The ELFUZZ fuzzers and the ANTLR4 grammars can both generate test cases that trigger it.
	4658	The bug happens in the code that handles DTD ID declarations. The ELFUZZ fuzzers and the ANTLR4 grammars can both generate these definitions.
`CPython`	5063, 5094, 5695	These three bugs are in code that handles keyword arguments. The ELFUZZ fuzzers and the ANTLR4 grammars can both generate test cases containing this feature.
	5078	The bug happens during calling the `warnings` module. Neither ELFUZZ fuzzers nor the ANTLR4 grammars specify such usage explicitly, while the mutation process may hit it by chance.
	5088	The bug is related to `match` statements, which is missing in the ELFUZZ fuzzers.
`SQLite`	469	The bug is in the code that handles the `round` function. Neither ELFUZZ fuzzers nor the ANTLR4 grammars specify such cases explicitly, while the mutation process may hit it by chance.
	474, 475	The bugs are in the code that handles matched glob patterns. Neither ELFUZZ fuzzers nor the ANTLR4 grammars specify such cases explicitly, while the mutation process may hit it by chance.
	534, 538	The bugs are in the code that handles nested `SELECT` statements. The feature is missing in the ELFUZZ fuzzers.

Table 7: Causes of the failure cases in RQ2

Listing 15 and Listing 16 include a test case that covers row-related and sorting operations in SQLite and the code snippet that generates it. The test cases and code have been truncated and simplified for illustration. The test case for `ctime.c` is simply a one-line `compile_option` pragma, and the synthesized fuzzer generates it if a random boolean value (the "switch" for this feature) is true.

Appendix E Extending ELFUZZ with ZEST

Introduction to ZEST. ZEST [58] is a generation-based fuzzing technique that boosts generation-based fuzzers via an evolutionary algorithm similar to greybox mutation-based fuzzers such as AFL++ [34]. The crux of ZEST is as follows:

Traditional generation-based fuzzers are typically blind fuzzers [39, 54, 73], i.e., they generate test cases without feedback from the SUTs. While the specifications of the SUts enable the generation-based fuzzers to generate test cases that can pass the early parsing and semantic check stages, the lack of feedback prevents them from continuous improvements as the fuzzing process proceeds. On the contrary, greybox mutation-based fuzzers like AFL++ [34, 54] generate non-structural test cases but leverage coverage feedback from the SUTs to drive an evolution loop, where

Source file	Hits	Percentage
ctime.c	239	0.8%
rowset.c	527	1.6%
vacuum.c	707	2.3%
vdbesort.c	1,081	3.5%
upsert.c	14,908	47.8%
random.c	16,030	51.4%
delete.c	17,941	57.5%
update.c	27,135	87.0%
utf.c	27,135	87.0%
wherecode.c	27,135	87.0%

Table 8: The top-10 least-hit source files in `SQLite`

```
CREATE TABLE IF NOT EXISTS t1(
    c1, c2, c3 INTEGER PRIMARY KEY AUTOINCREMENT
);
INSERT INTO t1 VALUES (0, 1, 0);
INSERT INTO t1 VALUES (1, 2, 2);
SELECT * FROM t1 ORDER BY c1 DESC, c2 ASC, c2 DESC;
```

Listing 15: Test case for `rowset.c` and `vdbesort.c`

test cases that cover more code are preferred to produce offspring (i.e., new test cases mutated from it). Thus, the fuzzing process keeps improving itself as it proceeds.

ZEST proposes to graft the evolution loop of greybox mutation-based fuzzers to the generation-based fuzzers to enable the self-improvements of the latter. The fuzzer in Listing 14 is an example of a blind generation-based fuzzer, where the generated test cases solely depend on random choices according to a random number generator. ZEST first "parameterizes" the fuzzer by replacing the random number generator with a caller-provided byte array, as shown in Listing 17. Thus, the byte array provided as an argument can specify the choices made while generating a test case. Then, the provided byte array can be evolved with the guidance of coverage feedback: byte arrays that result in more coverage are selected as survivors to be mutated to produce new test cases.

Implementing ZEST on ELFUZZ fuzzers. Adapting ZEST to the fuzzers synthesized by ELFUZZ is straightforward: These fuzzers are evolved from the seed fuzzer in Listing 14, and their entry points are all the same as in the list. They take a random number generator and an output stream as arguments, make choices according to the random number generator, and write the generated test case to the output. The adaptation is mainly "parameterizing" the fuzzers to replace the random number generator with a provided byte array, and implement the logic that evolves the byte array, as shown by Listing 17.

Correctness validation. We have implemented the Zest adaptation on the ELFUZZ fuzzers for `cvc5`. We checked the first 10 test cases and their coverage information. We have confirmed that the implementation works as the above ZEST crux demands, i.e., test cases are according to the byte arrays, and byte arrays that result in better coverage are selected and mutated to produce new test cases. For example, suppose the size of the survivor population is set to three, and test cases achieving edge coverage of 13,666, 4,336, and 24,958 are the current survivors. If the newly generated test case achieves coverage of 9,921, it will correctly replace the second test case as a new survivor.

Limitations. Our current implementation of ZEST on ELFUZZ fuzzers is only a proof-of-concept prototype to show the extensibility of ELFUZZ . We did not apply the various optimizations mentioned

```
testcase.write('CREATE TABLE IF NOT EXISTS ')
testcase.write(TABLE_NAME + '\n')
for col in COLUMN_NAMES[:-1]:
    testcase.write(col)
    testcase.write(', ')
testcase.write(COLUM_NAMES[-1])
testcase.write(
    'INTEGER PRIMARY KEY AUTOINCREMENT);\n'
)
... # Insert values into the table
orderby_columns = []
for _ in range(3):
    orderby_columns.append([
        COLUMN_NAMES[rand() % len(COLUMN_NAMES)],
        ['ASC', 'DESC'][rand() % 2]
    ])

testcase.write('SELECT * FROM ')
testcase.write(TABLE_NAME)
testcase.write(' ORDER BY ')
for col in orderby_columns[:-1]:
    testcase.write(col[0])
    testcase.write(' ')
    testcase.write(col[1]))
    testcase.write(', ')

testcase.write(orderby_columns[-1][0]
testcase.write(' ')
testcase.write(orderby_columns[-1][1])
testcase.write(';\n')
```

Listing 16: Code that generates Listing 15

```
def collect_cov(testcase_file): ...
def looks_good(new_cov, previous_covs) -> bool: ...
def mutate_bytes(bytes) -> bytes: ...
def gen_<FORMAT>(bytes: Bytes, output: TextIO): ...

def fuzz():
  survivors = [rand_bytes()] * SURVIVOR_SIZE
  covs = []
  ... # Collect initial covs
  while True:
    select = survivors[0]
    ...
    mutant = mutate_bytes(bytes)
    gen_<FORMAT>(mutant, file)
    mut_cov = collect_cov(file)
    if looks_good(mut_cov, covs):
      survivors.pop(0)
      covs.pop(0)
      survivors.append(mutant)
      covs.append(mut_cov)
```

Listing 17: ZEST implemented on ELFUZZ fuzzers

in the paper, which prevents the implementation from being used as a practical fuzzer. For example, we use `afl-showmap` to collect the coverage information for simplicity. At the same time, the original paper implements a binary instrumentation tool to insert coverage collection code into the SUTs. Relying on `afl-cov` brings huge overhead every time when collecting the coverage feedback.