Reducing Coverage-Equivalent Inputs in Grammar-Based Fuzzing by Avoiding Recurrent Rule Sequences

JAEHAN YOON*, Sungkyunkwan University, Republic of Korea
YUNJI SEO*, Korea University, Republic of Korea
HAKJOO OH, Korea University, Republic of Korea
SOOYOUNG CHA†, Sungkyunkwan University, Republic of Korea

We present RSFuzz, a new technique to enhance grammar-based fuzzing by reducing the generation of coverage-equivalent inputs during testing. Grammar-based fuzzers apply production rules from a given grammar (e.g., forming a derivation tree) to generate well-structured inputs for the target program. However, a key limitation is that many existing fuzzers still produce a large number of "coverage-equivalent" inputs—those that revisit already explored program paths—thereby restricting their ability to uncover new bugs and improve coverage. To address this issue, RSFuzz automatically identifies recurrent sequences of production rules that lead to coverage-equivalent inputs and prevents their reuse during fuzzing. A key challenge lies in the large number of coverage-equivalent input groups, each with many inputs and corresponding derivation trees, making it difficult to identify underlying recurrent sequences. RSFuzz tackles this challenge with a customized algorithm that iteratively groups coverage-equivalent inputs, selects promising groups, and extracts recurrent sequences by abstracting derivation trees based on accumulated data while running any grammar-based fuzzer. We integrated RSFuzz with existing random, probabilistic, and grammar-coverage based fuzzers and evaluated it on 12 real-world programs using JavaScript, JSON, CSV, and Markdown input formats. Experimental results show that incorporating RSFuzz into the three fuzzers detects 121, 46, and 17 additional crashes with distinct stack traces, increases line coverage by 6.0%, 4.8%, and 3.0%, and reduces duplicate-coverage input generation by 23.3%, 28.7% and 14.9%, respectively, compared to their performance without RSFuzz.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Grammar-based Fuzzing, Software Testing

ACM Reference Format:
Jaehan Yoon, Yunji Seo, Hakjoo Oh, and Sooyoung Cha. 2026. Reducing Coverage-Equivalent Inputs in Grammar-Based Fuzzing by Avoiding Recurrent Rule Sequences. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE203 (July 2026), 22 pages. https://doi.org/10.1145/3808210

1 Introduction

Grammar-based fuzzing is a mainstream software testing technique to effectively increase code coverage and find subtle bugs [2, 22, 29, 34, 37–39, 42]. This technique uses a grammar specifying the input format of the target program under test, and aims to automatically generate highly-structured inputs by iteratively applying production rules in the grammar. Unlike mutation-based fuzzing [6, 7, 13, 32], which is another major fuzzing method that relies on initial seed inputs to

*The first and second authors contributed equally to this work.
†Corresponding author

Authors' Contact Information: Jaehan Yoon, Sungkyunkwan University, Republic of Korea, jaehan.yoon@skku.edu; Yunji Seo, Korea University, Republic of Korea, yunji990105@korea.ac.kr; Hakjoo Oh, Korea University, Republic of Korea, hakjoo_oh@korea.ac.kr; Sooyoung Cha, Sungkyunkwan University, Republic of Korea, sooyoung.cha@skku.edu.

ACM 2994-970X/2026/7-ARTFSE203
https://doi.org/10.1145/3808210

generate new inputs, grammar-based fuzzing can produce valid inputs from the specified grammar that pass the syntax parsing stage without requiring initial seed inputs. As a result, it has been actively studied for testing programs with complex input formats such as JavaScript [22, 30, 35, 43], JSON [18, 20, 34, 37], and even for grammar generation itself [5, 14–16].

A key component of grammar-based fuzzers is determining which production rules to apply from a grammar (e.g., context-free grammar) to generate useful inputs. More specifically, these fuzzers need to decide the sequence of production rules for each input, represented as a derivation tree. Some basic grammar-based fuzzers, such as Grammarinator [21], randomly select these rules as a cost-effective strategy. More recent fuzzers [37, 38, 47] adopt more sophisticated methods: Soremekun et al. [37] use probabilistic context-free grammars (PCFGs), where the selection probability of each rule is learned offline from a set of inputs. It is widely recognized that intelligently selecting production rules is crucial for the performance of grammar-based fuzzers, since the input space defined by the grammar is typically infinite [30, 37, 38].

However, our observation is that both random and probabilistic grammar-based fuzzers often generate a substantial number of "coverage-equivalent" inputs. In this work, we define an input as coverage-equivalent if it executes the same set of lines as a previously generated one, even when its concrete values differ. Generating such inputs is undesirable, as they are less likely to expose new bugs or cover previously unseen code regions. As shown in Section 2, when testing real-world JavaScript engines, 80.3% of the total inputs produced by the random fuzzer [21]were coverage-equivalent. Similarly, the probabilistic fuzzer [37], which utilizes a learned PCFG, generated 81.0% unproductive inputs. Consequently, these fuzzers tend to spend valuable testing time generating inputs that contribute little to uncovering new bugs or exploring additional code regions.

In this paper, we present RSFuzz, a novel approach that enhances grammar-based fuzzers by preventing the generation of coverage-equivalent inputs during testing. The core idea is to iteratively identify recurrent sequences of grammar production rules, defined as sequences of production-rule applications that repeatedly appear across multiple coverage-equivalent inputs during fuzzing. Since every generated input has its own sequence of applied production rules explaining how it was produced, our hypothesis is that examining this information allows us to effectively identify patterns associated with the generation of coverage-equivalent inputs. By pinpointing these patterns, RSFuzz ensures that such sequences are not reapplied during fuzzing. However, a key challenge is the sheer scale of coverage-equivalent inputs. As reported in Section 2, when the fuzzer generated 100,000 inputs on real-world JavaScript benchmarks, grouping inputs by identical line coverage resulted in about 19,729 distinct groups, each corresponding to a potential source of recurrent sequences. Even when focusing only on the 200 largest groups, each group still contained over 388 inputs on average. Thus, identifying why these inputs are coverage-equivalent requires analyzing hundreds of production rule sequences within each group.

To address this challenge, we developed a specialized algorithm that iteratively collects informative data (e.g. coverage, derivation trees) from inputs generated by the fuzzer, groups inputs by code coverage, selects key groups, and captures the recurrent sequences responsible for executing the same code regions within each selected group by abstracting their derivation trees into simplified symbol sequences. By repeating these stages, RSFuzz produces recurrent sequences tailored to each program and its grammar, thereby improving the efficiency of grammar-based fuzzing.

Experimental results demonstrate that RSFuzz markedly enhances the performance of existing grammar-based fuzzers in terms of crash-finding capability and line coverage. We integrated RSFuzz with a random fuzzer [21], a probabilistic fuzzer [37] and a grammar-coverage based fuzzer [18]. To evaluate its effectiveness, we tested RSFuzz on 12 real-world benchmarks, each with up to 77K LOC, using four input formats: JavaScript, JSON, CSV, and Markdown. The results indicate that when integrated with RSFuzz, the random, probabilistic, and grammar-coverage based fuzzers were able

Algorithm 1 Grammar-based Fuzzing

Input: Program (*pgm*), budget (*time*), grammar ($\mathbb{G}$), recurrent sequences (*RS*).
Output: A set of covered lines (*TotalL*).

```
1: procedure FUZZING(pgm, time, 𝔾, RS)
2:     TotalL ← ∅                                   ▷ A set of covered lines
3:     repeat
4:         inp, tree ← Generate(𝔾, RS)
5:         L ← Execute(pgm, inp)
6:         TotalL ← TotalL ∪ L
7:     until time expires
8:     return TotalL
```

to exclusively detect 121, 46, and 17 crashes having distinct stack traces, respectively, which were not found without RSFuzz. Furthermore, integration of RSFuzz improved line coverage by 6.0% for the random fuzzer, 4.8% for the probabilistic fuzzer, and 3.0% for the grammar-coverage based fuzzer. This improvement is attributed to RSFuzz reducing the proportion of coverage-equivalent inputs generated by the random, probabilistic, and grammar-based fuzzers by averages of 23.3% and 28.7%, and 14.9%, respectively, across our benchmark suite.

Contributions. We summarize our contributions as follows:

- **Observation**: We demonstrate that approximately 80% of inputs generated by existing grammar-based fuzzers are coverage-equivalent, repeatedly executing the same set of lines.
- **New Technique**: We present RSFuzz, a complementary technique that enhances grammar-based fuzzers by automatically identifying factors associated with coverage-equivalent inputs. Our main technical contribution is a domain-specific algorithm that selects key groups from numerous coverage-equivalent input groups and derives recurrent sequences that should not be applied in certain orders, thereby reducing the likelihood of generating such inputs.
- **Evaluation**: We show the effectiveness of RSFuzz by integrating it into three grammar-based fuzzers and comparing their performance with the original fuzzers on 12 real-world programs. We make our tool (RSFuzz) publicly available[1].

2 Preliminaries

2.1 Basic Grammar-based Fuzzing

Algorithm 1 describes a basic grammar-based fuzzing algorithm. The algorithm takes four inputs: a program under test (*pgm*), a time budget (*time*), a grammar ($\mathbb{G}$), and recurrent sequences (*RS*). For now, let us ignore the last argument, *RS*, which will be explained later. The algorithm first initializes a total set of covered lines, *TotalL*, as an empty set. It then iteratively generates both the input (*inp*) and the derivation tree (*tree*) that includes the used production rules from the grammar $\mathbb{G}$ (line 4), executes the program with the input, and returns the set L of covered lines (line 5). This fuzzing algorithm repeats this process until the given budget is exhausted and finally returns the total set *TotalL* of lines covered so far.

A key component in Algorithm 1 is the Generate function that determines how to generate the input from a given grammar $\mathbb{G}$ (e.g. context-free grammar). For example, consider a simplified JSON grammar, $\mathbb{G}_J$, with nine production rules:

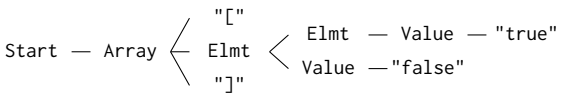
```
Start → Array
Array → "[" Elmt "]" | "[" Array "]" | "[]"
Elmt → Elmt Value | Value
Value → "true" | "false" | ""
```

[1] https://github.com/skkusal/rsfuzz

Table 1. The percentage of coverage-equivalent inputs among the total inputs generated during testing.

Input Format	Program	RandF	ProbF [37]	Input Format	Program	RandF	ProbF [37]
JavaScript	Jsish	**94.3%**	**96.7%**	JSON	Genson	**83.1%**	**97.3%**
	Rhino	**84.1%**	**85.9%**		Gson	**82.3%**	**99.1%**
	QuickJS	**82.1%**	**76.1%**		JsonToJava	**89.4%**	**76.2%**
	JerryScript	**60.6%**	**65.4%**		Argo	**83.6%**	**98.2%**

In this grammar, nonterminals such as Start and Array are denoted by names that begin with an uppercase letter, with Start being the starting nonterminal. This starting nonterminal serves as the root node for the derivation tree of each input. The function repeatedly expands leaf nodes based on the grammar until no nonterminals are left to expand. One simplistic method for generating inputs is to randomly apply the production rules of the grammar. For instance, possible inputs include "[]", "[[]]", "[true]", "[true false]". The last input is derived by the following derivation tree:



This tree illustrates how the Generate function iteratively produces inputs by expanding nonterminals until a complete input is formed.

A more recent and sophisticated approach [37] employs a probabilistic context-free grammar (PCFG) to generate input. With a PCFG, each production rule of a given grammar is selected stochastically. For instance, assume that inputs containing "false" from the aforementioned grammar $\mathbb{G}_J$ are more likely to increase coverage or trigger bugs. In such cases, the probabilistic method would learn a PCFG that favors selecting "false" over "true" or "" as:

```
Value → (20%) "true" | (70%) "false" | (10%) ""
```

By using this learned PCFG, existing methods have successfully generated many bug-triggering inputs in real-world benchmarks [37].

2.2 Limitation

Current grammar-based fuzzers often generate a large number of coverage-equivalent inputs during testing. In this work, we define coverage-equivalent inputs as those that execute the same set of lines as previously generated ones, even if their values differ. For instance, let us assume that two inputs, "[]" and "[[]]", execute the same set of lines despite having distinct values. In this case, we classify the second input as coverage-equivalent, since re-executing the same set of lines is unlikely to reveal bugs or increase code coverage.

Table 1 displays the percentage of coverage-equivalent inputs among 100,000 generated by random and probabilistic grammar-based fuzzers across eight real-world programs, ranging from 1.5K to 77K LoC, with two input formats: JavaScript and JSON. The random fuzzer [21] (RandF[2]), which randomly applies the production rules of the grammar, yields approximately 80.3% and 84.6% coverage-equivalent inputs while testing the JavaScript and JSON benchmarks, respectively. Interestingly, even the probabilistic fuzzer [37] (ProbF) with the learned PCFG produced about 81.0% and 92.7% coverage-equivalent inputs on average for the JavaScript and JSON benchmarks, respectively. This might stem from the higher sampling probabilities of certain production rules within the PCFG, leading to the generation of many inputs that cover similar code regions.

[2]Although both fuzzers have original names, we use the terms RandF and ProbF throughout this paper to concisely highlight their distinguishing characteristics.

Table 2. Number of coverage-equivalent sets (#Sets) and average sizes of the top 200 sets (Avg.Size) for the random fuzzer (RandF) and probabilistic fuzzer (ProbF).

Program	RandF		ProbF [37]		Program	RandF		ProbF [37]	
	# Sets	Avg. Size	# Sets	Avg. Size		# Sets	Avg. Size	# Sets	Avg. Size
Jsish	5,708	460	3,315	379.6	Genson	16,921	476.7	2,703	494.6
Rhino	15,897	405.9	14,072	374.4	Gson	17,702	410.7	893	495.1
QuickJS	17,901	388.1	23,914	400.6	JsonToJava	10,586	352.5	23,828	262.6
JerryScript	39,412	296.7	34,602	384.7	Argo	16,410	301.5	1,816	495.8

2.3 Goal

The goal of this paper is to reduce the generation of coverage-equivalent inputs during grammar-based fuzzing, thereby enhancing its performance. The key idea is to automatically identify recurrent sequences of production rules associated with coverage-equivalent inputs generated while running a grammar-based fuzzer. Each generated input has a unique sequence of applied production rules describing its derivation. Our hypothesis is that examining these sequences can effectively identify patterns associated with the generation of coverage-equivalent inputs. Consequently, we define a *recurrent sequence* as a sequence of production rules within the grammar responsible for producing these coverage-equivalent inputs. For example, assume that the two inputs "[]" and "[[]]", generated from grammar $\mathbb{G}_\jmath$ as described in Section 2.1, cover the same set of code lines that have already been traversed. The sequences of production rules applied for generating these inputs are:

```
[]: Start → Array → "[]"
[[]]: Start → Array → "[" Array "]" → "[[]]"
```

Our objective is to ensure that production rules commonly responsible for generating these coverage-equivalent inputs are avoided in subsequent input generations. One potential recurrent sequence for achieving this goal is: Start → Array → "[]". This sequence is designed to prevent the reuse of the rule Array → "[]" after the initial rule Start → Array is applied. Given a program and its grammar, our aim is to automatically capture a set of recurrent sequences that lead to the generation of coverage-equivalent inputs during fuzzing.

However, efficiently identifying "meaningful" recurrent sequences is far from straightforward in practice. As shown in Table 2, when generating 100,000 inputs for the eight benchmark programs, both the grammar-based fuzzers, RandF and ProbF, produced approximately 17,567 and 13,143 sets, respectively, where each set contained inputs covering the same lines. Each set represents a candidate for capturing recurrent sequences, making it practically infeasible to analyze all of them. Even when restricting attention to the top 200 coverage sets containing the largest numbers of coverage-equivalent inputs, the challenge remains: each set still includes, on average, 387 inputs for RandF and 411 for ProbF.

Thus, identifying why these inputs are coverage-equivalent requires analyzing hundreds of production rule sequences within each set. These observations highlight a key challenge: identifying appropriate recurrent sequences is particularly difficult.

3 Our Approach

3.1 Overview

Figure 1 illustrates an overview of how our approach, RSFuzz, interacts with a grammar-based fuzzer. At a high-level, RSFuzz takes a program under test and its grammar as input and aims to iteratively generate a set of recurrent sequences based on data accumulated during fuzzing. Specifically, RSFuzz performs four main steps: (1) collecting informative data (e.g., line coverage) from inputs generated by the fuzzer, (2) grouping these inputs based on their code coverage, (3)

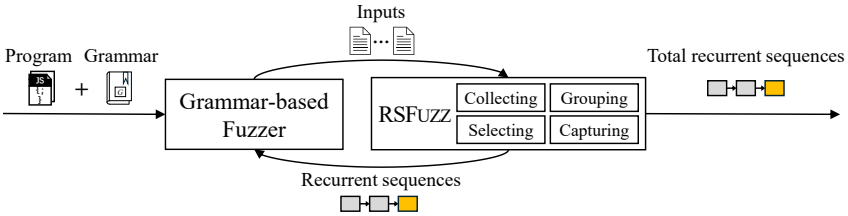


Fig. 1. Overview of RSFuzz

selecting groups from which recurrent sequences will be extracted, and (4) capturing the recurrent sequences responsible for repeatedly executing the same code regions within each selected group.

The key novelty of RSFuzz lies in addressing two significant challenges in the "selecting" and "capturing" steps. The first challenge is deciding which groups to target for capturing recurrent sequences. An extreme approach—capturing recurrent sequences from every group—has been shown to be inefficient, as demonstrated in Section 4.4. The second challenge involves determining how to extract recurrent sequences from the selected groups. A straightforward approach—identifying sequences of production rules commonly used to generate all inputs in each group—was also found to be cost-ineffective, as shown in Section 4.4.

Once the given budget expires, RSFuzz returns the complete set of recurrent sequences accumulated through these iterative interactions. We then perform grammar-based fuzzing (Algorithm 1) utilizing these recurrent sequences. A typical usage scenario is to first run RSFuzz (Algorithm 2) to generate recurrent sequences for half of the total budget, and then perform grammar-based fuzzing (Algorithm 1) with these sequences during the remaining budget.

3.2 RSFuzz

Algorithm 2 provides a detailed explanation of how our approach generates a set *RS* of recurrent sequences within a given budget. At line 1, the algorithm initializes three sets: *RS*, *PrevL* (covered lines), and *TotalG* (total groups), all as empty sets. The parameter value, η_{num}, is set to 2000, indicating the number of inputs to be generated in each iteration of the outer loop.

3.2.1 Collecting. The first step aims to gather the data D, which will be used to generate recurrent sequences in subsequent steps. To collect this data, Algorithm 2 repeatedly generates and executes inputs η_{num} times (lines 6–9). We define an element d in D as a pair $(tree, L)$, where *tree* is the derivation tree used to generate an input, and L is the set of lines executed by that input. At line 7, the Generate function produces both the input (*inp*) and the derivation tree (*tree*) based on the grammar ($\mathbb{G}$) and the recurrent sequences (*RS*); we explain later in Section 3.3 how this function generates inputs using both $\mathbb{G}$ and *RS*. The algorithm then gets a set of covered lines (L) by running the program with the input, and accumulates the pair $(tree, L)$ into D. Here, D is a multiset ($\{\!\{\}\!\}$) that allows duplicate elements, enabling the tracking of how often the exact same *tree* is generated.

3.2.2 Grouping. The second step aims to construct candidate groups from the set D (lines 12–17), which will be used to capture recurrent sequences in the next step. To achieve this, Algorithm 2 partitions D into subsets, where all elements within each subset G_θ share the same set L of covered lines. We selected line coverage as the grouping criterion because it provides a practical balance between overly fine-grained metrics, such as path coverage, and coarser metrics, such as method coverage, thereby enabling the formation of meaningful groups for recurrent-sequence extraction. Formally, assuming that an arbitrary element of G_θ is a pair $(tree, L)$, all elements in G_θ satisfy the following condition: $\forall(tree', L') \in G_\theta,\ L' = L$. In this work, each group G_θ is treated as a candidate for extracting recurrent sequences of production rules, as it represents the key reason for repeatedly

Algorithm 2 RSFuzz

Input: Program (*pgm*), grammar ($\mathbb{G}$), and budget (*time*).
Output: recurrent sequences (*RS*).

```
1: ⟨RS, PrevL, TotalG⟩ ← ⟨∅, ∅, ∅⟩                          ▷ (rs, L, G_θ) ∈ RS
2: η_num ← 2000
3: repeat
4:     /* Step 1: Collecting */
5:     D ← ∅
6:     for i = 1 to η_num do
7:         inp, tree ← Generate(𝔾, RS)
8:         L ← Execute(pgm, inp)
9:         D ← D ∪ {{ (tree, L) }}
10:
11:    /* Step 2: Grouping */
12:    for each (tree, L) ∈ D do
13:        if L ⊄ Cov(TotalG) then                          ▷ Check if L is newly covered.
14:            G_θ ← {{ (tree, L) }}                        ▷ Create new group G_θ with θ = 1.
15:        else
16:            G_θ ← G_θ ∪ {{ (tree, L) }}                  ▷ Update G_θ covering L.
17:        TotalG ← TotalG ∪ {G_θ}
18:
19:    NewL ← {l ∈ L | (_, L) ∈ D}
20:    if |NewL \ PrevL| = 0 or RS = ∅ then
21:        /* Step 3: Selecting */
22:        NewG, ReG ← Select(TotalG, D, RS)                ▷ NewG, ReG ⊆ TotalG
23:
24:        /* Step 4: Capturing */
25:        RS = Capture(NewG, ReG, RS)
26:    PrevL ← PrevL ∪ NewL
27: until time expires
28: return RS
```

reaching the same set of lines. The value θ of G_θ will be used later in the Capture step, where it determines the extent to which general recurrent sequences are captured within the group G_θ. We will explain the role in detail in the Capture step.

At line 12, for each element (*tree*, *L*) in *D*, Algorithm 2 checks whether *L* contains any lines not in the total set of covered lines. This total set can be calculated by the Cov function as follows:

$$\text{Cov}(\mathit{TotalG}) = \{L \mid (\_, L) \in G \wedge G \in \mathit{TotalG}\}$$

Intuitively, the function takes the total set *TotalG* of groups constructed so far and returns the total set of lines collectively covered by *TotalG*. If *L* contains newly covered lines, a new group G_θ is initialized at line 14 with the element (*tree*, *L*) and the θ value is set to one. Conversely, if the set *L* has already been reached, it adds the element to the existing group G_θ associated with *L* (line 16). Finally, the algorithm adds a new group or an updated group to the total set *TotalG* of groups (line 17). Note that each group G_θ is deliberately defined as a multiset ($\{\!\{\}\!\}$), allowing duplicated elements to collect identical derivation trees that contribute to executing the same code lines.

After completing the grouping process, Algorithm 2 determines whether to proceed with the remaining two steps to update recurrent sequences (lines 19–20). Since this update in every iteration (lines 3–27) is computationally expensive, Algorithm 2 attempts to do so only when line coverage no longer increases. To achieve this, it collects the set *NewL* of covered lines from the newly accumulated data *D* in the current iteration (line 19). It then compares *NewL* with *PrevL*, the set of lines collectively covered in previous iterations (line 20). If no new lines have been covered, the algorithm proceeds with the remaining two steps to update recurrent sequences. Also, if *RS* is empty (first iteration), it generates recurrent sequences. At the end of each iteration, the set *NewL* is merged to *PrevL* (line 26).

3.2.3 Selecting. In the third step, Algorithm 2 (RSFuzz) aims to select the most suitable groups for extracting recurrent sequences from the total set *TotalG* of groups (line 22), which was constructed in the previous step. This selection step is critical because the total number of groups is so large that extracting recurrent sequences from all of them would significantly degrade the fuzzing performance, as demonstrated in Section 4.4. RSFuzz selects two sets of suitable groups, *NewG* and *ReG*, using two methods.

Selecting NewG. RSFuzz chooses new groups, *NewG*, that are large in size but have not yet captured recurrent sequences, thereby minimizing coverage-equivalent inputs generated during fuzzing. Specifically, it identifies uncaptured groups using two sets: *TotalG* and *RS* (line 22). As defined at line 1 in Algorithm 2, each element in *RS* is a triple (rs, L, G_θ), where G_θ represents the group used to extract the recurrent sequence *rs*. Using *TotalG* and *RS*, these uncaptured groups (*UG*) can be formally defined as:

$$UG = TotalG \setminus \{G_\theta \mid (\_, \_, G_\theta) \in RS\}$$

Intuitively, *UG* denotes the subset of *TotalG* that has not been utilized for generating recurrent sequences. After identifying *UG*, RSFuzz sorts the groups by their sizes in descending order and selects those with the largest gap from the next group. The intuition is that larger groups provide greater benefits by preventing coverage-equivalent input generation through recurrent sequences.

Example 3.1. Assume that the set *UG* consists of four uncaptured groups, $\{UG_1, UG_2, UG_3, UG_4\}$, with the following sizes:

$$|UG_1| = 100, |UG_2| = 90, |UG_3| = 20, |UG_4| = 10$$

In this example, we choose the top two groups, UG_1 and UG_2, as *NewG*, since the largest gap exists between UG_2 and the next group.

Selecting ReG. RSFuzz re-selects the groups, *ReG*, that have previously captured recurrent sequences but failed to prevent duplicate-coverage input generation with them. Since these groups contain a large number of duplicated-coverage inputs, it is essential to replace their recurrent sequences with more effective ones. RSFuzz identifies this failure in each group using three sets: *TotalG*, *D*, and *RS* (line 22). Specifically, leveraging the data *D* accumulated during the current iteration between lines 6 and 9, RSFuzz constructs the set *CL* as:

$$CL = \{L \mid (tree, L) \in D\}$$

where each element in *CL* represents a set of covered lines. RSFuzz then determines *ReG* by verifying whether L', the set of lines covered in each group G_θ, exists in *CL* as:

$$ReG = \{G_\theta \mid L' \in CL \wedge (\_, L', G_\theta) \in RS\}$$

Intuitively, each element in *ReG* represents a group G_θ whose inputs have been re-generated, despite its intended role of capturing recurrent sequences to prevent coverage-equivalent inputs.

3.2.4 Capturing. The objective of the Capture step is to extract effective recurrent sequences from each group G_θ in both *NewG* and *ReG*, which were selected in the previous step. Recall that each group G_θ, which achieves the same set of covered lines *L*, is defined:

$$G_\theta = \{(tree_1, L), (tree_2, L), \cdots, (tree_n, L)\}$$

where *tree* represents the derivation tree used to generate an input, and *L* is the set of lines covered by that input. From G_θ, RSFuzz first collects these derivation trees (e.g., $\{tree_1, tree_2, \cdots, tree_n\}$) and extracts recurrent sequences of production rules from them—the key factor contributing to covering the same set *L*.

However, extracting recurrent sequences is non-trivial. For example, consider capturing recurrent sequences from two derivation trees in Figure 2, where inputs generated from these trees, "[false true]" and "[false true ""]", cover the same set of lines. A simple approach is to define the sequence of production rules that appear simultaneously in both trees, starting from the first rule (Start → Array), as a recurrent sequence, as follows:

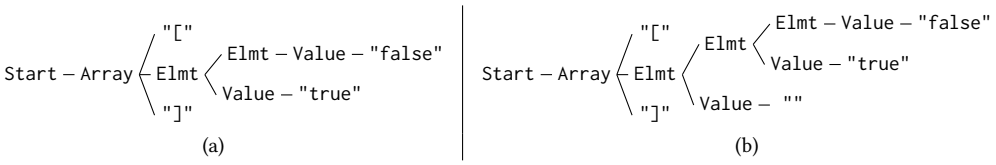


Fig. 2. Derivation trees of two coverage-equivalent inputs

```
Start → Array → "[" Elmt "]" → "[" Elmt Value "]"
```

Unfortunately, this recurrent sequence is overly general, preventing the generation of all multi-element arrays and ultimately reducing fuzzing performance. Moreover, in real-world fuzzing scenarios, identifying the appropriate recurrent sequences from complex derivation trees remains particularly challenging.

To address this challenge, we capture recurrent sequences from each group G_θ through the following three processes: (1) transforming each derivation tree in G_θ into simplified symbol sequences, (2) clustering these sequences based on their frequencies within G_θ, and (3) extracting recurrent sequences from the top-θ clusters in the group G_θ, where the θ value varies depending on each group.

Capturing from NewG. We now describe in detail how recurrent sequences are captured from each group G_θ in *NewG*. The Capture function in Algorithm 2 first divides a derivation tree into symbol sequences. In this work, we define a symbol sequence as a single path that reaches a terminal symbol and consists of a series of symbols. For example, the derivation tree in Figure 2-(a) is split into four symbol sequences as:

```
Start → Array → "[", Start → Array → "]",
Start → Array → Elmt → Value → "true",
Start → Array → Elmt → Elmt → Value → "false"
```

Next, RSFuzz simplifies these sequences by eliminating consecutive duplicated symbols. In this case, only the last sequence is modified by removing the duplicate "Elmt → Elmt" transition, as:

```
Start → Array → "[", Start → Array → "]",
Start → Array → Elmt → Value → "true",
Start → Array → Elmt → Value → "false"
```
(1)

By representing each derivation tree as these simplified sequences, RSFuzz can treat structurally different trees with similar characteristics as equivalent. Applying this simplification method, the tree in Figure 2-(b) is transformed into five symbol sequences as:

```
Start → Array → "[", Start → Array → "]",
Start → Array → Elmt → Value → "true",
Start → Array → Elmt → Value → "false",
Start → Array → Elmt → Value → ""
```
(2)

where the first four sequences are identical to those in (1).

The Capture function then clusters all generated symbol sequences from (1) and (2) based on their frequency. To achieve this, we employ the MeanShift clustering algorithm [8], which automatically determines the optimal number of clusters based on the accumulated data. For instance, the sequences in (1) and (2) are divided into two clusters. The first cluster contains as:

```
{Start → Array → "[", Start → Array → "]",
Start → Array → Elmt → Value → "true",
Start → Array → Elmt → Value → "false"}
```
(3)

These sequences are grouped together as they appear in both derivation trees. The second cluster contains only the last sequence from (2), which appears once in Figure 2:

$$\{\texttt{Start} \rightarrow \texttt{Array} \rightarrow \texttt{Elmt} \rightarrow \texttt{Value} \rightarrow \texttt{""}\}$$

In this work, we also considered alternative clustering algorithms, such as DBSCAN [12] and OPTICS [1]. However, we observed that these methods were generally less effective than MeanShift, since their noise filtering mechanisms often discarded meaningful symbol sequences as noise.

Finally, using the parameter θ of the group G_θ, the Capture function extracts all symbol sequences in top-θ clusters as recurrent sequences. Recall that the θ value for every group in *NewG* is set to one (line 14). This means that recurrent sequences are extracted only from the top cluster, which contains the most frequently occurring sequences in the group. Consequently, the generated recurrent sequences are identical to those in (3). The role of these sequences is to prevent the generation of inputs containing parentheses (e.g., "[]") and terminal symbols "false" and "true" within the parentheses. At line 25, Algorithm 2 adds an element containing these sequences, *rs*, to the set *RS*, i.e., $(rs, L, G_\theta) \in RS$. Intuitively, L represents the set of covered lines by inputs generated from two trees in Figure 2, while G_θ denotes the group used for generating *rs*.

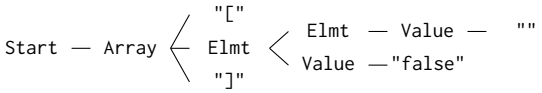
Capturing from ReG. To capture recurrent sequences in *ReG*, RSFuzz basically employs a method similar to the one used for extracting recurrent sequences in *NewG*. Recall that *ReG* consists of groups that have previously extracted recurrent sequences but failed to prevent duplicated-coverage input generation. For each group G_θ in *ReG*, RSFuzz appropriately updates the value of θ, allowing it to re-capture more meaningful sequences. In this work, increasing θ corresponds to increasing the number of top-θ clusters in each group G_θ, which are used for capturing recurrent sequences.

To determine the proper value of θ for each group G_θ, Algorithm 2 (RSFuzz) compares rs_{prev} with rs_{new}. Here, rs_{prev} represents the previously extracted recurrent sequences from G_θ, while rs_{new} represents the new sequences generated from the newly updated group G_θ through the loop between lines 12–17. Formally, rs_{prev}, which consists of the previously extracted recurrent sequences from G_θ covering the set L' of lines, is defined as:

$$rs_{prev} = \{rs_i \in rs \mid L' = L \wedge (rs, L, G) \in RS\}$$

If rs_{prev} and rs_{new} differ, RSFuzz directly employs the newly generated recurrent sequences rs_{new} without updating the θ. The rationale behind this approach is that as more data accumulates, RSFuzz naturally captures more generalized recurrent sequences (rs_{new}) for G_θ, even without updating θ.

Example 3.2. Assume that rs_{prev} represents the recurrent sequences in (3), which were generated from the two derivation trees in Figure 2. Now, suppose that RSFuzz generates new recurrent sequences when an additional derivation tree is incorporated into these two trees, as follows:



This tree is then transformed into four symbol sequences using our simplification method as:

$$\begin{array}{l}\texttt{Start} \rightarrow \texttt{Array} \rightarrow \texttt{"["},\ \texttt{Start} \rightarrow \texttt{Array} \rightarrow \texttt{"]"},\\ \texttt{Start} \rightarrow \texttt{Array} \rightarrow \texttt{Elmt} \rightarrow \texttt{Value} \rightarrow \texttt{""},\\ \texttt{Start} \rightarrow \texttt{Array} \rightarrow \texttt{Elmt} \rightarrow \texttt{Value} \rightarrow \texttt{"false"}\end{array} \tag{4}$$

Next, RSFuzz clusters all generated symbol sequences from (1), (2), and (4) based on their frequency. Finally, as new recurrent sequences rs_{new}, RSFuzz selects the top cluster containing the most

frequently occurring sequences in the group:

$$\begin{array}{l}\texttt{Start} \rightarrow \texttt{Array} \rightarrow \texttt{"["},\ \texttt{Start} \rightarrow \texttt{Array} \rightarrow \texttt{"]"},\\ \texttt{Start} \rightarrow \texttt{Array} \rightarrow \texttt{Elmt} \rightarrow \texttt{Value} \rightarrow \texttt{"false"}\end{array} \tag{5}$$

In this example, since rs_{prev} in (3) and rs_{new} in (5) differ, RSFuzz updates the recurrent sequences for group G_θ, replacing rs_{prev} with rs_{new}. Intuitively, this indicates that as more data accumulates, RSFuzz successfully captures more generalized recurrent sequences from G_θ.

Conversely, if rs_{new} remains unchanged as rs_{prev} despite the accumulation of new data, RSFuzz increases the value of θ for group G_θ by one. Increasing θ expands the number of top-θ clusters in G_θ, which are used for extracting recurrent sequences. In other words, RSFuzz incorporates more symbol sequences as recurrent sequences. For example, in the two derivation trees shown in Figure 2, increasing θ from 1 to 2 results in all two clusters generated from (1) and (2) being utilized for recurrent sequence extraction, causing all five symbol sequences in (2) to be considered redundant. However, in this case, directly using them as a single recurrent sequence fails to prevent the derivation tree in Figure 2-(a) from being re-generated, as it only restricts the generation of inputs that simultaneously contain "", "false", and "true" in an array. To address it, RSFuzz computes the intersection between the symbol sequences derived from Figure 2-(a) and those in (2), then applies the intersected sequences, which are in (1), as recurrent sequences.

Algorithm 2 (RSFuzz) iteratively performs the process of collecting, grouping, selecting, and capturing stages until the allotted time budget (*time*) is exhausted. Upon completion, it returns the set *RS*, which is the recurrent sequences accumulated during the process.

3.3 Fuzzing with Recurrent Sequences

After obtaining recurrent sequences from Algorithm 2, we now perform grammar-based fuzzing (Algorithm 1), using these sequences to guide rule expansion decisions during input generation. Let us delve into how the Generate function in Algorithm 1 generates each input while utilizing both grammar $\mathbb{G}$ and recurrent sequences *RS*. First, let us define I as the set of all possible pairs (*inp*, *tree*) that the Generate function can produce without using recurrent sequences, i.e., Generate($\mathbb{G}, \emptyset$):

$$I = \{(inp_1, tree_1), (inp_2, tree_2), \cdots (inp_n, tree_n)\}$$

where an element (*inp*, *tree*) in I consists of both the input derived from the grammar $\mathbb{G}$ and the derivation tree used to produce *inp*. When the Generate function employs recurrent sequences, e.g., Generate($\mathbb{G}$, *RS*), it generates one of the pairs (*inp*, *tree*) that belong to the following set I_R as:

$$I_R = \{(inp, tree) \in I \mid \bigwedge_{(rs,\_,\_) \in RS} (rs \not\subseteq \text{Simple}(tree))\}$$

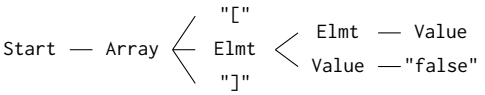
where the Simple function denotes our simplification method that transforms the derivation tree into a set of simplified symbol sequences.

That is, *inp* in I_R is generated from a derivation tree *tree* such that *tree* does not simultaneously include all symbol sequences in any recurrent sequence *rs* from *RS*.

Example 3.3. Assume that *RS* contains a recurrent sequence with the following two sequences:

$$\{\texttt{Start} \rightarrow \texttt{Array} \rightarrow \texttt{Elmt} \rightarrow \texttt{Value} \rightarrow \texttt{"true"},\ \texttt{Start} \rightarrow \texttt{Array} \rightarrow \texttt{Elmt} \rightarrow \texttt{Value} \rightarrow \texttt{"false"}\}$$

This recurrent sequence prevents inputs with an array containing both "true" and "false". Suppose that the Generate function attempts to derive the nonterminal symbol (Value) into a terminal symbol in the following derivation tree:



In this case, since the symbol sequence Start → Array → Elmt → Value →"false" is already presented in this tree, the Generate function prevents Value from deriving to "true" based on the recurrent sequence. As a result, the Generate function will produce inputs such as ["false" "false"] and ["" "false"], but not ["true", "false"]. This method ensures that the derivation tree does not simultaneously include all symbol sequences from any recurrent sequence, effectively preventing coverage-equivalent input generation.

4 Experiments

In this section, we experimentally evaluate RSFuzz to answer the following four research questions:

(1) **Effectiveness**: How effectively does our approach (RSFuzz) enhance grammar-based fuzzing in terms of crash-finding ability and line coverage?
(2) **Reduction of coverage-equivalent inputs**: To what extent does RSFuzz reduce the generation of coverage-equivalent inputs?
(3) **Efficacy of each step**: How significantly does each step in RSFuzz contribute to its overall performance?
(4) **Reusability**: Are the recurrent sequences derived by RSFuzz for a specific program applicable to different programs with the same input format?

We implemented RSFuzz on top of three distinct grammar-based fuzzers [18, 21, 37]. All experiments were conducted on a machine equipped with an Intel Xeon Gold 6230R processor.

4.1 Experimental Settings

4.1.1 Baselines. We conducted our evaluation using three baseline fuzzers—RandF, ProbF, and Tribble—and their corresponding variants integrated with RSFuzz: RandF+RSFuzz, ProbF+RSFuzz, and Tribble+RSFuzz. RandF is a random fuzzer that generates inputs by arbitrarily applying production rules from a given grammar. More precisely, RandF corresponds to our re-implementation of Grammarinator [21], which follows the ANTLR-based generation model and sets the cooldown hyperparameter to its default value of 1.0, resulting in uniform random selection of production rules. ProbF [37] is a probabilistic fuzzer that produces inputs by sampling production rules from a learned probabilistic context-free grammar (PCFG). We employed the two PCFG construction methods described in [37] without modification: (1) learning from common inputs to capture frequently used rule patterns, and (2) inverting those probabilities to promote structurally rare inputs. Using these methods, along with 10,000 sample inputs generated by a random fuzzer, we obtained two distinct PCFGs for each input format (e.g., JSON). The probabilistic fuzzer was executed with both PCFGs, and the best result was reported as ProbF. Tribble [18] is a grammar-based fuzzer designed to maximize grammar coverage (i.e., k-path coverage) by iteratively selecting a target k-path and deriving inputs that cover it at minimal cost, where a k-path denotes a sequence of k consecutive production rules in a grammar. In our experiments, we set k to 5, the best-performing value reported in Tribble's evaluation. Additionally, whenever Tribble achieves 100% k-path coverage within the total testing budget (e.g., 24 hours), it restarts the process and continues generating inputs to reach 100% k-path coverage again, thereby utilizing the remaining time effectively.

RandF+RSFuzz, ProbF+RSFuzz and Tribble+RSFuzz are variants of the baseline fuzzers that incorporate recurrent sequences produced by RSFuzz, involving two stages. First, we executed RSFuzz (Algorithm 2) on top of each baseline fuzzer for half of the total time budget to obtain recurrent sequences. Then, we ran each baseline fuzzer using the extracted recurrent sequences (Algorithm 1) for the remaining budget. For Tribble+RSFuzz, we ensured that Tribble+RSFuzz could still achieve 100% k-path coverage—Tribble's objective—by not applying recurrent sequences

Table 3. 12 Benchmark programs.

Input Format	Programs (LoC)	Source	Input Format	Programs (LoC)	Source
JavaScript	Jsish-2.0 (77.4K)	[11, 19]	JSON	Genson-1.4 (4.8K)	[10, 17, 18, 37]
	Rhino-1.7.10 (42.0K)	[10, 27, 35, 37, 44]		Gson-2.8.5 (4.4K)	[10, 17, 20, 37]
	QuickJS-2020.09 (40.4K)	[11, 19, 44]		JsonToJava-1.0.0 (3K)	[10, 17, 37]
	JerryScript-2.4.0 (33.4K)	[11, 19, 21, 31, 43, 44]		Argo-5.4 (1.5K)	[10, 17, 18, 37]
CSV	jackson-csv-2.9.8 (3.2K)	[17, 18]	Markdown	CommonMark-0.11.0 (2.2K)	[17, 18]
	super-csv-2.4.0 (1.7K)	[17, 18]		Txtmark-0.13 (1.9K)	[17, 18]

to the specific derivations required to cover the target k-path, while applying them elsewhere as usual. In summary, we evaluated the performance of each baseline fuzzer with and without RSFuzz.

4.1.2 Benchmarks and Time Budgets. We used 12 benchmark programs across four different input formats. To construct our benchmark suite, we followed two criteria. First, we chose JavaScript, JSON, CSV and Markdown, which are the most commonly used input formats in previous works on fuzzing [10, 11, 17–21, 27, 31, 35, 37, 43, 44]. Second, among benchmarks using these formats, we included only those used at least twice in prior work. Notatbly, most benchmarks (8 out of 12) are not standalone parsers but non-trivial functionality beyond parsing. Specifically, the four JavaScript benchmarks are full-fledged engines with substantial interpreter and runtime components beyond parsing; the four benchmarks—Genson, Gson, CommonMark, and Txtmark—also include additional processing logic beyond parsing, as noted in [18]. All four grammars we employed are unambiguous, meaning that each valid input can be derived from exactly one derivation tree. Table 3 provides details on the input format, benchmark program, total number of lines of code measured using `jacoco` [40] and `gcov` [41], and source from which each program was obtained.

For our experiments, we allocated a 24-hour time budget per program, which exceeds the durations used in evaluations with three baseline fuzzers [18, 21, 37]. More specifically, the evaluation for the probabilistic fuzzer [37] was not based on time, but rather on the number of inputs generated, which was capped at 100,000. In contrast, running the baseline fuzzers combined with RSFuzz for 24 hours generated an average of 1,648K inputs for the four JavaScript benchmarks and 2,617K inputs for the four JSON benchmarks. Thus, each fuzzer was executed for a significantly longer period than in prior evaluations, enabling a more comprehensive assessment of its effectiveness. Within the 24-hour budget, RSFuzz interacts with the baseline fuzzer by spending the first 12 hours generating recurrent sequences (*RS*) using Algorithm 2. The baseline fuzzer then uses the remaining time to generate inputs based on those sequences (e.g., running Algorithm 1 with *RS*). Each experiment was repeated five times, and we reported the average results.

4.2 Effectiveness

4.2.1 Crash-Finding Ability. Running grammar-based fuzzers with recurrent sequences produced by RSFuzz significantly enhanced their crash-finding abilities. Table 4 reports the number of crashes with distinct stack traces detected by each fuzzer, following previous works [10, 18, 18, 35, 37, 38], which evaluates crash-finding capabilities based on unique stack traces. Overall, RandF+RSFuzz found 236 crashes with distinct stack traces across the seven benchmarks, whereas RandF detected only 117. Similar trends were observed for the probabilistic fuzzer and Tribble: incorporating RSFuzz increased the number of crashes from 103 to 145 for the ProbF and from 28 to 45 for Tribble.

Table 4 provides details on each benchmark program, the exception types, the number of crashes with different stack traces detected by each fuzzer, and the number of crashes exclusively identified by each. All results were obtained over 120 hours per fuzzer (24 hours × 5 repetitions). We

Table 4. The number of crashes with distinct stack traces detected during fuzzing. R: Crashes detected by the random fuzzer. P: Crashes detected by the probabilistic fuzzer. T: Crashes detected by the Tribble. R+RS, P+RS and T+RS : Crashes detected when RSFuzz is integrated.

Program	Exception Type	R+RS	R	R+RS\ R	R\ R+RS	P+RS	P	P+RS\ P	P\ P+RS	T+RS	T	T+RS\ T	T\ T+RS
Rhino	java.lang.IllegalArgument	**2**	-	2	-	**2**	-	2	-	-	-	-	-
	java.lang.IllegalState	**9**	3	6	-	**3**	2	1	-	**15**	11	4	-
	java.lang.NullPointer	**1**	**1**	-	-	-	-	-	-	**2**	**2**	-	-
Genson	java.lang.NullPointer	**14**	7	7	-	**8**	6	2	-	**6**	2	4	-
	java.lang.NumberFormat	**8**	7	1	-	**8**	3	5	-	-	-	-	-
Gson	java.lang.ClassCast	**6**	5	1	-	**6**	4	2	-	**1**	-	1	-
	java.lang.IllegalArgument	**11**	9	2	-	**11**	-	11	-	-	-	-	-
	java.lang.NullPointer	**1**	**1**	-	-	**1**	**1**	-	-	**1**	**1**	-	-
JsonToJava	java.lang.ArrayIndexOutOfBounds	**18**	9	9	-	**17**	13	4	-	**2**	1	1	-
	java.lang.IllegalArgument	**27**	17	10	-	**26**	22	4	-	**2**	1	1	-
	java.lang.NumberFormat	**90**	22	70	2	-	-	-	-	**4**	3	1	-
Super-CSV	java.lang.NullPointer	**2**	**2**	-	-	**2**	**2**	-	-	**2**	**2**	-	-
CommonMark	java.lang.StackOverflow	**14**	10	4	-	**13**	9	5	1	**3**	-	3	-
Txtmark	java.lang.StringIndexOutOfBounds	**33**	24	9	-	**48**	41	10	3	**7**	5	2	-
Total		**236**	117	121	2	**145**	103	46	4	**45**	28	17	0

deliberately filtered out syntax-related errors (e.g., InvalidSyntax), as these are typically considered less critical than semantic errors [24].

Across the seven benchmarks, integrating RSFuzz increased the number of crashes with unique stack traces to 236 for RandF, 145 for ProbF, and 45 for Tribble. In `Rhino`, RandF+RSFuzz detected three different exception types (e.g., java.lang.NullPointerException), whereas the baseline fuzzers failed to detected one of them. In terms of crashes with distinct stack traces in `Rhino`, RandF+RSFuzz and ProbF+RSFuzz found 3× and 2.5× more crashes than RandF and ProbF, respectively; Tribble +RSFuzz also detected four more crashes than Tribble. For the JSON benchmarks (Genson, Gson, JsonToJava), the impact of RSFuzz was even more pronounced; RandF+RSFuzz found 175 total crashes with non-overlapping stack traces—98 more than RandF—while ProbF+RSFuzz identified 28 more than ProbF, and Tribble+RSFuzz detected 16 crashes, eight more than Tribble. Similar improvements were observed for the Markdown benchmarks, where RandF+RSFuzz, ProbF+ RSFuzz, and Tribble+RSFuzz found 13, 11, and five more crashes, respectively, than their baselines. For `Super-CSV`, all six fuzzers found the same two unique crashes.

Regarding exclusively detected crashes, RSFuzz proved considerably more effective than the baseline fuzzers. RSFuzz detected 121 additional crashes with distinct stack traces that RandF failed to find, while missing only two. For example, on the `Genson` benchmark, RandF+RSFuzz generated additional crash-inducing inputs—such as "[[[[-1,[[]]]]]]"—that RandF did not produce. Compared to ProbF,ProbF+RSFuzz uniquely identified 46 crashes that ProbF failed to detect, while missing four crashes detected by ProbF. Similarly, Tribble+RSFuzz discovered 17 additional crashes without missing any detected by Tribble. Our further analysis suggests that the improvement in RSFuzz's crash-finding performance primarily stems from reducing coverage-equivalent non-crashing inputs. This can be attributed to the observation that non-crashing inputs and their coverage-equivalent variants rarely differ in observable failure outcomes, whereas most detected crashes were triggered by coverage-inequivalent inputs rather than coverage-equivalent ones.

We further examined whether the crashes found by RSFuzz remain reproducible in the latest versions of the benchmark programs. To this end, we re-executed the crashing inputs on the latest versions of the benchmarks, excluding JsonToJava, Super-CSV and Txtmark, as they were already up to date. For RandF+RSFuzz, 41 out of 66 crashes (based on distinct stack traces) remained reproducible; for ProbF+RSFuzz and Tribble+RSFuzz, 35 out of 52 and 20 out of 28 crashes, respectively, remained reproducible. We manually confirmed that several of these reproducible crashes correspond to previously reported bugs and, in some cases, have already been fixed.

Table 5. The average number of covered lines achieved by our approaches and three baselines on 12 benchmarks. *Cover*: The number of covered lines. *#Inputs*: The number of generated inputs(Values are shown in millions (M)), Δ%: The percentage change in the number of lines covered by each variant relative to RSFuzz.

Program	LoC	RandF		RandF+RSFuzz			ProbF		ProbF+RSFuzz			Tribble		Tribble+RSFuzz		
		Cover	*# Inputs*	*Cover*	Δ%	*# Inputs*	*Cover*	*# Inputs*	*Cover*	Δ%	*# Inputs*	*Cover*	*# Inputs*	*Cover*	deltaratio	*# Inputs*
Jsish	77,375	6,980.2	1.3M	**7,263.4**	4.1%	0.7M	6,075.4	2.2M	**6,260.8**	3.0%	1.7M	7,413.2	10.3M	**7,472.8**	0.8%	2.6M
Rhino	42,048	6,952.0	0.3M	**7,748.2**	11.4%	0.2M	5,327.8	0.3M	**5,805.2**	9.0%	0.2M	7,822.2	0.8M	**8,058.6**	3.0%	0.4M
QuickJS	40,393	11,240.6	1.5M	**12,143.4**	8.0%	0.8M	9,109.4	3.7M	**9,349.0**	2.6%	2.8M	11,992.2	9.4M	**12,510.4**	4.3%	5.1M
JerryScript	33,463	11,171.8	1.3M	**11,686.4**	4.6%	0.7M	8,031.8	2.3M	**8,598.8**	7.1%	1.7M	11,254.6	8.6M	**11,590.6**	3.0%	2.8M
Genson	4,889	813.6	1.5M	**822.6**	1.1%	1.2M	736.0	1.6M	**758.4**	3.0%	1.2M	774.0	3.7M	**791.0**	2.2%	1.7M
Gson	4,488	1,056.6	2.4M	**1,080.4**	2.2%	1.9M	941.8	2.5M	**1,004.6**	6.7%	1.7M	971.6	6.2M	**1046.8**	7.7%	2.9M
JsonToJava	3,009	528.2	3.7M	**543.8**	3.0%	2.6M	497.8	2.8M	**508.8**	2.2%	2.3M	482.0	5.2M	**521.2**	8.1%	3.9M
Argo	1,516	797.4	4.3M	**803.8**	0.8%	3.2M	709.2	4.5M	**751.0**	5.9%	2.8M	772.0	5.2M	**783.2**	1.5%	5.9M
Jackson-CSV	3,191	728.0	50.0M	**728.0**	0.0%	28.9M	728.0	46.5M	**728.0**	0.0%	28.1M	718.0	146.1M	**722.8**	0.7%	89.1M
Super-CSV	1,685	273.0	45.7M	**273.0**	0.0%	27.1M	273.0	37.3M	**273.0**	0.0%	27.2M	237.0	120.7M	**272.0**	14.8%	69.2M
CommonMark	2,187	1,569.2	2.4M	**1,596.8**	1.8%	2.0M	1,558.4	13.0M	**1,599.2**	2.6%	10.0M	1,583.4	38.4M	**1,595.2**	0.8%	16.6M
Txtmark	1,925	1,257.8	2.3M	**1,291.0**	2.6%	1.9M	1,251.6	12.6M	**1,280.8**	2.3%	6.7M	1,264.4	37.1M	**1,279.6**	1.2%	16.3M
Total	207,230	43,368.4	116.7M	**45,980.8**	6.0%	71.2M	35,240.2	129.3M	**36,917.6**	4.8%	86.5M	45,284.6	394.9M	**46,644.2**	3.0%	216.4M

Furthermore, we reported these reproducible crashes to the corresponding developers and have received confirmation that one bug found in the `Rhino` program has been fixed.

Case study. We conducted a detailed case study to investigate how the recurrent sequences produced by RSFuzz could enhance crash-finding capabilities within our benchmark suite. Specifically, we examined crashes that were exclusively detected when integrating RSFuzz. For instance, in the `Gson` benchmark, ProbF+RSFuzz successfully generated an input that triggered an IllegalArgumentException, whereas ProbF alone failed: {"":[[],[30e400]]}. To produce this input, a sequence of 12 specific production rules from the JavaScript grammar had to be applied in the order:

`Start` → `Object` → `Members` → `Pair` → `Value` → `Array` → `Elements` → ... → `{"":[[],[30e400]]}`

Since ProbF selects production rules stochastically, the probability of generating this exact sequence is approximately 1 in 4,000,000. In contrast, ProbF+RSFuzz successfully produced this crash-inducing input using five generated recurrent sequences:

`{Start → Array → "[]"}, {Start → Object → "{}"}, {Start → Array → Elements → Value → null},`
`{Start → Array → Elements → Value → "true"}, {Start → Array → Elements → Value → "false"}`

By incorporating these sequences, ProbF+RSFuzz increased the probability of detecting the exception by about 160,000 times compared to ProbF, enabling successful crash detection.

4.2.2 Line Coverage. Table 5 shows that integrating RSFuzz into each baseline fuzzer generally increased line coverage across all benchmarks compared to the corresponding baseline without RSFuzz. Specifically, out of 36 benchmark–fuzzer combinations (12 benchmarks × 3 baseline fuzzers), RSFuzz improved line coverage in 32 cases and maintained the same coverage in the remaining four. We utilized `gcov` [41], a popular tool, to measure the line coverage for the C benchmark program (`Jsish`, `QuickJS`, and `JerryScript`). For the JAVA benchmark programs—`Rhino`, as well as the JSON, Markdown, and CSV benchmarks—we used `jacoco` [40], another widely-used tool. On average across all 12 programs, the random fuzzer with RSFuzz (RandF+RSFuzz) achieved 6.0% higher line coverage than the random fuzzer (RandF). Similarly, the probabilistic fuzzer with our approach (ProbF+RSFuzz) achieved 4.8% higher average line coverage than the probabilistic fuzzer (ProbF), and Tribble+RSFuzz achieved 3.0% higher coverage compared to Tribble.

The improvements on the `Rhino` and `Gson` benchmarks are particularly noteworthy. On `Rhino`, RandF+RSFuzz covered 7,748.2 lines on average, compared to 6,952.0 lines covered by RandF within the same time budget. For `Gson`, ProbF+RSFuzz achieved 6.7% higher line coverage than ProbF, while Tribble+RSFuzz achieved 7.7% higher coverage than Tribble. For `JerryScript`, integrating RSFuzz with RandF, ProbF, and Tribble improved line coverage by 4.6%, 7.1%, and 3.0%, respectively, compared to the respective fuzzer without RSFuzz. For Super-CSV, Tribble+

RSFuzz achieved 14.8% higher coverage than Tribble, whereas the coverage results for RandF and ProbF remained unchanged even when integrated with RSFuzz.

Among the three baselines without RSFuzz, Tribble generally achieves higher line coverage than the other baselines across our benchmark suite (4.4% higher than RandF and 28.5% higher than ProbF, on average). This is because Tribble systematically explores the grammar by guaranteeing coverage of all k-paths, enabling it to achieve high structural diversity. This advantage is more pronounced when testing programs with complex grammars (e.g., JavaScript), where achieving 100% k-path coverage without specific guidance is difficult. On the four JavaScript benchmarks, Tribble achieved line coverage that was 5.9% higher than RandF and 34.8% higher than ProbF.

Table 5 also reports the number of inputs generated by each fuzzer with and without integration of RSFuzz. When combined with RSFuzz, the number of inputs generated decreased by 40% for RandF, 33% for ProbF, and 45% for Tribble. This reduction occurs because RSFuzz allocates half of the total testing budget (12 hours) generating recurrent sequences, whereas the baseline fuzzer spends the entire budget generating inputs. Despite the negative side effect, integrating RSFuzz resulted in noticeably higher line coverage. This finding highlights the effectiveness of RSFuzz, even when accounting for the time spent generating recurrent sequences.

We measured the standard deviations of line coverage for the RSFuzz-integrated variants and compared them with those of their corresponding baseline fuzzers. Overall, the standard deviations for the RSFuzz-integrated variants were comparable to those of the baselines. For example, on the two largest programs (`Jsish` and `Rhino`), the standard deviations for each fuzzer were as follows: (1) RandF: `Jsish` (69.2), `Rhino` (126.3); (2) RandF+RSFuzz: `Jsish` (24.5), `Rhino` (127.4); (3) ProbF: `Jsish` (10.7), `Rhino` (138.7); (4) ProbF+RSFuzz: `Jsish` (39.5), `Rhino` (106.4); (5) Tribble: `Jsish` (8.6), `Rhino` (65.3); (6) Tribble+RSFuzz: `Jsish` (13.6), `Rhino` (93.2). Furthermore, the Mann-Whitney U test yielded p-values below 0.05 for all 12 benchmarks when comparing the RSFuzz-integrated variants with their respective baselines, indicating statistically significant differences.

Case study. To better understand the mechanisms underlying the observed coverage improvements, we further analyzed whether certain recurrent sequences are consistently shared across benchmarks with the same input format. We identified a recurrent sequence that was consistently generated by all RSFuzz-integrated baseline fuzzers—RandF+RSFuzz, ProbF+RSFuzz, and Tribble +RSFuzz—across the four JavaScript benchmarks (12 cases in total). This recurrent sequence is as:

```
program → sourceElements → sourceElement → statement → expressionStatement
→ expressionSequence → singleExpression → Identifier → IdentifierStart
```

This sequence prevents the derivation of identifier-based expressions in which only the identifier name varies, as such derivations tend to produce coverage-equivalent executions. To evaluate the impact of this sequence on line coverage, we excluded it from the set of recurrent sequences generated by RSFuzz and measured the performance of the RSFuzz-integrated fuzzers. The results show that average line coverage across the four JavaScript benchmarks decreased by 1.3%, 1.2%, and 0.9% for RandF+RSFuzz, ProbF+RSFuzz, and Tribble+RSFuzz, indicating its practical importance.

4.3 Ratio of Coverage-equivalent Inputs

For each program in Table 3, we compared the percentage of coverage-equivalent inputs generated by each baseline fuzzer, with and without RSFuzz, over 100,000 inputs. Figure 3 shows the average percentage of coverage-equivalent inputs among the total inputs generated by six different fuzzers: RandF, ProbF, Tribble, and their RSFuzz-integrated variants. Overall, RSFuzz achieved an average reduction of 23.3%, 28.7% and 14.9% in the ratio of coverage-equivalent inputs generated by RandF, ProbF, and Tribble, respectively, across the 12 benchmarks. Specifically, as shown in Figure 3, on

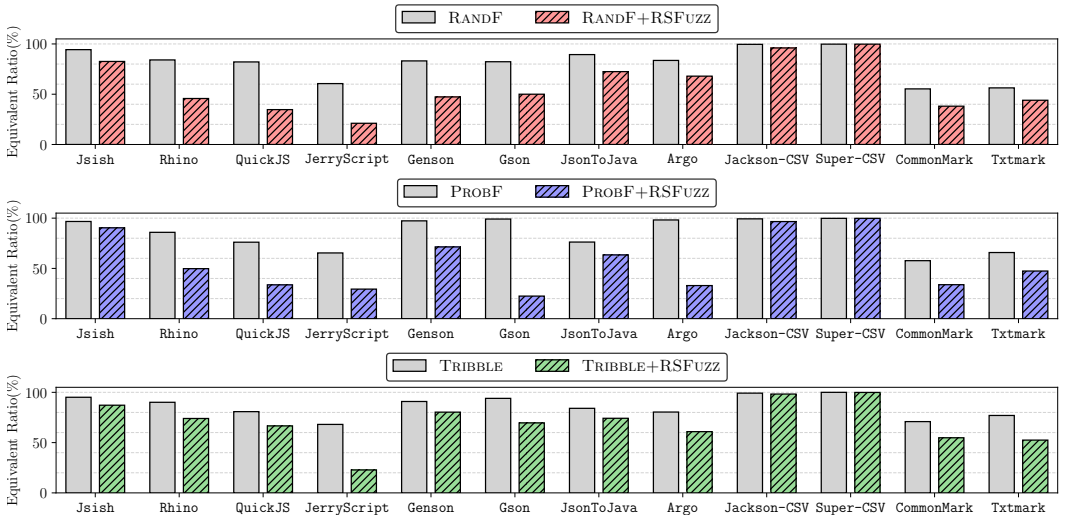


Fig. 3. The percentage of coverage-equivalent inputs generated by three baseline fuzzers with/without RSFuzz on 12 benchmarks.

the Rhino benchmark, the ratio dropped from 84.1% to 45.7% for RandF, from 85.9% to 49.8% for ProbF and from 90.1% to 74.0% for Tribble when integrated with RSFuzz.

This improvement stems from RSFuzz's iterative strategy of promoting inputs that cover new sets of code lines. Across the 12 benchmarks, the baseline fuzzers—RandF, ProbF, and Tribble—covered an average of 17,621, 14,721 and 14,625 unique line sets, respectively. With RSFuzz, these numbers increased by 128.3%, 186.4% and 102.2%, reaching 40,226, 42,157 and 29,572 unique line sets. An exception was observed for the CSV benchmarks, where the increase in covered line sets did not translate into a substantial reduction in coverage-equivalent inputs. For example, on Jackson-CSV, the number of unique line sets increased from 428 to 1,960 for RandF, from 2,059 to 3,379 for ProbF, and from 800 to 1,680 for Tribble, respectively, while the reduction in coverage-equivalent inputs was less than 1%. However, on 10 out 12 benchmarks, this increase in input diversity directly enhanced fuzzing performance, as shown in Section 4.2, leading to higher line coverage and improved crash-finding capability.

4.4 Efficacy of Each Step

To evaluate the effectiveness of the Select and Capture steps in RSFuzz, we compared it against its naive variations. More specifically, RSFuzz without the Select step refers to a variant that does not select the most promising groups but instead uses all groups for capturing recurrent sequences. RSFuzz without the Capture step employs a simplistic capturing method, extracting only the sequence of production rules that appear simultaneously in each selected group as recurrent sequences. Lastly, RSFuzz without both steps is a naive variant that employs this simplistic capturing method across all groups without selecting promising ones. We conducted 24-hour experiments for each naive version of RSFuzz and compared their line coverage.

The results in Table 6 confirm that both the Select and Capture steps in RSFuzz play a crucial role in its performance. The most naive variant, which lacks both steps, reduced the average line coverage of RandF+RSFuzz by 23.0%, ProbF+RSFuzz by 12.5%, and Tribble by 9.6% across 12 benchmarks, performing even worse than the baseline fuzzers alone. This decline occurs because, without both the Select and Capture steps, RSFuzz captures only superficial recurrent sequences

Table 6. The average number of covered lines achieved by three naive variants of RSFuzz on 12 benchmarks. *Cover*: The number of covered lines. Δ%: The percentage change in the number of lines covered by each variant relative to RSFuzz. S and C: Select and Capture steps of RSFuzz, respectively.

Program	RandF							ProbF							Tribble						
	RSFuzz	RSFuzz − (C, S)		RSFuzz − (C)		RSFuzz − (S)		RSFuzz	RSFuzz − (C, S)		RSFuzz − (C)		RSFuzz − (S)		RSFuzz	RSFuzz − (C, S)		RSFuzz − (C)		RSFuzz − (S)	
	Cover	*Cover*	Δ%	*Cover*	Δ%	*Cover*	Δ%	*Cover*	*Cover*	Δ%	*Cover*	Δ%	*Cover*	Δ%	*Cover*	*Cover*	Δ%	*Cover*	Δ%	*Cover*	Δ%
Jsish	7,263	5,941	-18%	7,004	-4%	6,960	-4%	6,261	5,943	-5%	6,123	-2%	6,113	-2%	7,473	7,293	-2%	7,429	-1%	7,402	-1%
Rhino	7,748	5,486	-29%	6,714	-13%	7,298	-6%	5,805	4,424	-24%	5,174	-11%	4,814	-17%	8,059	6,758	-16%	7,686	-5%	7,562	-6%
QuickJS	12,143	9,283	-24%	11,115	-8%	10,950	-10%	9,349	7,917	-15%	9,117	-2%	8,879	-5%	12,510	11,118	-11%	11,374	-9%	11,135	-11%
JerryScript	11,686	7,932	-32%	11,199	-4%	10,773	-8%	8,599	7,363	-14%	8,380	-3%	7,946	-8%	11,591	10,209	-12%	10,791	-7%	10,634	-8%
Genson	823	810	-2%	815	-1%	816	-1%	758	743	-2%	741	-2%	745	-2%	791	771	-3%	779	-2%	775	-2%
Gson	1,080	1,038	-4%	1,060	-2%	1,049	-3%	1,005	997	-1%	1,001	0%	999	-1%	1,047	979	-6%	1003	-4%	1014	-3%
JsonToJava	544	508	-7%	511	-6%	514	-6%	509	486	-5%	498	-2%	487	-4%	521	477	-8%	504	-3%	511	-2%
Argo	804	742	-8%	778	-3%	773	-4%	751	708	-6%	745	-1%	748	0%	783	770	-2%	779	-1%	773	-1%
Jackson-CSV	728	728	0%	728	0%	728	0%	728	725	0%	727	0%	728	0%	723	721	0%	722	0%	722	0%
Super-CSV	273	273	-3%	273	0%	273	0%	273	273	0%	273	0%	273	0%	272	272	0%	272	0%	272	0%
CommonMark	1,597	1,492	-7%	1,496	-6%	1,502	-6%	1,599	1,540	-4%	1,580	-1%	1,573	-2%	1,595	1,571	-2%	1,586	-1%	1,583	-1%
Txtmark	1,291	1,170	-9%	1,194	-8%	1,247	-3%	1,281	1,183	-8%	1,193	-7%	1,204	-6%	1,280	1,240	-3%	1,260	-2%	1,262	-1%
Total	45,981	35,397	-23.0%	42,887	-6.7%	42,884	-6.7%	36,918	32,303	-12.5%	35,552	-3.7%	34,509	-6.5%	46,644	42,180	-9.6%	44,186	-5.3%	43,646	-6.4%

from all groups, resulting in excessive and ineffective recurrent sequences. This substantially increases the cost of input generation, limiting the fuzzer's ability to produce sufficient inputs.

The variant, RSFuzz without the Capture step, reduced the average coverage of RandF+RSFuzz, ProbF+RSFuzz, and Tribble+RSFuzz by 6.7%, 3.7%, and 5.3%, respectively. The most significant decrease was observed in `Rhino` for RandF+RSFuzz and ProbF+RSFuzz, where the coverage dropped by 13% and 11%, respectively, while for Tribble+RSFuzz, the largest decrease (9%) was observed on `QuickJS`. Similarly, in another variant—RSFuzz without the Select step—the average coverage of RandF+RSFuzz, ProbF+RSFuzz, and Tribble+RSFuzz decreased by 6.7%, 6.5%, and 6.4%, respectively. The largest declines were observed on `QuickJS` for RandF+RSFuzz (10%) and Tribble+RSFuzz (11%), and on `Rhino` for ProbF+RSFuzz (17%). We also observed that the relative importance of the Select and Capture steps varied across benchmarks and fuzzers. For instance, in `JerryScript` with ProbF+RSFuzz, removing the Capture step decreased average coverage by only 3%, whereas removing the Select step led to a 8% drop. In contrast, in `Rhino` with RandF+RSFuzz, the Capture step had a greater impact than the Select step. These findings demonstrate that both the Select and Capture steps are essential for the effectiveness of RSFuzz.

We further conducted an ablation study to assess the necessity of capturing redundant rule sequences from *ReG*. Recall that *ReG* denotes groups that previously captured recurrent sequences but failed to prevent coverage-equivalent input generation with them. To demonstrate the role of *ReG*, we modified RSFuzz to capture redundant rules only from *NewG* and integrated this variant into all three baseline fuzzers. Under this configuration, RandF, ProbF and Tribble achieved 2.9%, 1.2%, and 1.4% lower line coverage, respectively, compared to their integration with the original RSFuzz across the 12 benchmark programs. These results indicate that *ReG* also plays a crucial role in ensuring the effectiveness of RSFuzz.

4.5 Reusability

The recurrent sequences generated by RSFuzz for a specific program can be reused for others with the same input format. To validate this, we experimented on the two largest programs for two format: JavaScript and JSON. For JavaScript, we first extracted recurrent sequences from `Jsish` and applied them to `Rhino`, measuring its line coverage over 12 hours. We then reversed the process, using `Rhino`'s recurrent sequences to test `Jsish` within the same time budget. The same procedure was applied to the JSON format, using `Genson` and `Gson`, which share the same input grammar.

Figure 4 presents the average number of lines covered by two variants of RSFuzz for each input format. Specifically, for the JavaScript input format, $\text{RS}_{\texttt{Jsish}}$ represents a random fuzzer using recurrent sequences captured from `Jsish`, while $\text{RS}_{\texttt{Rhino}}$ refers to a random fuzzer using redundant seqeunces captured from `Rhino`. Similarly, $\text{RS}_{\texttt{Genson}}$ and $\text{RS}_{\texttt{Gson}}$ denote the probabilistic fuzzer

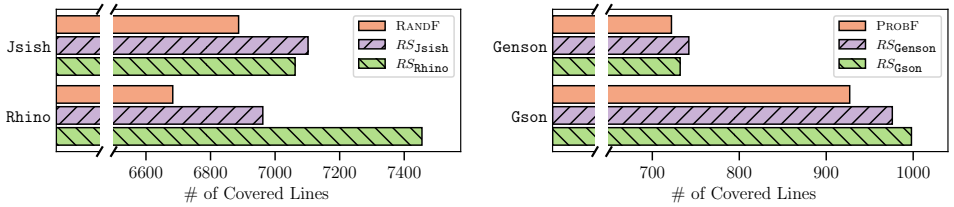


Fig. 4. Avg. coverage of RandF and ProbF when using recurrent sequences optimized for different benchmarks.

utilizing recurrent sequences obtained from Genson and Gson, respectively. For a clear comparison, this figure also includes the line coverage achieved by the baseline fuzzers (RandF and ProbF).

The results in Figure 4 demonstrate that recurrent sequences generated for one program can be effectively reused for testing another program with the same input format. When testing Rhino with RS_{Jsish}, line coverage increased by approximately 4.2% compared to using RandF alone. Similarly, when testing Jsish, RS_{Rhino} covered an average of 200 additional lines compared to RandF. For the JSON benchmarks, testing Gson with RS_{Genson} led to a 5.2% increase in coverage compared to ProbF, while RS_{Gson} achieved a 1.4% increase on Genson. A key reason why recurrent sequences generated for one program can be reused for another is that programs sharing the same input grammar often exhibit similar "coverage-equivalent" inputs. We observed that the recurrent sequences generated for Jsish closely resembled those of Rhino. Unquestionably, RSFuzz achieves the best results when it generates recurrent sequences from a specific program and uses them exclusively for testing that program. When testing Rhino, RS_{Rhino} covered on average 450 more lines than RS_{Jsish}. However, if the goal is to reduce the overhead of generating recurrent sequences, reusing sequences from one program to test others with the same input format is a practical and effective alternative.

4.6 Threats to Validity

(1) Benchmarks: For our evaluation, we collected 12 programs that use four input formats: JavaScript, JSON, CSV, and Markdown. These programs were frequently used in prior works on fuzzing [10, 11, 18–21, 27, 31, 35, 37, 43, 44]. However, these 12 programs might not be representative. (2) Generality: To demonstrate the generality of our approach, we integrated RSFuzz with a random [21], a probabilistic [37], and a grammar-coverage based fuzzer [18]. Experimental evidence shows that RSFuzz successfully enhanced the performance of these fuzzers. However, these outcomes may not necessarily hold when RSFuzz is applied to more sophisticated grammar-based fuzzers [2, 38, 47]. (3) Other grammars: RSFuzz is designed to accept any CFG as input without restrictions, and we expect it to be applicable to other CFG-based grammars (e.g., XML, CSS) and even to languages approximated by CFGs. However, its effectiveness in such cases cannot be guaranteed.

5 Related Work

5.1 Probabilistic Approach

Several techniques have focused on learning grammars with probabilistic production rules to enhance performance [10, 26, 28, 37, 42]. EvoGFuzz [10] introduced an evolutionary algorithm that iteratively updates the probabilities of a probabilistic context-free grammar (PCFG) to effectively find many defects. Similarly, Soremekun et al. [37] presented a technique that learns the probabilities of PCFG from a given set of inputs and inverted the learned probabilities to increase method coverage and produce many failure-inducing inputs. Kifetew et al. [26] proposed a novel approach that learns a probabilistic grammar from a corpus and evolves inputs based on this grammar with the evolutionary guidance of genetic programming, resulting in high branch coverage. Skyfire [42] learns a probabilistic context-sensitive grammar (PCSG) from a large corpus of inputs to generate

syntactically and semantically valid seeds, enabling mutation fuzzers to achieve greater coverage. Falcon [46] aims to efficiently discover hidden bugs in Satisfiability Modulo Theories (SMT) solvers, such as CVC4 [3] and Z3 [9], using a PCFG model for the SMT-LIB2 language. Our work, RSFuzz, can be integrated with these probabilistic techniques to further enhance grammar-based fuzzing.

5.2 Grammar-Aware Approach

Another line of work [2, 22, 36, 42, 43] integrates grammar awareness into mutation-based fuzzing. LangFuzz [22] recombines fragments of real-world inputs to implicitly preserve grammatical structure. TypeFuzz [36] uses the SMT-LIB type system as grammar-like constraints to ensure well-formed formulas. Superion [43] mutates abstract syntax trees (ASTs) to preserve syntactic validity during coverage-guided fuzzing. Nautilus [2] extends this line of work by generating an initial seed corpus from a probabilistic context-free grammar (PCFG) and then applying grammar-aware, feedback-guided mutations to explore programs more effectively. Unlike these techniques, which focus on how to mutate inputs effectively while preserving their grammars, our approach addresses a different problem: determining which production rules of a grammar should not be derived in a particular order, thereby reducing coverage-equivalent inputs.

5.3 Grammar Generation Approach

Our work is orthogonal to approaches that generate input grammars [4, 5, 14–16, 23, 25, 33, 45]. Learn&Fuzz [14] uses recurrent neural networks to automatically generate input grammars for PDF objects from sample inputs, while REINAM [45] synthesizes probabilistic context-free grammars via reinforcement learning. Mimid [15] mines more precise and readable input grammars by tracking dynamic control flow during program executions. CLIFuzzer [16] extracts grammars of command-line options from `getopt()` calls. Bettscheider et al.[5] propose symbolic parsing to derive grammars from parser code without sample inputs. We anticipate that grammar-based fuzzers can be further improved if RSFuzz uses grammars produced by these works.

6 Conclusion

We introudce RSFuzz a novel approach designed to minimize the generation of coverage-equivalent inputs during grammar-based fuzzing. Our approach employs a specialized algorithm that repeatedly groups coverage-equivalent inputs, identifies key groups responsible for generating them, and extracts recurrent sequences of production rules based on accumulated data while interacting with a grammar-based fuzzer. Experimental results demonstrate that integrating RSFuzz with random, probabilistic, and grammar-coverage based fuzzers leads to noticeable improvements in line coverage and crash-finding while reducing coverage-equivalent input generation.

Data Availability

Our tool (RSFuzz) is publicly available at the following URL: https://github.com/skkusal/rsfuzz

Acknowledgments

This work was partly supported by the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.RS-2024-00438686, 30%, Development of software reliability improvement technology through identification of abnormal open sources and automatic application of DevSecOps) and ICT Creative Consilience Program through the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (IITP-2026-RS-2020-II201819, 5%). This work was also supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT)(No.2021R1A5A1021944(10%), RS-2026-25470524(20%), RS-2026-25497428(35%)).

References

[1] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. 1999. OPTICS: ordering points to identify the clustering structure. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data* (Philadelphia, Pennsylvania, USA) *(SIGMOD '99)*. Association for Computing Machinery, New York, NY, USA, 49–60. doi:10.1145/304182.304187

[2] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars.. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS'19)*. doi:10.14722/ndss.2019.23412

[3] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. cvc4. In *Computer Aided Verification: 23rd International Conference (CAV 2011)*. 171–177. doi:10.1007/978-3-642-22110-1

[4] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. *ACM SIGPLAN Notices* 52, 6 (2017), 95–110. doi:10.1145/3062341.3062349

[5] Leon Bettscheider and Andreas Zeller. 2024. Look Ma, No Input Samples! Mining Input Grammars from Code with Symbolic Parsing. Association for Computing Machinery (ACM), 522–526. doi:10.1145/3663529.3663790

[6] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. 2329–2344.

[7] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2019. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering* (2019), 489–506. doi:10.1145/2976749.2978428

[8] Yizong Cheng. 1995. Mean shift, mode seeking, and clustering. *IEEE transactions on pattern analysis and machine intelligence* (1995), 790–799. doi:10.1109/34.400568

[9] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver *(TACAS'08/ETAPS'08)*. 337–340.

[10] Martin Eberlein, Yannic Noller, Thomas Vogel, and Lars Grunske. 2020. Evolutionary grammar-based fuzzing. In *International Symposium on Search Based Software Engineering*. 105–120.

[11] Jueon Eom, Seyeon Jeong, and Taekyoung Kwon. 2024. Fuzzing JavaScript Interpreters with Coverage-Guided Reinforcement Learning for LLM-Based Mutation *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1656–1668. doi:10.1145/3650212.3680389

[12] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining* (Portland, Oregon) *(KDD'96)*. AAAI Press, 226–231.

[13] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*.

[14] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 50–59. doi:10.1109/ASE.2017.8115618

[15] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining Input Grammars from Dynamic Control Flow. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[16] Abhilash Gupta, Rahul Gopinath, and Andreas Zeller. 2022. CLIFuzzer: Mining Grammars for Command-Line Invocations. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[17] Nikolas Havrikov, Alexander Kampmann, and Andreas Zeller. 2022. From Input Coverage to Code Coverage: Systematically Covering Input Structure with k-Paths. (2 2022). doi:10.60882/cispa.24612432.v1

[18] Nikolas Havrikov and Andreas Zeller. 2019. Systematically Covering Input Structure. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 189–199. doi:10.1109/ASE.2019.00027

[19] Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, and Wei Huo. 2021. SoFi: Reflection-Augmented Fuzzing for JavaScript Engines. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2229–2242. doi:10.1145/3460120.3484823

[20] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L Hosking. 2021. Seed selection for successful fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 230–243.

[21] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. Grammarinator: A Grammar-Based Open Source Fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 45–48.

[22] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium (USENIX Security '12)*. 38.

[23] Matthias Höschele and Andreas Zeller. 2016. Mining Input Grammars from Dynamic Taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. Association

for Computing Machinery, New York, NY, USA, 720–725. doi:10.1145/2970276.2970321
[24] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA'14)*. 437–440.
[25] Paul Kalbitzer, Jose Antonio Zamudio Amaya, and Andreas Zeller. 2025. XAVIER: Grammar-Based Testing for XML Injection Attacks. Association for Computing Machinery (ACM), 46–50. doi:10.1145/3713081.3731736
[26] Fitsum Meshesha Kifetew, Roberto Tiella, and Paolo Tonella. 2014. Combining stochastic grammars and genetic programming for coverage testing at the system level. In *Search-Based Software Engineering: 6th International Symposium (SSBSE '14)*. 138–152.
[27] Fitsum Meshesha Kifetew, Roberto Tiella, and Paolo Tonella. 2017. Generating valid grammar-based test inputs by means of genetic programming and annotated grammars. *Empirical Software Engineering* (2017), 928–961.
[28] Fitsum meshesha Kifetew, Roberto Tiella, and Paolo Tonella. 2017. Generating Valid Grammar-Based Test Inputs by Means of Genetic Programming and Annotated Grammars. *Empirical Softw. Engg.* (2017), 928–961.
[29] Xuan-Bach D. Le, Corina Pasareanu, Rohan Padhye, David Lo, Willem Visser, and Koushik Sen. 2019. Saffron: Adaptive Grammar-based Fuzzing for Worst-Case Analysis. 44, 4 (Dec. 2019), 14. doi:10.1145/3364452.3364455
[30] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. 2020. Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer. In *29th USENIX Security Symposium (USENIX Security '20)*. 2613–2630.
[31] Hongyang Lin, Junhu Zhu, Jianshan Peng, and Dixia Zhu. 2019. Deity: Finding Deep Rooted Bugs in JavaScript Engines. In *2019 IEEE 19th International Conference on Communication Technology (ICCT)*. IEEE, 1585–1594.
[32] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *28th USENIX Security Symposium (USENIX Security '19)*. 1949–1966.
[33] Facundo Molina, Marcelo D'Amorim, and Nazareno Aguirre. 2022. Fuzzing Class Specifications. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 1008–1020. doi:10.1145/3510003.3510120
[34] Mitchell Olsthoorn, Arie van Deursen, and Annibale Panichella. 2020. Generating highly-structured input data by combining search-based testing and grammar-based fuzzing. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20)*. 1224–1228.
[35] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19)*. 329–340.
[36] Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2021. Generative Type-Aware Mutation for Testing SMT Solvers. *Proc. ACM Program. Lang.* (2021).
[37] Ezekiel Soremekun, Esteban Pavese, Nikolas Havrikov, Lars Grunske, and Andreas Zeller. 2022. Inputs From Hell: Learning Input Distributions for Grammar-Based Test Generation. *IEEE Transactions on Software Engineering* (2022), 1138–1153.
[38] Dominic Steinhöfel and Andreas Zeller. 2022. Input invariants. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 583–594. doi:10.1145/3540250.3549139
[39] Dominic Steinhöfel and Andreas Zeller. 2024. Language-Based Software Testing. *Commun. ACM* 67, 4 (March 2024), 80–84. doi:10.1145/3631520
[40] JaCoCo Team. 2024. https://github.com/jacoco/jacoco.
[41] Gcov. A tool for measuring coverage. 2021. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.
[42] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (S&P '17)*. 579–594.
[43] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE'19)*. IEEE, 724–735.
[44] Yuli Wang, Xiaoqing Gong, Hao Chen, Jing Li, Baoying Liu, and Shuai Cao. 2020. JSTIFuzz: Type-Inference-based JavaScript Engine Fuzzing. In *2020 International Conference on Networking and Network Applications (NaNA)*. 381–387. doi:10.1109/NaNA51271.2020.00071
[45] Zhengkai Wu, Evan Johnson, Wei Yang, Osbert Bastani, Dawn Song, Jian Peng, and Tao Xie. 2019. REINAM: reinforcement learning for input-grammar inference. In *Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 488–498.
[46] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2021. Fuzzing SMT Solvers via Two-Dimensional Input Space Exploration. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 322–335. doi:10.1145/3460319.3464803
[47] José Antonio Zamudio Amaya, Marius Smytzek, and Andreas Zeller. 2025. FANDANGO: Evolving Language-Based Testing. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA040 (June 2025), 23 pages. doi:10.1145/3728915

Received 2025-09-12; accepted 2026-03-24