Inputs from Hell:

Learning Input Distributions for Grammar-Based Test Generation

Ezekiel Soremekun, Esteban Pavese, Nikolas Havrikov, Lars Grunske, and Andreas Zeller

Abstract—*Grammars* can serve as *producers* for structured test inputs that are syntactically correct by construction. A probabilistic grammar assigns probabilities to individual productions, thus controlling the distribution of input elements. Using the grammars as input parsers, we show how to *learn input distributions from input samples,* allowing to create inputs that are *similar* to the sample; by *inverting* the probabilities, we can create inputs that are *dissimilar* to the sample. This allows for three *test generation strategies*: 1) "Common inputs" – by learning from common inputs, we can create inputs that are *similar* to the sample; this is useful for regression testing. 2) "Uncommon inputs" – learning from common inputs and inverting probabilities yields inputs that are *strongly dissimilar* to the sample; this is useful for completing a test suite with "inputs from hell" that test uncommon features, yet are syntactically valid. 3) "Failure-inducing inputs" – learning from inputs that caused failures in the past gives us inputs that share similar features and thus also have a *high chance of triggering bugs*; this is useful for testing the completeness of fixes. Our evaluation on three common input formats (JSON, JavaScript, CSS) shows the effectiveness of these approaches. Results show that "common inputs" reproduced 96% of the methods induced by the samples. In contrast, for almost all subjects (95%), the "uncommon inputs" covered significantly different methods from the samples. Learning from failure-inducing samples reproduced all exceptions (100%) triggered by the failure-inducing samples and discovered new exceptions not found in any of the samples learned from.

Index Terms—test case generation, probabilistic grammars, input samples

1 Introduction

During the process of software testing, software engineers typically attempt to satisfy three goals:

1) First, the software should work well on *common* inputs, such that the software delivers its promise on the vast majority of cases that will be seen in typical operation. To cover such behavior, one typically has a set of dedicated tests (manually written or generated).
2) Second, the software should work well on *uncommon* inputs. The rationale for this is that such inputs would exercise code that is less frequently used in production, possibly less tested, and possibly less understood [1].
3) Third, the software should work well on inputs that *previously caused failures*, such that it is clear that previous bugs have been fixed. Again, these would be covered via specific tests.

How can engineers obtain such inputs? In this paper, we introduce a novel test generation method that *learns from a set of sample inputs* to produce additional inputs that are markedly *similar* or *dissimilar* to the sample. By learning from past failure-inducing inputs, we can create inputs with similar features; by learning from common inputs, we can create uncommon inputs with dissimilar features not seen in the sample.

- *E. Soremekun is with the Interdisciplinary Centre for Security, Reliability and Trust (SnT), Luxembourg. This work was conducted while working at CISPA Helmholtz Center for Information Security, Saarbrücken, Germany. E-mail: ezekiel.soremekun@uni.lu*
- *N. Havrikov and A. Zeller are with CISPA Helmholtz Center for Information Security, Saarbrücken, Germany. E-mail: see https://cispa.saarland/people/database/*
- *E. Pavese and L. Grunske are with the Department of Computer Science, Humboldt-Universität zu Berlin, Berlin, Germany. E-mail: {pavesees, grunske}@informatik.hu-berlin.de*

The key ingredient to our approach is a *context-free grammar* that describes the input language to a program. Using such a grammar, we can *parse* existing input samples and *count* how frequently specific elements occur in these samples. Armed with these numbers, we can enrich the grammar to become a *probabilistic grammar*, in which production alternatives carry different likelihoods. Since these probabilities come from the samples used for the quantification, such a grammar captures properties of these samples, and producing from such a grammar should produce inputs that are similar to the sample. Furthermore, we can *invert* the learned probabilities in order to obtain a second probabilistic grammar, whose production would produce inputs that are *dissimilar* to the sample. We thus can produce three kinds of inputs, covering the three testing goals listed above:

1) **"Common inputs".** By learning from *common* samples, we obtain a "common" probability distribution, which allows us to produce more "common" inputs. This is useful for regression testing.
2) **"Uncommon inputs".** Learning from common samples, the resulting *inverted* grammar describes in turn the distribution of legal, but *uncommon* inputs. This is useful for completing test suites by testing uncommon features.
3) **"Failure-inducing inputs".** By learning from samples that caused failures in the past, we can produce similar inputs that test the *surroundings* of the original inputs. This is useful for testing the completeness of fixes.

Both the "uncommon inputs" and "failure-inducing inputs" strategies have high chances of triggering failures. Since they combine features rarely seen or having caused issues in the past, we gave them the nickname "inputs from hell". As an example, consider the following JavaScript input generated by focusing on uncommon features:

```
var { a: {} = 'b' } = {};
```

This snippet is valid JavaScript code, but causes the Mozilla Rhino 1.7.7.2 JavaScript engine to crash during interpretation.[1] This input makes use of so-called *destructuring assignments*: In JavaScript, one can have several variables on the left hand side of an assignment or initialization. In such a case, each gets assigned a part of the structure on the right hand side, as in

```
var [one, two, three] = [1, 2, 3];
```

where the variable `one` is assigned a value of `1`, `two` a value of `2`, and so on. Such destructuring assignments, although useful in some contexts, are rarely found in JavaScript programs and tests. It is thus precisely the aim of our approach to generate such uncommon "inputs from hell".

This article makes the following contributions:

1) We use context-free grammars to determine production probabilities from a given set of input samples.
2) We use mined probabilities to produce inputs that are *similar to a set of given samples*. This is useful for thoroughly testing commonly used features (regression testing), or to test the surroundings of previously failure-inducing inputs. Our approach thus leverages probabilistic grammars for both mining and test case generation. In our evaluation using the JSON, CSS and JavaScript formats, we show that our approach repeatedly covers the same code as the original sample inputs; learning from failure-inducing samples, we produce the same exceptions as the samples as well as new exceptions.
3) We use mined probabilities to produce inputs that are *markedly dissimilar* to a set of given samples, yet still valid according to the grammar. This is useful for robustness testing, as well as for exploring program behavior not triggered by the sample inputs. We are not aware of any other technique that achieves this objective. In our evaluation using the same subjects, we show that our approach is successful in repeatedly covering code not covered in the original samples.

The remainder of this paper is organized as follows. After giving a motivational example in Section 2, we detail our approach in Section 3. Section 4 evaluates our three strategies ("common inputs", "uncommon inputs", and "failure-inducing inputs") on various subjects. After discussing related work (Section 6), Section 7 concludes and presents future work.

2 Inputs from Hell in a Nutshell

To demonstrate how we produce both common and uncommon inputs, let us illustrate our approach using a simple example grammar. Let us assume we have a program P that processes *arithmetic expressions*; its inputs follow the standard syntax given by the grammar G below.

Expr → *Term* | *Expr* **"+"** *Term* | *Expr* **"−"** *Term*;
Term → *Factor* | *Term* **"*"** *Factor*
| *Term* **"/"** *Factor*;
Factor → *Int* | **"+"** *Factor*
| **"−"** *Factor* | **"("** *Expr* **")"**;
Int → *Digit Int* | *Digit*;
Digit → **"0"** | **"1"** | **"2"** | **"3"** | ... | **"9"**;

Let us further assume we have discovered a bug in P: The input $I =$ `1 * (2 + 3)` is not evaluated properly. We have fixed the bug in P, but want to ensure that similar inputs would also be handled in a proper manner.

To obtain inputs that are *similar* to I, we first use the grammar G to *parse* I and determine the *distribution* of the individual choices in productions. This makes G a *probabilistic* grammar G_p in which the productions' choices are tagged with their probabilities. For the input I above, for instance, we obtain the probabilistic rule

Digit → 0% **"0"** | 33.3% **"1"** | 33.3% **"2"**
| 33.3% **"3"** | 0% **"4"** | 0% **"5"**
| 0% **"6"** | 0% **"7"** | 0% **"8"** | 0% **"9"**;

which indicates the distribution of digits in I. Using this rule for production, we would obtain ones, twos, and threes at equal probabilities, but none of the other digits. Figure 2 shows the grammar G_p as extension of G with all probabilities as extracted from the derivation tree of I (Figure 1). In this derivation tree we see, for instance, that the nonterminal *Factor* occurs 4 times in total. 75% of the time it produces integers (*Int*), while in the remaining 25%, it produces a parenthesis expression (**"("** *Expr* **")"**). Expressions using unary operators like **"+"** *Factor* and **"−"** *Factor* do not occur.

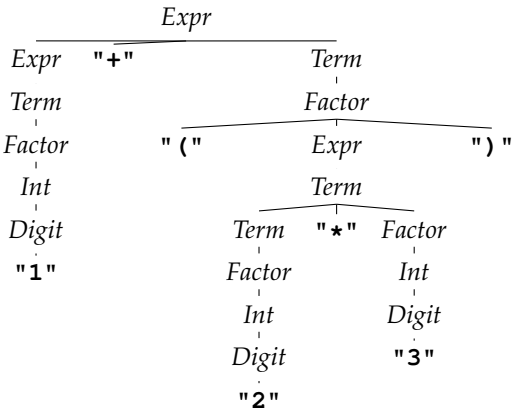


Fig. 1. Derivation tree representing **"1 + (2 * 3)"**

If we use G_p from Figure 2 as a probabilistic production grammar, we obtain inputs according to these probabilities. As listed in Figure 3, these inputs uniquely consist of the digits and operators seen in our sample `1 * (2 + 3)`. All of these inputs are likely to cover the same code in P as the original sample input, yet with different input structures that trigger the same functionality in P in several new ways.

When would one want to replicate the features of sample inputs? In the "common inputs" strategy, one would create test cases that are similar to a set of common inputs; this is helpful for regression testing. In the more interesting "failure-inducing inputs" strategy, one would learn from a set of failure-inducing samples to replicate their features; this is useful for testing the surroundings of past bugs.

If one only has sample inputs that work just fine, one would typically be interested in inputs that are *different* from

1. We have reported this snippet as Rhino issue #385 and it has been fixed by the developers.

Expr → 66.7% *Term* | 33.3% *Expr* `"+"` *Term*
| 0% *Expr* `"-"` *Term*`;`
Term → 75% *Factor* | 25% *Term* `"*"` *Factor*
| 0% *Term* `"/"` *Factor*`;`
Factor → 75% *Int* | 0% `"+"` *Factor*
| 0% `"-"` *Factor* | 25% `"("` *Expr* `")";`
Int → 0% *Digit Int* | 100% *Digit*`;`
Digit → 0% `"0"` | 33.3% `"1"` | 33.3% `"2"`
| 33.3% `"3"` | 0% `"4"` | 0% `"5"`
| 0% `"6"` | 0% `"7"` | 0% `"8"` | 0% `"9";`

Fig. 2. Probabilistic grammar G_p, expanding G

```
(2 * 3)
2 + 2 + 1 * (1) + 2
((3 * 3))
3 * (((3 + 3 + 3) * (2 * 3 + 3))) * (3)
3 * 1 * 3
((3) + 2 + 2 * 1) * (1)
1
((2)) + 3
```

Fig. 3. Inputs generated from G_p in Figure 2

our samples—the "uncommon inputs" strategy. We can easily obtain such inputs by *inverting* the mined probabilities: if a rule previously had a weight of p, we now assign it a weight of $1/p$, normalized across all production alternatives. For our *Digit* rule, this gives the digits not seen so far a weight of $1/0 = \infty$, which is still distributed equally across all seven alternatives, yielding individual probabilities of $1/7 = 14.3\%$. Proportionally, the weights for the digits already seen in I are infinitely small, yielding a probability of effectively zero. The "inverted" rule reads now:

Digit → 14.3% `"0"` | 0% `"1"` | 0% `"2"` | 0% `"3"`
| 14.3% `"4"` | 14.3% `"5"` | 14.3% `"6"`
| 14.3% `"7"` | 14.3% `"8"` | 14.3% `"9";`

Applying this inversion to rules with non-terminal symbols is equally straightforward. The resulting probabilistic grammar $G_{p^{-1}}$ is given in Figure 4.

Expr → 0% *Term* | 0% *Expr* `"+"` *Term*
| 100% *Expr* `"-"` *Term*`;`
Term → 0% *Factor* | 0% *Term* `"*"` *Factor*
| 100% *Term* `"/"` *Factor*`;`
Factor → 0% *Int* | 50% `"+"` *Factor*
| 50% `"-"` *Factor* | 0% `"("` *Expr* `")";`
Int → 100% *Digit Int* | 0% *Digit*`;`
Digit → 14.3% `"0"` | 0% `"1"` | 0% `"2"` | 0% `"3"`
| 14.3% `"4"` | 14.3% `"5"` | 14.3% `"6"`
| 14.3% `"7"` | 14.3% `"8"` | 14.3% `"9";`

Fig. 4. Grammar $G_{p^{-1}}$ inverted from G_p in Figure 2

This inversion can lead to infinite derivations, for example, the production rule in $G_{p^{-1}}$ for generating *Expr* is recursive 100% of the time, expanding only to *Expr* `"-"` *Term*, without chance of hitting the base case. As a result, we take special measures to avoid such infinite productions during input generation (*see Section 3.3*).

If we use $G_{p^{-1}}$ as a production grammar—and avoiding infinite production—we obtain inputs as shown in Figure 5. These inputs now focus on operators like subtraction or division or unary operators not seen in our input samples. Likewise, the newly generated digits cover the complement of those digits previously seen. Yet, all inputs are syntactically valid according to the grammar.

In summary, with common inputs as produced by G_p, we can expect to have a good set of regression tests—or a set replicating the features of failure-inducing inputs when learning from failure-inducing samples. In contrast, uncommon inputs as produced by $G_{p^{-1}}$ would produce features rarely found in samples, and thus cover complementary functionality.

```
+5 / -5 / 7 - +0 / 6 / 6 - 6 / 8 - 5 - 4
-4 / +7 / 5 - 4 / 7 / 4 - 6 / 0 - 5 - 0
+5 / ++4 / 4 - 8 / 8 - 4 / 8 / 7 - 8 - 9
-6 / 9 / 5 / 8 - +7 / -9 / 6 - 4 - 4 - 6
+8 / ++8 / 5 / 4 / 0 - 5 - 4 / 8 - 8 - 8
-9 / -5 / 9 / 4 - -9 / 0 / 5 - 8 / 4 - 6
++7 / 9 / 5 - +8 / +9 / 7 / 7 - 6 - 8 - 4
-+6 / -8 / 9 / 6 - 5 / 0 - 5 - 8 - 0 - 5
```

Fig. 5. Inputs generated from $G_{p^{-1}}$ from Figure 4

3 Approach

In order to explain our approach in detail, we start with introducing basic notions of probabilistic grammars.

3.1 Probabilistic Grammars

The probabilistic grammars that we employ in this paper are based on the well-known context-free grammars (CFGs) [2].

Definition 1 (Context-free grammar)**.** *A context-free grammar is a 4-tuple (V, T, P, S_0), where V is the set of* non-terminal symbols*, T the terminals, $P : V \to (V \cup T)^*$ the set of productions, and $S_0 \in V$ the start symbol.*

In a *non-probabilistic grammar*, rules for a non-terminal symbol S provide n alternatives A_i for expansion:

$$S \to A_1 \mid A_2 \mid \ldots \mid A_n \tag{1}$$

In a *probabilistic context-free grammar* (PCFG), each of the alternatives A_i in Equation (1) is augmented with a probability p_i, where $\sum_{i=1}^{n} p_i = 1$ holds:

$$S \to p_1\, A_1 \mid p_2\, A_2 \mid \ldots \mid p_n\, A_n \tag{2}$$

If we are using these grammars for generation of a sentence of the language described by the grammar, each alternative A_i has a probability of p_i to be selected when expanding S.

By convention, if one or more p_i are not specified in a rule, we assume that their value is the complement probability, distributed equally over all alternatives with these unspecified probabilities. Consider the rule

Letter → 40.0% `"a"` | `"b"` | `"c"`

Here, the probabilities for `"b"` and `"c"` are not specified; we assume that the complement of `"a"`, namely 60%, is equally distributed over them, yielding effectively

Letter $\rightarrow$ 40.0% `"a"` | 30.0% `"b"` | 30.0% `"c"`

Formally, to assign a probability to an unspecified p_i, we use

$$p_i = \frac{1 - \sum\{p_j | p_j \text{ is specified for } A_j\}}{\text{number of alternatives } A_k \text{ with unspecified } p_k} \quad (3)$$

Again, this causes the invariant $\sum_{i=1}^{n} p_i = 1$ to hold. If no p_i is specified for a rule with n alternatives, as in Equation (1), then Equation (3) makes each $p_i = 1/n$, as intended.

3.2 Learning Probabilities

Our aim is to turn a classical context-free grammar G into a probabilistic grammar G_p capturing the probabilities from a set of samples—that is, to determine the necessary p_i values as defined in Equation (2) from these samples. This is achieved by *counting* how frequently individual alternatives occur during parsing in each production context, and then to determine appropriate probabilities.

In language theory, the result of parsing a sample input I using G is a *derivation tree* [3], representing the structure of a sentence according to G. As an example, consider Figure 1, representing the input `"1 + (2 * 3)"` according to the example arithmetic expression grammar in Section 2. In this derivation tree, we can now *count* how frequently a particular alternative A_i was chosen in the grammar G during parsing. In Figure 1, the rule for *Expr* is invoked three times during parsing. This rule expands once (33.3%) into *Expr* `"+"` *Term* (at the root); and twice (66.7%) into *Term* in the subtrees. Likewise, the *Term* symbol expands once (25%) into *Term* `"*"` *Factor* and three times (75%) into *Factor*. Formally, given a set T of derivation trees from a grammar G applied on sample inputs, we determine the probabilities p_i for each alternative A_i of a symbol $S \rightarrow A_1 \mid \ldots \mid A_n$ as

$$p_i = \frac{\text{Expansions of } S \rightarrow A_i \text{ in } T}{\text{Expansions of } S \text{ in } T} \quad (4)$$

If a symbol S does not occur in T, then Equation (4) makes $p_i = 0/0$ for all alternatives A_i; in this case, we treat all p_i for S as *unspecified*, assigning them a value of $p_i = 1/n$ in line with Equation (3). In our example, Equation (4) yields the probabilistic grammar G_p in Figure 2.

3.3 Inverting Probabilities

We turn our attention now to the converse approach; namely producing inputs that *deviate* from the sample inputs that were used to learn the probabilities described above. This "uncommon input" approach promises to be useful if we accept that our samples are not able to cover all the possible system behavior, and if we want to find bugs in behaviors that are either not exercised by our samples, or do so rarely.

The key idea is to *invert* the probability distributions as learned from the samples, such that the input generation focuses on the complement section of the language (w.r.t. the samples and those inputs generated by the probabilistic grammar). If some symbol occurs frequently in the parse trees corresponding to the samples, this approach should generate the symbol less frequently, and vice versa: if the symbol seldom occurs, then the approach should definitely generate it often.

For a moment, let us ignore probabilities and focus on *weights* instead. That is, the absolute (rather than relative) number of occurrences of a symbol in the parse tree of a sample. We start by determining the occurrences of a symbol A during a production S found in a derivation tree T:

$$w_{A,S} = \begin{array}{l}\text{Occurrence count of } A \text{ in the} \\ \text{expansions of symbol } S \text{ in } T\end{array} \quad (5)$$

To obtain *inverted* weights $w'_{A,S}$, a simple way is to make each $w'_{A,S}$ based on the reciprocal value of $w_{A,S}$, that is

$$w'_{A,S} = {w_{A,S}}^{-1} = \frac{1}{w_{A,S}} \quad (6)$$

If the set of samples is small enough, or focuses only on a section of the language of the grammar, it might be the case that some production or symbol never appears in the parsing trees. If this is the case, then the previous equations end up yielding $w_{A,S} = 0$. We can compute ${w_{A,S}}^{-1} = \infty$, assigning the elements not seen an infinite weight. Consequently, all symbols B that were indeed seen before (with $w_{B,S} > 0$) are assigned an infinitesimally small weight, leading to $w'_{B,S} = 0$. The remaining infinite weight is then distributed over all of the originally "unseen" elements with original weight $w_{A,S} = 0$. Recall the arithmetic expression grammar in Section 2; such a situation arises when we consider the rule for the symbol *Digit*: the inverted probabilities for the rule focus exclusively on the complement of the digits seen in the sample.

All that remains in order to obtain actual probabilities is to *normalize* the weights back into a probability measure, ensuring for each rule that its invariant $\sum_{i=1}^{n} p'_i = 1$ holds:

$$p'_i = \frac{w'_i}{\sum_{i=1}^{n} w'_i} \quad (7)$$

3.4 Producing Inputs from a Grammar

Given a probabilistic grammar G_p for some language (irrespective of whether it was obtained by learning from samples, by inverting, or simply written that way in the first place), our next step in the approach is to generate inputs following the specified productions. This generation process is actually very simple, since it reduces to produce instances by traversing the grammar, as if it were a Markov chain. However, this generation runs the serious risk of probabilistically choosing productions that lead to an excessively large parsing tree. Even worse, the risk of generating an *unbounded* tree is very real, as can be seen in the rule for the symbol *Int* in the arithmetic expression grammar in Section 2. The production rule for said symbol triggers, with probability 1.0, a recursion with no base case, and will never terminate.

Our inspiration for constraining the growth of the tree during input generation comes from the PTC2 algorithm [4]. The main idea of this algorithm is to allow the expansion of not-yet-expanded productions, while ensuring that the number of productions does not exceed a certain threshold of performed expansions. This threshold would be set as parameter of the input generation process. Once

this threshold is exceeded, the partially generated instance cannot be truncated, as that would result in an illegal input. Alternatively, we choose to allow further expansion of the necessary non-terminal symbols. However, from this point on, expansions are not chosen probabilistically. Rather, the choice is constrained to those expansions that generate the *shortest* possible expansion tree. This ensures both termination of the generation procedure, as well as trying to keep the input size close to the threshold parameter. This choice, however, does introduce a bias that may constitute a threat to the validity of our experiments that we discuss in Section 4.3.

3.5 Implementation

As a prerequisite for carrying out our approach, we only assume we have the context-free grammar of the language available for which we are interested in generating inputs, and a collection (no matter the size) of inputs that we will assume are *common* inputs. Armed with these elements, we perform the workflow detailed in Figure 6.

The first step of the approach is to obtain a *counting grammar* from the original grammar. This counting grammar is, from the parsing point of view, completely equivalent to the original grammar. However, it is augmented with *actions* during parsing which perform all necessary counting of symbol occurrences parallel to the parsing phase. Finally, it outputs the probabilistic grammar. Note that this first phase requires not only the grammar of the target language, but also the grammar of the *language in which the grammar itself is written*. That is, generating the probabilistic grammar not only requires parsing sample inputs, but also the grammar itself. In the particular case of our implementation, we make use of the well-known parser generator ANTLR [5].

Once the probabilistic grammar is obtained, we derive the probabilistically-inverted grammar as described in this section. Armed with both probabilistically annotated grammars, we can continue with the input generation procedure.

4 Experimental Evaluation

In this section we evaluate our approach by applying the technique in several case studies. In particular, we ask the following research questions:

- **RQ1 ("Common inputs").** Can a learned grammar be used to generate inputs that resemble those that were employed during the grammar training?
- **RQ2 ("Uncommon inputs").** Can a learned grammar be modified so it can generate inputs that, opposed to **RQ1**, are *in contrast* to those employed during the grammar training?
- **RQ3 ("Sensitivity to training set variance").** Is our approach sensitive to variance in the initial samples?
- **RQ4 ("Sensitivity to size of training set").** Is our approach sensitive to the size of the initial samples?
- **RQ5 ("Bugs found").** What kind of crashes (exceptions) do we trigger in **RQ1** and **RQ2**?
- **RQ6 ("Failure-inducing inputs").** Can a learned grammar be used to generate inputs that reproduce failure-inducing behavior?

To answer **RQ1** and **RQ2**, we need to compare inputs in order to decide whether these inputs are "similar" or "contrasting". In the scope of this evaluation, we will use the *method coverage* and *call sequences* as measures of input similarity. We will define these measures later in this section, and we will discuss their usefulness. We address **RQ3** by comparing the method calls and call sequences induced for three randomly selected training sets, each containing five inputs. Likewise, we evaluate **RQ4** by comparing the method calls and call sequences induced for four randomly selected training sets, each containing N sample inputs, where $N \in \{1, 5, 10, 50\}$. We assess **RQ5** by categorizing, inspecting and reporting all exceptions triggered by our test suites in **RQ1** and **RQ2**. Finally, we address **RQ6** by investigating if the "(un)common inputs" strategy can reproduce (or avoid) a failure and explore the surroundings of the buggy behavior.

TABLE 1
Depth and size of derivation trees for "common inputs" (PROB) and "uncommon inputs" (INV)

Grammar	Mode	Depth of derivation tree				Nodes
		min	max	avg.	median	avg.
JSON	PROB	14	2867	96	63	3058
	INV	5	37	23	37	68
JavaScript	PROB	1	79	19	8	400
	INV	1	38	19	1	11,061
CSS	PROB	3	44	41	44	19,380
	INV	9	30	29	30	11,269

4.1 Evaluation Setup

4.1.1 Generated Inputs

Once a probabilistic grammar is learned from the training instances, we generate several inputs that are fed to each subject. Our evaluation involves the generation of three types of test suites:

a) *Probabilistic* - choice between productions is governed by the distribution specified by the learned probabilities in the grammar.
b) *Inverse* - choice is governed by the distribution obtained by the inversion process described in Section 3.3.
c) *Random* - choice between productions is governed by a uniform distribution (see **RQ6**).

Expansion size control is carried out in order to avoid unbounded expansion as described in Section 3.4. Table 1 reports the details of the produced inputs, i.e. the depth and average number of nodes in the derivation trees for the "common inputs" (i.e., probabilistic/PROB) and "uncommon inputs" (i.e., inverse/INV).

4.1.2 Research Protocol

In our evaluation, we generate test suites and measure the frequency of method calls, the frequency of call sequences and the number of failures induced in our subject programs. For each input language, the experimental protocol proceeds as follows:

a) We randomly selected five files from a pool of thousands of sample files crawled from GitHub code repositories, and through our approach produced a probabilistic grammar out of them[2]. Since one of the main use cases of

2. To evaluate **RQ6**, we learned a PCFG from at most five random failure-inducing inputs.

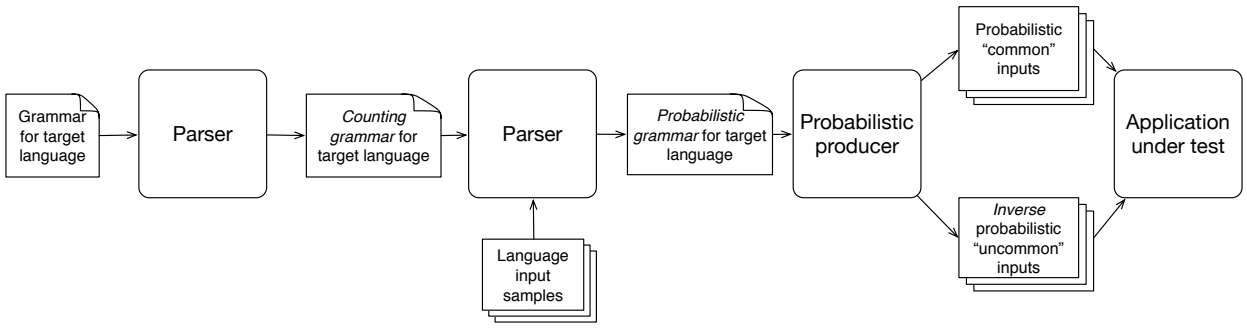


Fig. 6. Workflow for the generation of *"common inputs"* and *"uncommon inputs"*

our tool is to complete a test suite, we perform grammar training with few (i.e. five) initial sample tests.

b) We feed the sampled input files into the subject program and record the triggered failures, the induced call sequences and the frequency of method calls using the HPROF [6] profiler for Java.

c) Using the probabilistic grammar, we generate test suites, each one containing 100 input files. We generate a total of 1000 test suites, in order to control for variance in the input files. Overall, each experiment contains 100,000 input files (100 files x 1,000 runs). We perform this step for both probabilistic and inverse generations. Hence, the total number of inputs generated for each grammar is 200,000 (1,000 suites of 100 inputs each, a set of suites for each experiment).

d) We test each subject program by feeding the input files into the subject program and recording the induced failures, the induced call sequences and the frequency of method calls using HPROF.

All experiments were conducted on a server with 64 cores and 126 GB of RAM; more specifically an Intel Xeon CPU E5-2683 v4 @ 2.10GHz with 64 virtual cores (Intel Hyper-threading), running Debian 9.5 Linux.

4.1.3 Subject Programs

We evaluated our approach by generating inputs and feeding them to a variety of Java applications. All these applications are open source programs using three different input formats, namely JSON, JavaScript and CSS3. Table 2 summarizes the subjects to be analyzed, their input format and the number of methods in each implementation.

The initial, unquantified grammars for the input subjects were adapted from those in the repository of the well-known parser generator ANTLR [5]. We handle grammar ambiguity that may affect learning probabilities by ensuring every input has only one parse tree. Specifically, we adapt the input grammars by (re-)writing lexer modes for the grammars, shortening lexer tokens and re-writing parser rules. Training samples were obtained by scraping GitHub repositories for the required format files. The probabilistic grammars developed from the original ones, as well as the obtained training samples can be found in our replication package.

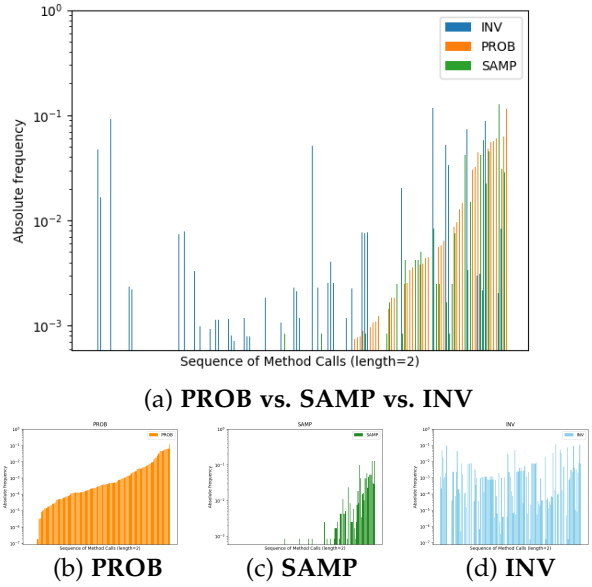
4.1.4 Measuring (Dis)similarity

Questions **RQ1** and **RQ2** refer to a notion of similarity between inputs. Although white-box approaches exist that aim to measure test-case similarity and dissimilarity [7], [8], applying them to complex grammar-based inputs is not straightforward. However, in this paper, since we are dealing with evaluating the behavior of a certain piece of software, it makes sense to aim for a notion of *semantic* similarity. In this sense, two inputs are semantically similar if they incite similar behaviors in the software that processes them. In order to achieve this, we define two measures of input similarity based on structural and non-structural program coverage metrics. The *non-structural measure* of input similarity is the *frequency of method calls* induced in the programs. The *structural measure* is the *frequency of call sequences* induced in the program, a similar measure was used in [9]. Thus, we will say two inputs are similar if they induce similar (distribution of) method call frequencies for the same program. The *frequency of call sequences* refers to the number of times a specific method call sequence is triggered by an input, for a program. For this measure, we say two inputs are similar if they trigger a similar distribution in the frequency with which the method sequences are called, for the same program. These notions allow for a great variance drift if we were to compare only two inputs. Therefore, we perform these comparisons on test suites as a whole to dampen the effect of this variance.

Using these measures, we aim at answering **RQ1** and **RQ2**. **RQ1** will be answered satisfactorily if the (distribution of) call frequencies and sequences induced by the "common

TABLE 2
Subject details

Input Format	Subject	Version	#Methods	LOC
JSON	Argo	5.4	523	8,265
	Genson	1.4	1,182	18,780
	Gson	2.8.5	793	25,172
	JSONJava	20180130	202	3,742
	Jackson	2.9.0	5,378	117,108
	JsonToJava	1880978	294	5,131
	MinimalJson	0.9.5	224	6,350
	Pojo	0.5.1	445	18,492
	json-simple	a8b94b7	63	2,432
	cliftonlabs	3.0.2	183	2,668
	fastjson	1.2.51	2,294	166,761
	json2flat	1.0.3	37	659
	json-flattener	0.6.0	138	1,522
JavaScript	Rhino	1.7.7	4873	100,234
	rhino-sandbox	0.0.10	49	529
CSS3	CSSValidator	1.0.4	7774	120,838
	flute	1.3	368	8,250
	jstyleparser	3.2	2,589	26,287
	cssparser	0.9.27	2,014	18,465
	closure-style	0.9.27	3,029	35,401



(a) **PROB vs. SAMP vs. INV**

(b) **PROB** (c) **SAMP** (d) **INV**

Fig. 7. Frequency analysis of call sequences for json-flattener (length=2)

inputs" strategy is *similar* to the call frequency and sequences obtained when running the software on the *training samples*. Likewise, **RQ2** will be answered positively if the (distribution of) call frequencies and sequences for suites generated with the "uncommon inputs" strategy are markedly *dissimilar*.

4.1.5 Visual test

For each test suite, we compare the frequency distribution of the call sequences and method calls triggered in a program, using grouped and single bar charts. These comparisons are in line with the visual tests described in [10].

For instance, Figure 7 shows the frequency analysis of the call sequences induced in `json-flattener` by our test suites. The grouped bar chart compares the frequency distribution of call sequences for all three test suites, (i.e. (a) PROB vs. SAMP vs. INV) and the single bar chart shows the frequency distribution of call sequences for each test suite (i.e., (b.) PROB, (c) SAMP and (d) INV). Frequency analysis (in (a.)) shows that the (distribution of) call sequences of PROB and SAMP align (see rightmost part of bar chart), and INV often induces a different distribution of call sequences from the initial samples (see leftmost part of bar chart). The single bar chart for a test suite shows the frequency distribution of the call sequences triggered by the test suite. For instance, Figure 7 (b) and (c) show the call sequence distribution triggered by the "common inputs" and initial samples respectively. The comparison of both charts shows that all call sequences covered by the samples, were also frequently covered by the "common inputs".

Likewise, Figure 9 to Figure 11 show the call frequency analysis of the test suites using a grouped bar chat for comparison (i.e. (a) PROB vs. SAMP vs. INV) and a single bar chart to show the call frequency distribution of each test suite (i.e., (b.) PROB, (c) SAMP and (d) INV). The grouped bar chart shows the call frequency for each test suite grouped together by method, with bars for each test suite appearing side by side per method. For instance, analysing Figure 9 (a)

TABLE 3
Call Sequence analysis for "common inputs" (PROB) and "uncommon inputs" (INV) for all subject programs

Length	Call Sequences covered by Sample					Call sequences covered by	
	#	also by PROB		also by INV		PROB	INV
2	1210	1157	(96%)	937	(74%)	6348	5196
3	1152	1099	(95%)	782	(62%)	7946	5930
4	849	803	(90%)	479	(47%)	9236	5825
Total	3211	3059	(94%)	2198	(61%)	23 530	16 951

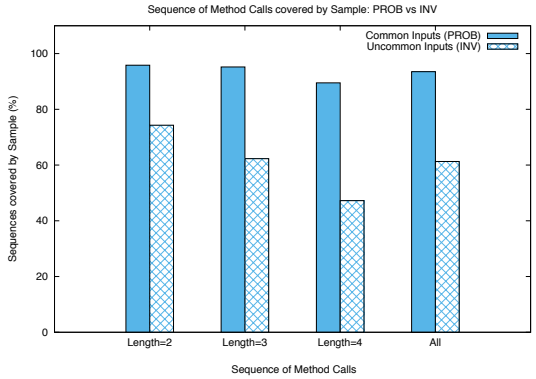


Fig. 8. Call sequences covered by Sample for "common inputs" (PROB) and "uncommon inputs" (INV)

shows that the call frequencies of PROB and SAMP align (see rightmost part of bar chart), and INV often induces a different call frequency for most methods (see leftmost part of bar chart). Moreover, the single bar chart for a test suite shows the call frequency distribution of the test suite. For instance, Figure 9 (b) and (c) show the call frequency distribution of the "common inputs" and initial samples respectively, their comparison shows that all methods covered by the samples, were also frequently covered by the "common inputs".

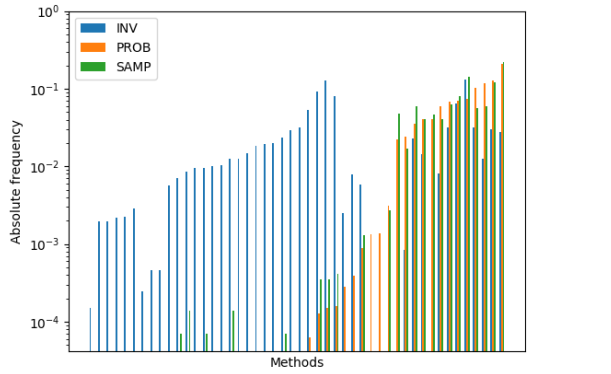
4.1.6 Collecting failure-inducing inputs

For each input file in our Github corpus, we fed it to every subject program of the input language and observe if the subject program crashes, i.e. the output status code is non-zero after execution. Then, we collect such inputs as failure-inducing inputs for the subject program and parse the standard output for the raised exception. In total, we fed 10,853 files to the subject programs, 1,000 each for CSS and JavaScript, and 8853 for JSON. Exceptions were triggered for two input languages, namely JavaScript and JSON, no exception was triggered for CSS. In total, we collected 15 exceptions in seven subject programs (*see Table 10*).

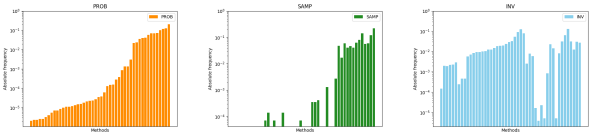
4.2 Experimental Results

In Figure 9 to Figure 11, we show a representative selection of our results[3]. For each subject, we constructed a chart that represents the absolute call frequency of each method in the subject. The horizontal axis (which is otherwise unlabelled)

3. The full range of charts is omitted for space reasons. However, all charts, as well as the raw data, are available as part of the artifact package. Moreover, the charts shown here have been selected so that they are representative of the whole set; that is, the omitted charts do not deviate significantly.

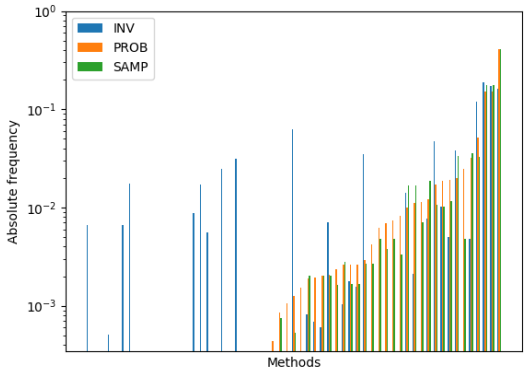


(a) **PROB vs. SAMP vs. INV**

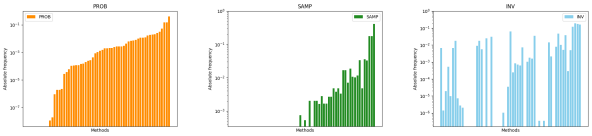


(b) **PROB** (c) **SAMP** (d) **INV**

Fig. 9. Call frequency analysis for json-simple-cliftonlabs



(a) **PROB vs. SAMP vs. INV**



(b) **PROB** (c) **SAMP** (d) **INV**

Fig. 10. Call frequency analysis for JSONJava

represents the set of methods in the subject, ordered by the *frequency of calls* in the experiment on *probabilistic inputs*.

***RQ1** ("Common inputs"): Can a learned grammar be used to generate inputs that resemble those that were employed during the grammar training?*

To answer **RQ1**, we compare the methods covered by the sample inputs and the "common inputs" strategy (*Table 4 and Figure 9 to Figure 11*). We also examine the call sequences covered by the sample inputs and the "common inputs", for consecutive call sequences of length two, three and four (*Table 3 and Figure 8*). In particular, we investigate if the "common inputs" strategy covered at least the same methods or the same call sequences as the initial sample inputs.

Do the "common inputs" trigger similar non-structural program behavior (i.e., method calls) as the initial samples? For all subjects, the "common inputs" strategy covered almost all (96%) of the methods covered by the sample (see Table 4). This result shows that the "common inputs" strategy learned the input properties in the samples and reproduced the same (non-structural) program behavior as the initial samples. Besides, this strategy also covered other methods that were not covered by the samples.

The "common inputs" strategy triggered almost all methods (96%) called by the initial sample inputs.

Do the "common inputs" also trigger similar structural program behavior (i.e., sequences of method calls)? In our evaluation, the "common inputs" strategy covered most of the call sequences that were covered by the initial samples. For instance, Figure 7 shows that the call sequences covered by the samples were also frequently covered by the "common inputs", for `json-flattener`. Overall, the "common inputs" strategy covered 94% of the method call sequences induced by the sample (*see Table 3 and Figure 8*). For all call sequences, the "common inputs" strategy also covered 90% to 96% of the method call sequences covered by the samples. This result shows that the 'common inputs" strategy triggers the same structural program behavior as the initial samples.

The "common inputs" strategy triggered most call sequences (94%) covered by the initial sample inputs.

Additionally, we compare the statistical distributions resulting from our strategies. We need to be able to see a pattern in frequency calls such that the frequency curves for the *sample* runs and the *probabilistic* runs match as described in the visual test (see Section 4.1.5). Figure 9 to Figure 11 show that this match does hold for all subjects.

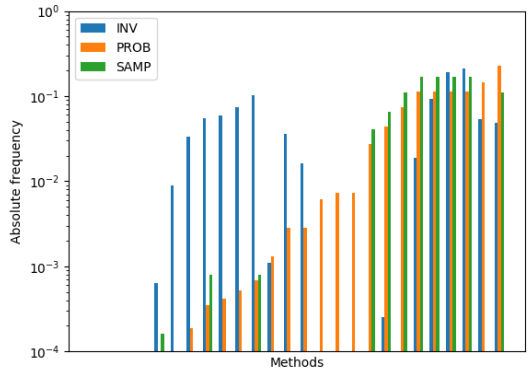
For all subjects, the method call frequency curves for the sample *runs and the* probabilistic *runs match.*

We also perform a statistical analysis on the distributions to increase the confidence in our conclusion. We performed a distribution fitness test (KS - Kolmogorov-Smirnov) on the sample vs. the probabilistic call distribution; and on the sample vs. the inverse probabilistic distribution. It must be noted that the KS test aims at determining whether the distributions are *exactly* the same, whereas we want to ascertain if they are *similar* or *dissimilar*. KS tests are *very sensitive* to small variations in data, which makes it, in principle, inadequate for this objective. In this work, we employ the approach used by Fan [11]—we first estimate the kernel density functions of the data distributions, which smoothen the estimated distribution. Then, we bootstrap and resample new data on the kernel density estimates, and perform the KS test on the bootstrapped data.

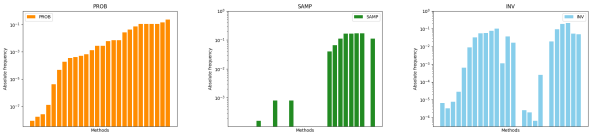
The KS test confirms the results from the the visual inspection, the distribution of the method call frequency of "common inputs" matches the distribution in the sample (*see Table 4*), for some subjects. However, there are also subjects, where the hypothesis is rejected ($p < 0.05$) that method call frequency distributions (sample and "common inputs") come from the same distribution, which is indicated by the blue

TABLE 4
Method coverage for "common inputs" (PROB) and "uncommon inputs" (INV)

Subject	#	Methods covered by sample also by PROB		also by INV		Methods covered by PROB	by INV	Kolmogorov-Smirnov (KS) test of sample vs. PROB D-statistic(p-value)	sample vs. INV D-statistic(p-value)
Argo	52	52	(100%)	32	(62%)	256	165	0.28 (1.11E−9)	0.50 (1.84E−30)
Genson	12	12	(100%)	10	(83%)	218	188	0.25 (3.46E−7)	0.73 (5.88E−64)
Gson	24	24	(100%)	14	(58%)	287	239	0.52 (1.09E−40)	0.25 (1.94E−9)
JSONJava	29	29	(100%)	23	(79%)	51	42	0.08 (0.99)	0.63 (3.45E−11)
Jackson	2	2	(100%)	1	(50%)	957	732	N/A	N/A
JsonToJava	29	29	(100%)	9	(31%)	82	33	0.25 (8.48E−3)	0.24 (1.38E−2)
MinimalJson	24	24	(100%)	18	(75%)	110	100	0.34 (2.36E−6)	0.83 (5.39E−42)
Pojo	23	23	(100%)	7	(30%)	159	93	0.19 (1.81E−3)	0.29 (1.04E−7)
json-simple	11	11	(100%)	10	(91%)	26	24	0.35 (0.09)	0.46 (7.13E−3)
cliftonlabs	23	23	(100%)	23	(100%)	48	48	0.21 (0.25)	0.54 (8.29E−7)
fastjson	70	70	(100%)	62	(89%)	245	231	0.37 (3.07E−15)	0.41 (8.84E−19)
json2flat	6	6	(100%)	5	(83%)	17	14	0.35 (0.24)	0.65 (1.15E−3)
json-flattener	36	36	(100%)	32	(89%)	83	81	0.15 (0.29)	0.15 (0.29)
Rhino	23	6	(26%)	23	(100%)	107	201	0.34 (3.18E−12)	0.45 (6.48E−21)
rhino-sandbox	3	3	(100%)	3	(100%)	17	17	0.47 (0.04)	0.53 (0.02)
CSSValidator	10	10	(100%)	10	(100%)	97	124	0.42 (1.20E−10)	0.38 (7.95E−9)
flute	58	57	(98%)	51	(88%)	148	131	0.29 (3.35E−6)	0.50 (7.34E−18)
jstyleparser	75	74	(99%)	59	(79%)	183	169	0.34 (1.53E−13)	0.38 (1.83E−17)
cssparser	71	71	(100%)	66	(93%)	177	152	0.36 (2.91E−12)	0.62 (4.21E−37)
closure-style	104	95	(91%)	103	(99%)	229	238	0.16 (2.03E−6)	0.09 (3.34E−2)
Total	685	657	(96%)	561	(82%)	3,497	3,022		



(a) **PROB vs. SAMP vs. INV**



(b) **PROB** (c) **SAMP** (d) **INV**

Fig. 11. Call frequency analysis for json-simple

entries. In the case of the Jackson subject, frequencies for the sample calls are all close to zero, which makes the data inadequate for the KS test.

RQ2 *("Uncommon inputs"): Can a learned grammar be modified such it can generate inputs that, opposed to* ***RQ1****, are* in contrast *to those employed during the grammar training?*

For all subjects, the "uncommon inputs" produced by inverting probabilities covered markedly fewer (82%) of the methods covered by the sample (see Table 4). This result shows that the "uncommon inputs" strategy learned the input properties in the samples and produced inputs that avoid several methods covered by the samples.

The "uncommon inputs" strategy triggered markedly fewer methods (82%) called by the initial sample inputs.

Do the "uncommon inputs" trigger fewer of the call sequences covered by the initial samples? Table 3 shows that the "uncommon inputs" strategy triggered significantly fewer (61%) of the call sequences covered by the samples. The number of call sequences induced by the uncommon inputs decreases significantly as the length of the call sequence increases (*see Figure 8*). For instance, comparing frequency charts of call sequences in Figure 7 ((a), (c) and (d)) also show that "uncommon inputs" frequently avoided inducing the call sequences triggered by the initial samples. Notably, for sequences of four consecutive method calls, the "uncommon inputs" strategy covered only 47% of the sequences covered by the initial samples (*see Table 3*). Overall, the "uncommon inputs" avoided inducing the call sequences that were triggered by the initial samples.

The "uncommon inputs" strategy induced significantly fewer call sequences (61%) covered by the initial samples.

Do the "uncommon inputs" only cover *fewer*, or also *different* methods? We perform a visual test to examine if we see a *markedly different* call frequency between the samples and the inputs generated by the "uncommon inputs" strategy. In almost all charts this is the case (see *Figure 9 to Figure 11*). The only exception is the CSSValidator subject.

For all subjects (except CSSValidator), the method call frequency curves for the sample *runs and 'uncommon inputs' runs are markedly different.*

Besides, we examine if the frequency of distribution of method calls for the samples and the "uncommon inputs" are *significantly dissimilar*. In particular, the KS tests shows that for all subjects (except json-flattener) the distributions of method calls in the sample and the "uncommon inputs" are significantly different ($p < 0.05$, *see sample vs. INV in Table 4*).

RQ3 *("Sensitivity to training set variance"): Is our approach sensitive to variance in the initial samples?*

We examine the sensitivity of our approach to the variance in the training set. We randomly selected three distinct training

TABLE 5
Sensitivity to training set variance using three different sets of initial samples containing five inputs each

Set#	Methods covered by sample					Methods covered		Call Sequences covered by sample					Call Sequence covered	
	#	also by PROB		also by INV		by PROB	by INV	#	also by PROB		also by INV		by PROB	by INV
1	685	657	(96%)	561	(82%)	3,497	3,022	3,211	3,059	(94%)	2,198	(61%)	23,530	16,951
2	2,963	2,924	(97%)	2,764	(85%)	8,623	8,246	6,044	5,643	(93%)	4,110	(68%)	22,531	19,896
3	2,656	2,639	(100%)	2,516	(87%)	8,655	8,165	5,005	4,915	(98%)	3,306	(66%)	20,792	19,892

sets, each containing five inputs. Then, for each set, we compare the methods and call sequences covered by the samples to those induced by the generated inputs (Table 5).

Our evaluation showed that *our approach is not sensitive to training set variance*. In particular, for all training sets, the "common inputs" strategy covered most of the methods and call sequences induced by the initial sample inputs. Table 5 shows that the "common inputs" (PROB) consistently covered almost all call sequences (93 to 98%) covered by the initial samples, while "uncommon inputs" (INV) covered significantly fewer call sequences (61 to 68%). Likewise, the "common inputs" consistently covered almost all methods (96 to 100%) covered by the initial samples, while "uncommon inputs" covered fewer methods (82 to 87%) (*cf. Table 5*).

Both strategies, the "common inputs" and the "uncommon inputs", are insensitive to training set variance.

***RQ4** ("Sensitivity to the size of training set"): Is our approach sensitive to the size of the initial samples?*

We evaluate the sensitivity of our approach to the size of the training set. For each input format, we randomly selected four distinct training sets containing N sample inputs, where $N \in \{1, 5, 10, 50\}$. Then, for each set, we compare the methods and call sequences induced by the samples to those induced by the generated inputs (Table 6).

Regardless of the size of the training set, the "common inputs" strategy consistently covered most of the methods and call sequences covered by the initial samples. Specifically, for all sizes, the "common inputs" covered almost all (94 to 99%) of the call sequences covered by the initial samples, while "uncommon inputs" covered significantly fewer call sequences (58 to 79%). In the same vein, the "common inputs" consistently covered almost all methods (96 to 100%) covered by the initial samples, while "uncommon inputs" covered fewer methods (79 to 89%) (*cf. Table 6*). These results demonstrates that the effectiveness of our approach is independent of the size of the training set.

The effectiveness of our approach is independent of the size of the training set used for grammar training.

***RQ5** ("Bugs found"): What kind of crashes (exceptions) do we trigger?*

To address **RQ5**, we examine all of the exceptions triggered by our test suites. We inspect the exceptions triggered during our evaluation of the "common inputs" strategy (in **RQ1**) and the "uncommon inputs" strategy (in **RQ2**). To evaluate if our approach is capable of finding real-world bugs, we compare the exceptions triggered in both **RQ1** and **RQ2** to the exceptions triggered by the input files crawled from *Github* (using the setup described in Section 4.1.6).

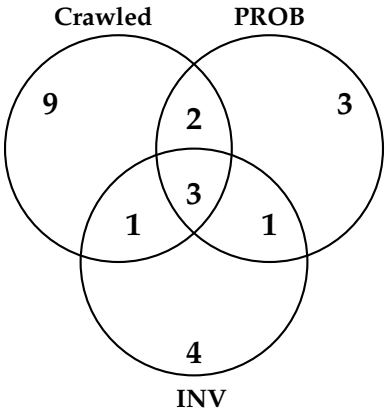


Fig. 12. Number of exceptions triggered by the test suites

Both of our strategies triggered 40% of the exceptions triggered by the crawled files, i.e. six (out of 15) exceptions causing thousands of crashes in four subjects (*cf. Table 7 and Table 8*). Half (three) of these exceptions had no samples of failure-inducing inputs in their grammar training. This indicates that, even without failure-inducing input samples during grammar training, our approach is able to trigger *common buggy behaviors* in the wild, i.e. bugs triggered by the crawled input samples. Exceptions were triggered for JSON and JavaScript input formats, however, no exception was triggered for CSS.

Probabilistic grammar-based testing induced two fifths of all exceptions triggered by the crawled files.

Our strategies were able to trigger eight other exceptions that could not be found by the crawled files (*cf. Figure 12*). This result shows the utility of our approach in finding *rare buggy behaviors*, i.e. uncommon bugs in the crawled input samples. Besides, all of these exceptions were triggered despite a lack of "failure-inducing input" samples in the grammar training. In particular, both strategies triggered nine exceptions each, three and four of which were triggered only by the "common inputs" and only by the "uncommon inputs", respectively.

Probabilistic grammar-based testing induced eight new exceptions that were not triggered by the crawled files.

The "common inputs" strategy triggered all of the exceptions triggered by the original sample inputs used in grammar training. Three exceptions were triggered by the sample inputs and all three exceptions were triggered by the "common inputs" strategy, while "uncommon inputs" triggered only two of these exceptions (*cf. Table 7 and Table 8*). Again, this result confirms that our grammar training approach can learn the input properties that characterize

TABLE 6
Sensitivity to the size of the training set using initial sample size $N \in \{1, 5, 10, 50\}$

Size	Methods covered by sample					Methods covered		Call Sequences covered by sample					Call Sequence covered	
	#	also by PROB		also by INV		by PROB	by INV	#	also by PROB		also by INV		by PROB	by INV
1	1,496	1,490	(100%)	1,352	(79%)	8,715	7,954	1,955	1,942	(99%)	1,135	(58%)	21,279	15,341
5	685	657	(96%)	561	(82%)	3,497	3,022	3,211	3,059	(94%)	2,198	(61%)	23,530	16,951
10	3,546	3,517	(100%)	3,339	(89%)	9,388	11,497	6,297	6,105	(97%)	4,474	(71%)	26,575	21,214
50	5,347	5,313	(100%)	4,961	(89%)	8,950	8,217	9,389	9,076	(97%)	7,421	(79%)	23,512	18,391

TABLE 7
Exception details

Subject	#Exceptions	#Subjects	Average subject crash rate (All)	(Crashed)
SAMP	3	1	0.05263	1
PROB	9	4	0.05999	0.28493
INV	9	7	0.03139	0.08521

specific program behaviors.

The "common inputs" induced all of the exceptions triggered by the original sample inputs.

Overall, 14 exceptions in seven subject programs were found in our experiments (*see Table 7 and Table 8*). On inspection, six of these exceptions affecting five subject programs have been reported to developers as severe bugs. These exceptions have been extensively discussed in the bug repository of each subject program. This result reveals that our approach can generate inputs that reveal real-world buggy behaviors. Additionally, from the evaluation of crawled files, 15 exceptions in five subjects were found. In particular, one exception (Rhino issue #385, which is also reproducible with our approach) has been confirmed and fixed by the developers.

RQ6 *("Failure-inducing inputs"): Can a learned grammar be used to generate inputs that reproduce failure-inducing behavior?*

Let us now investigate if our approach can learn a PCFG from failure-inducing samples and reproduce the failure.

For each exception triggered by the crawled files (in Section 4.1.6), we learned a PCFG from at most five failure-inducing inputs that trigger the exception. Then, we run our PROB approach on the PCFG, using the protocol setting in Section 4.1.2. The goal is to demonstrate that the PCFG learns the input properties of the "failure-inducing inputs", i.e. inputs generated via PROB should trigger *the same exception* as the failure-inducing samples. This is useful for exploring the surroundings of a bug.

In addition, for each exception, we run the inverse of "failure-inducing inputs" (i.e., INV), in order to evaluate if the "uncommon inputs" avoid reproducing the failures. In contrast, for each exception, we run the random generator (RAND) on the CFG (according to Section 4.1.2), in order to evaluate the probability of randomly triggering (these) exceptions without a (learned) PCFG. In the random configuration (RAND), production choices have equal probability of generation.

In Table 9, we have summarized the number of reproduced exceptions. We see that probabilistic generation (PROB) reproduced all (15) failure-inducing inputs collected in our corpus. This shows that the grammar training approach effectively captured the distribution of input properties in the failure-inducing inputs. Moreso, it reproduced the program behavior using the learned PCFG.

Learning probabilities from failure-inducing inputs strategy reproduces 100% of the exceptions in our corpus.

For the inverse of "failure-inducing inputs", our evaluation showed that the "uncommon inputs" strategy could avoid reproducing the learned failure-inducing behavior for most (10 out of 15) of the exceptions (*cf . Table 9 and Table 10*).

The "uncommon inputs" strategy could avoid reproducing the learned program behavior for two-thirds of the exceptions.

However, this strategy reproduced a third (five out of 15) of the exceptions in our corpus (*cf . Table 9 and Table 10*). On inspection, we found that "uncommon inputs" reproduced these five exceptions by generating new counter-examples, i.e., new inputs that are different from the initial samples but trigger the same exception. This is because the initial sample of failure-inducing inputs was not general enough to fully characterize all input properties triggering the crash. This result demonstrates that the inverse of "failure-inducing inputs" can explore the boundaries of the learned behavior in the PCFG, hence, it is useful for generating counter-examples.

The "uncommon inputs" strategy generated new counter-examples for one-third of the exceptions in our corpus.

In contrast, the random test suite (RAND) could not trigger *any* of the exceptions in our corpus, as shown in Table 9. This is expected, since a random traversal of the input grammar would need to explore numerous path combinations to find the specific paths that trigger an exception. This result demonstrates the effectiveness of the grammar training and the importance of the PCFG in capturing input properties.

Random input generation could not reproduce any of the exceptions in our corpus.

Furthermore, we examined the proportion of the generated inputs that trigger an exception. In total, for each test configuration and each exception we generated 100,000 inputs. We investigate the proportion of these inputs that trigger the exception.

Our results for this analysis are summarized in Table 10. We see that about 18% of the inputs generated by the "failure-inducing inputs" strategy (PROB) trigger the learned exception, on average. This rate is three times as high as the exception occurrence rate in our corpus (SAMP; 6%).

TABLE 8
Exceptions induced by "Common Inputs" (PROB), and "Uncommon Inputs" (INV)

Input Format	Subject	Exception	#Failure-inducing samples	Occurrence rate in PROB	INV	Crawled Files
CSS	No exceptions triggered					
JSON	Gson	java.lang.NullPointer	0	0	0.0001	0
	Pojo	java.lang.StringIndexOutOfBounds	0	0.0259	0	0.0024
	Pojo	java.lang.NumberFormat	0	0	0.0001	0
	Argo	argo.saj.InvalidSyntax	0	0	0.0023	0.0225
	JSONToJava	org.json.JSON	0	0.0200	0.0200	0.0223
	json2flat	com.fasterxml.jackson.core.JsonParse	0	0	0.0013	0
	json-flattener	com.eclipsesource.json.Parse	0	0	0.0013	0
	json-flattener	java.lang.UnsupportedOperation	0	0.2981	0	0
	json-flattener	java.lang.IllegalArgument	0	0.2554	0	0
	json-flattener	java.lang.NullPointer	0	0.0398	0	0
	json-flattener	java.lang.NumberFormat	0	0.0028	0.0745	0
JavaScript	rhino-sandbox	org.mozilla.javascript.Evaluator	3	0.4905	0.4469	0.5290
	rhino-sandbox	java.lang.IllegalState	1	0.0016	0	0.0010
	rhino-sandbox	org.mozilla.javascript.EcmaError	1	0.0058	0.0500	0.3740

TABLE 9
Reproduced exceptions by sample inputs (SAMP), "failure-inducing inputs" (PROB), inverse of "failure-inducing inputs" (INV) and random grammar-based generation (RAND)

	#Exceptions Reproduced	Other	Average # Failure-inducing inputs
SAMP	15	0	87
PROB	15	21	18,429
INV	5	6	18,080
RAND	0	0	0

About one in five inputs generated by the "failure-inducing inputs" strategy reproduced the failure-inducing exception.

Finally, the "failure-inducing inputs" strategy also produced *new exceptions* not produced by the original sample of failure-inducing inputs. As shown in the "Other" column in Table 9, "failure-inducing inputs" triggered at least one new exception for each exception in our corpus. This result suggests that the PCFG is also useful for exploring the boundaries of the learned behavior, in order to trigger other program behaviors different from the learned program behavior. This is possible because "failure-inducing inputs" not only reproduces the exact features found in the samples, but also their variations and combinations.

The "failure-inducing inputs" strategy discovered new exceptions not triggered by the samples or random generation.

4.3 Threats to Validity

4.3.1 Internal Validity

The main threat to internal validity is the correctness of our implementation. Namely, whether our implementation does indeed learn a probabilistic grammar corresponding to the distribution of the real world samples used as training set. Unfortunately, this problem is not a simple one to resolve. The probabilistic grammar can be seen as a Markov chain, and the aforementioned problem is equivalent to verifying that its equilibrium distribution corresponds to the posterior distribution of the real world samples. The problem is two-fold: first, the number of samples necessary in order to ascertain the posterior distribution is inordinate. Second, even if we had a chance to process such a number of inputs, or if the posterior distribution were otherwise known, it might well be the case that the probabilistic grammar actually has no equilibrium distribution. However, our tests on smaller and simpler grammars suggest that this is not an issue.

A second internal validity threat is present in the technique we use for controlling the size of the generated samples. As described before, a sample's size is defined in terms of the number of expansions in its parsing tree. In order to control the size, we keep track of the number of expansions generated. Once this number crosses a certain threshold (if it actually crosses it at all), all open derivations are closed via their shortest path. This does introduce a bias in the generation that does not exactly correspond to the distribution described by the probabilistic grammar. The effects of such a bias are difficult to determine, and merit further and deeper study. However, not performing this termination procedure would render useless any approach based on probabilistic grammars.

4.3.2 External Validity

Threats to external validity relate to the generalizability of the experimental results. In our case, this is specifically related to the subjects used in the experiments. We acknowledge that we have only experimented with a limited number of input grammars. However, we have selected the subjects with the intention to test our approach on practically relevant input grammars with different complexities, from small to medium size grammars like JSON; and rather complex grammars like JavaScript and CSS. As a result, we are confident that our approach will also work on inputs that can be characterized by context-free grammars with a wide range of complexity. However, we do have evidence that the approach does not seem to be generalizable to combinations of grammars and samples such that they induce the learning of an almost-uniform probabilistic grammar.

4.3.3 Construct Validity

The main threat to construct validity is the metric we use to evaluate the similarity between test suites, namely method call frequency. While the uses of coverage metrics as adequacy criteria is extensively discussed by the community

TABLE 10
Reproducing exceptions by "failure-inducing inputs" (PROB), inverse of "failure-inducing inputs" (INV), and random grammar-based test generation (RAND)

Input Format (#Crawled files)	Subject	Exception	#Failure-inducing inputs	Occurence rate in crawled files	Reproduction rate in PROB	RAND	INV
JSON (8853)	Gson	java.lang.ClassCast	6	0.0007	0.0090	0	0
	Gson	java.lang.IllegalState	22	0.0025	1	0	1
	JSONToJava	java.lang.ArrayIndexOutOfBounds	38	0.0043	0.0025	0	0
	JSONToJava	java.lang.IllegalArgument	1	0.0001	0.0003	0	0
	JSONToJava	org.json.JSON_1	167	0.0189	0.1811	0	0.1811
	JSONToJava	org.json.JSON_2	30	0.0034	1	0	1
	Pojo	java.lang.IllegalArgument	88	0.0099	0.0002	0	0
	Pojo	java.lang.StringIndexOutOfBounds	21	0.0024	0.0471	0	0
JavaScript (1000)	Rhino	java.util.concurrent.Timeout	11	0.0110	0.0048	0	0
	Rhino	java.lang.IllegalState	2	0.0030	0.0001	0	0
	rhino-sandbox	delight.rhinosandox.ScriptDuration	11	0.0110	0.0073	0	0
	rhino-sandbox	org.mozilla.javascript.Evaluator	529	0.5290	0.4560	0	0.4982
	rhino-sandbox	org.mozilla.javascript.EcmaError_1	372	0.3720	0.0056	0	0.0326
	rhino-sandbox	org.mozilla.javascript.EcmaError_2	2	0.0020	0.0002	0	0
	rhino-sandbox	org.mozilla.javascript.JavaScript	1	0.0010	0.0502	0	0
AVERAGE				0.0646	0.1842	0	0.1808

[12], [13], [14], their binary nature (that is, we can either report *covered* or *not covered*) makes them too shallow to differentiate for behavior. The variance intrinsic to the probabilistic generation makes it very likely that at least one sample will cover parts of the code unrelated to those covered by the rest of the suite. Besides, method call frequency is considered a non-structural coverage metric. To mitigate this threat, we also evaluate our test suites using a structural metric, in particular, (frequency of) call sequences.

5 Limitations

Context sensitivity: Although, our probabilistic grammar learning approach captures the distribution of input properties, the learned input distribution is limited to production choices at the syntactic level. This approach does not handle context-sensitive dependencies such as the order, sequences or repetitions of specific input elements. However, our approach can be extended to learn contextual dependencies, e.g. by learning sequences of elements using N-grams [15] or hierarchies of elements using k-paths [16].

Input Constraints: Beyond lexical and syntactical validity, structured inputs often contain input semantics such as checksums, hashes, encryption, or references. Context-free grammars, as applied in this work, do not capture such complex input constraints. Automatically learning such input constraints for test generation is a challenging task [17]. In the future, we plan to automatically learn input constraints to drive test generation, e.g. using attribute grammars.

6 Related Work

Generating software tests. The aim of *software test generation* is to find a sample of inputs that induce executions that sufficiently cover the possible behaviors of the program—including undesired behavior. Modern software test generation relies, as surveyed by Anand et al. [12] on *symbolic code analysis* to solve the path conditions leading to uncovered code [1], [18], [19], [20], [21], [22], [23], [24], *search-based approaches* to systematically evolve a population of inputs towards the desired goal [25], [26], [27], [28], random inputs to programs and functions [29], [30] or a combination of these techniques [31], [32], [33], [34], [35]. Additionally, machine learning techniques can also be applied to create test sequences [36], [37]. All these approaches have in common that they do not require an additional model or annotations to constrain the set of generated inputs; this makes them very versatile, but brings the risk of producing false alarms—failing executions that cannot be obtained through legal inputs.

Grammar-based test generation. The usage of grammars as *producers* was introduced in 1970 by Hanford in his *syntax machine* [38]. Such producers are mainly used for testing compilers and interpreters: CSmith [39] produces syntactically correct C programs, and LANGFUZZ [40] uses a JavaScript grammar to parse, recombine, and mutate existing inputs while maintaining most of the syntactic validity. GENA [41], [42] uses standard symbolic grammars to produce test cases and only applies stochastic annotation during the derivation process to distribute the test cases and to limit recursions and derivation depth. Grammar-based white-box fuzzing [43] combines grammar-based fuzzing with symbolic testing and is now available as a service from Microsoft. As these techniques operate with system inputs, any failure reported is a true failure—there are no false alarms. None of the above approaches use probabilistic grammars, though.

Probabilistic grammars. The foundations of probabilistic grammars date back to the earliest works of Chomsky [44]. The concept has seen several interactions and generalizations with physics and statistics; we recommend the very nice article by Geman and Johnson [45] as an introduction. Probabilistic grammars are frequently used to analyze ambiguous data sequences—in computational linguistics [46] to analyze natural language, and in biochemistry [47] to model and parse macromolecules such as DNA, RNA, or protein sequences. Probabilistic grammars have been used also to model and produce input data for specific domains, such as 3D scenes [48] or processor instructions [49].

The usage of probabilistic grammars for test generation seems rather straightforward, but is still uncommon. The *Geno* test generator for .NET programs by Lämmel and Schulte [50] allowed users to specify probabilities for individual production rules. Swarm testing [51], [52] uses statistics and a variation of random testing to generate

tests that deliberately targets or omits features of interest. These approaches, in contrast to the one we present in this paper, does not use existing samples to learn or estimate probabilities. The approach by Poulding et al. [53], [54] uses a stochastic context-free grammar for statistical testing. The goal of this work is to correctly imitate the operational profile and consequently the generated test cases are similar to what one would expect during normal operation of the system. The test case generation [55], [56] and failure reproduction [57] approaches by Kifetew et al. combine probabilistic grammars with a search-based testing approach. In particular, like our work, StGP [55] also learns stochastic grammars from sample inputs.

Our approach aims to generate inputs that induce (dis)similar program behaviors as the sample inputs. In contrast to our paper, StGP [55] is focused on evolving and mutating the learned grammars to *improve code coverage*. Although, StGP's goal of generating realistic inputs is very similar to our "common inputs" strategy (*see* ***RQ1***), our approach can further generate realistic inputs that are dissimilar to the sample inputs (*see* ***RQ2***). Meanwhile, StGP is not capable of generating dissimilar inputs.

Mining probabilities. Related to our work are approaches that mine grammar rules and probabilities from existing samples. Patra and Pradel [58] use a given parser to mine probabilities for subsequent fuzz testing and to reduce tree-based inputs for debugging [59]. Their aim, however, is not to produce inputs that would be similar or dissimilar to existing inputs, but rather to produce inputs that have a higher likelihood to be syntactically correct. This aim is also shared by two *mining* approaches: GLADE [60] and Learn&Fuzz [61], which learn producers from large sets of input samples even without a given grammar.

All these approaches, however, share the problem of producing only "common inputs"—they can only focus on common features rather than uncommon features, creating a general "tension between conflicting learning and fuzzing goals" [61]. In contrast, our work can specifically focus on "uncommon inputs"—that is, the complement of what has been learned.

Like us, the Skyfire approach [62] aims at also leveraging "uncommon inputs" for probabilistic fuzzing. Their idea is to learn a probabilistic distribution from a set of samples and use this distribution to generate seeds for a standard fuzzing tool, namely AFL [63]. Here, favoring low probability rules is one of many heuristics applied besides low frequency, low complexity, or production limits. Although the tool has shown good results for XML-like languages, results for other, general grammar formats such as JavaScript are marked as "preliminary" only, though.

Mining grammars. Our approach requires a grammar that can be used both for parsing and producing inputs. While engineering such a grammar may well pay off in terms of better testing, it is still a significant investment in the case of specific domain inputs where such a grammar might not be immediately available. Mining input structures [64], as exemplified using the above GLADE [60] and Learn&Fuzz [61] approaches, may assist in this task. AUTOGRAM [65] and MIMID [66] mine human-readable input grammars exploiting structure and identifiers of a program processing the input, which makes them particularly promising.

7 Conclusions and Future Work

In this paper we have presented an approach that allows engineers, using a grammar and a set of input samples, to generate instances that are either similar or dissimilar to these samples. Similar samples are useful, for instance, when learning from failure-inducing inputs; while dissimilar samples could be used to leverage the testing approach to explore previously uncovered code. Our approach provides a simple, general, and cost-effective means to generate test cases that can then be targeted to the commonly used portions of the software, or to the rarely used features.

Despite their usefulness for test case generation, grammars—including probabilistic grammars—still have a lot of potential to explore in research, and a lot of ground to cover in practice. Our future work will focus on the following topics:

Deep models. At this point, our approach captures probabilistic distributions only at the level of individual rules. However, probabilistic distributions could also capture the occurrence of elements in particular *contexts*, and differentiate between them. For instance, if a `"+"` symbol rarely occurs within parentheses, yet frequently outside of them, this difference would, depending on how the grammar is structured, not be caught by our approach. The domain of computational linguistics [46] has introduced a number of models that take context into account. In our future work, we shall experiment with deeper context models, and determining their effect on capturing common and uncommon input features.

Grammar learning. The big cost of our approach is the necessity of a formal grammar for both parsing and producing—a cost that can boil down to 1–2 programmer days if a formal grammar is already part of the system (say, as an input file for parser generators), but also extend to weeks if it is not. In the future, we will be experimenting with approaches that *mine grammars* from input samples and programs [65], [66] with the goal of extending the resulting grammars with probabilities for probabilistic fuzzing.

Debugging. Mined probabilistic grammars could be used to characterize the features of failure-inducing inputs, separating them from those of passing inputs. Statistical fault localization techniques [67], for instance, could then identify input elements most likely associated with a failure. Generating "common inputs", as in this paper, and testing whether they cause failures, could further strengthen correlations between input patterns and failures, as well as narrow down the circumstances under which the failure occurs.

We are committed to making our research accessible for replication and extension. The source code of our parsers and production tools, the raw input samples, as well as all raw obtained data and processed charts is available as a replication package:

https://tinyurl.com/inputs-from-hell

Acknowledgment

We thank the reviewers for their helpful comments. This work was (partially) funded by Deutsche Forschungsgemeinschaft, Project "Extracting and Mining of Probabilistic Event Structures from Software Systems (EMPRESS)".

REFERENCES

[1] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: automatically generating inputs of death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, pp. 10:1–10:38, 2008.

[2] J. E. Hopcroft, R. Motwani, and J. D. Ullman, "Introduction to automata theory, languages, and computation," *Acm Sigact News*, vol. 32, no. 1, pp. 60–65, 2001.

[3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[4] S. Luke, "Two fast tree-creation algorithms for genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 4, no. 3, pp. 274–283, Sep 2000.

[5] T. Parr, *The Definitive ANTLR 4 Reference*, 2nd ed. Pragmatic Bookshelf, 2013.

[6] K. O'Hair, "HPROF: a Heap/CPU profiling tool in J2SE 5.0," *Sun Developer Network, Developer Technical Articles & Tips*, vol. 28, 2004.

[7] R. Feldt, R. Torkar, T. Gorschek, and W. Afzal, "Searching for cognitively diverse tests: Towards universal test diversity metrics," in *First International Conference on Software Testing Verification and Validation, ICST 2008*. IEEE Computer Society, 2008, pp. 178–186.

[8] Q. Shi, Z. Chen, C. Fang, Y. Feng, and B. Xu, "Measuring the diversity of a test set with distance entropy," *IEEE Trans. Reliability*, vol. 65, no. 1, pp. 19–27, 2016.

[9] W. Jin and A. Orso, "Bugredux: reproducing field failures for in-house debugging," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 474–484.

[10] A. Buja, D. Cook, H. Hofmann, M. Lawrence, E.-K. Lee, D. F. Swayne, and H. Wickham, "Statistical inference for exploratory data analysis and model diagnostics," *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 367, no. 1906, pp. 4361–4383, 2009.

[11] Y. Fan, "Testing the goodness of fit of a parametric density function by kernel method," *Econometric Theory*, vol. 10, no. 2, pp. 316–356, 1994.

[12] S. Anand, E. K. Burke, T. Y. Chen, J. A. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[13] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*, L. C. Briand and A. L. Wolf, Eds. IEEE Computer Society, 2007, pp. 85–103.

[14] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, 1997.

[15] M. Damashek, "Gauging similarity with n-grams: Language-independent categorization of text," *Science*, vol. 267, no. 5199, pp. 843–848, 1995.

[16] N. Havrikov and A. Zeller, "Systematically covering input structure," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '19. IEEE Press, 2019, p. 189–199.

[17] M. Mera, "Mining constraints for grammar fuzzing," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 415–418.

[18] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test input generation with Java PathFinder," in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '04. New York, NY, USA: ACM, 2004, pp. 97–107.

[19] M. Böhme, V. Pham, M. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM, 2017, pp. 2329–2344.

[20] C. Cadar, D. Dunbar, and D. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224.

[21] S. Khurshid, C. S. Pasareanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003*, ser. Lecture Notes in Computer Science, H. Garavel and J. Hatcliff, Eds., vol. 2619. Springer, 2003, pp. 553–568.

[22] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, A. L. Hosking, P. T. Eugster, and C. V. Lopes, Eds. ACM, 2013, pp. 19–32.

[23] M. Christakis, P. Müller, and V. Wüstholz, "Guiding dynamic symbolic execution toward unverified program executions," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, L. K. Dillon, W. Visser, and L. Williams, Eds. ACM, 2016, pp. 144–155.

[24] N. Tillmann and J. de Halleux, "Pex—white box test generation for .NET," in *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, ser. Lecture Notes in Computer Science, B. Beckert and R. Hähnle, Eds., vol. 4966. Springer, 2008, pp. 134–153.

[25] P. McMinn, "Search-based software testing: Past, present and future," in *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ser. ICSTW '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 153–163.

[26] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 416–419.

[27] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, Feb 2018.

[28] K. Mao, M. Harman, and Y. Jia, "Sapienz: multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016*, A. Zeller and A. Roychoudhury, Eds. ACM, 2016, pp. 94–105.

[29] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 75–84.

[30] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990.

[31] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 213–223.

[32] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for c," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 263–272.

[33] M. Böhme, V. Pham, M. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM, 2017, pp. 2329–2344.

[34] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[35] L. D. Toffola, C. Staicu, and M. Pradel, "Saying 'hi!' is not enough: mining inputs for effective test generation," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, G. Rosu, M. D. Penta, and T. N. Nguyen, Eds. IEEE Computer Society, 2017, pp. 44–49.

[36] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng, "Automatic text input generation for mobile testing," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017*, S. Uchitel, A. Orso, and M. P. Robillard, Eds. IEEE / ACM, 2017, pp. 643–653.

[37] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based GUI testing of Android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017*, E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds. ACM, 2017, pp. 245–256.

[38] K. V. Hanford, "Automatic generation of test cases," *IBM Systems Journal*, vol. 9, no. 4, pp. 242–257, 1970.

[39] 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: ACM, 2011, pp. 283–294.

[40] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX, 2012, pp. 445–458.

[41] H. Guo and Z. Qiu, "Automatic grammar-based test generation," in *Testing Software and Systems - 25th IFIP WG 6.1 International Conference, ICTSS 2013, Istanbul, Turkey, November 13-15, 2013, Proceedings*, ser. Lecture Notes in Computer Science, H. Yenigün, C. Yilmaz, and A. Ulrich, Eds., vol. 8254. Springer, 2013, pp. 17–32.

[42] H. Guo and Zongyan Qiu, "A dynamic stochastic model for automatic grammar-based test generation," *Softw., Pract. Exper.*, vol. 45, no. 11, pp. 1519–1547, 2015.

[43] 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: ACM, 2008, pp. 206–215.

[44] N. Chomsky, *Syntactic structures*. Mouton, 1957.

[45] S. Geman and M. Johnson, "Probabilistic grammars and their applications," in *In International Encyclopedia of the Social & Behavioral Sciences. N.J. Smelser and P.B*, 2000, pp. 12 075–12 082.

[46] C. D. Manning and H. Schütze, *Foundations of Statistical Natural Language Processing*. Cambridge, MA, USA: MIT Press, 1999.

[47] Y. Sakakibara, M. Brown, R. Hughey, I. S. Mian, K. Sjölander, R. C. Underwood, and D. Haussler, "Stochastic context-free grammers for tRNA modeling," *Nucleic Acids Research*, vol. 22, no. 23, pp. 5112–5120, 1994.

[48] T. Liu, S. Chaudhuri, V. G. Kim, Q. Huang, N. J. Mitra, and T. Funkhouser, "Creating consistent scene graphs using a probabilistic grammar," *ACM Trans. Graph.*, vol. 33, no. 6, pp. 211:1–211:12, Nov. 2014.

[49] O. Cekan and Z. Kotasek, "A probabilistic context-free grammar based random test program generation," in *2017 Euromicro Conference on Digital System Design (DSD)*, Aug 2017, pp. 356–359.

[50] R. Lämmel and W. Schulte, "Controllable combinatorial coverage in grammar-based testing," in *Testing of Communicating Systems*, M. Ü. Uyar, A. Y. Duale, and M. A. Fecko, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 19–38.

[51] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr, "Swarm testing," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 78–88.

[52] M. A. Alipour, A. Groce, R. Gopinath, and A. Christi, "Generating focused random tests using directed swarm testing," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 70–81.

[53] R. Feldt and S. M. Poulding, "Finding test data with specific properties via metaheuristic search," in *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013*. IEEE Computer Society, 2013, pp. 350–359.

[54] S. M. Poulding, R. Alexander, J. A. Clark, and M. J. Hadley, "The optimisation of stochastic grammars to enable cost-effective probabilistic structural testing," *Journal of Systems and Software*, vol. 103, pp. 296–310, 2015.

[55] F. M. Kifetew, R. Tiella, and P. Tonella, "Combining stochastic grammars and genetic programming for coverage testing at the system level," in *Search-Based Software Engineering - 6th International Symposium, SSBSE 2014, Fortaleza, Brazil, August 26-29, 2014. Proceedings*, ser. Lecture Notes in Computer Science, C. L. Goues and S. Yoo, Eds., vol. 8636. Springer, 2014, pp. 138–152.

[56] F. M. Kifetew, R. Tiella, and Paolo Tonella, "Generating valid grammar-based test inputs by means of genetic programming and annotated grammars," *Empirical Software Engineering*, vol. 22, no. 2, pp. 928–961, 2017.

[57] F. M. Kifetew, W. Jin, R. Tiella, A. Orso, and P. Tonella, "Reproducing field failures for programs with complex grammar-based input," in *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*. IEEE Computer Society, 2014, pp. 163–172.

[58] J. Patra and M. Pradel, "Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data," Technical University of Darmstadt, Tech. Rep. TUD-CS-2016-14664, Nov. 2016.

[59] S. Herfert, J. Patra, and M. Pradel, "Automatically reducing tree-structured test inputs," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, G. Rosu, M. D. Penta, and T. N. Nguyen, Eds. IEEE Computer Society, 2017, pp. 861–871.

[60] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 95–110.

[61] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 50–59.

[62] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy, SP 2017*. IEEE Computer Society, 2017, pp. 579–594.

[63] M. Zalewski, "American fuzzy lop," http://lcamtuf.coredump.cx/afl/, 2018, accessed: 2018-01-28.

[64] Z. Lin and X. Zhang, "Deriving input syntactic structure from execution," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*, M. J. Harrold and G. C. Murphy, Eds. ACM, 2008, pp. 83–93.

[65] 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 2016. New York, NY, USA: ACM, 2016, pp. 720–725.

[66] R. Gopinath, B. Mathis, and A. Zeller, "Mining input grammars from dynamic control flow," in *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) 2020*, 2020.

[67] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.