CISPA Helmholtz-Zentrum i.G.

Inputs from Hell
Generating Uncommon Inputs from Common Samples

Esteban Pavese[1], **Ezekiel Soremekun**[2], **Nikolas Havrikov**[2], **Lars Grunske**[1], **and Andreas Zeller**[2]

[1] Humboldt-Universität zu Berlin, Berlin, Germany
{pavesees, grunske}@informatik.hu-berlin.de

[2] CISPA / Saarland University, Saarbrücken, Germany
{ezekiel.soremekun, nikolas.havrikov, zeller}@cispa.saarland

CISPA
HELMHOLTZ-ZENTRUM i.G.

Inputs from Hell
Generating Uncommon Inputs from Common Samples

Esteban Pavese[1], Ezekiel Soremekun[2], Nikolas Havrikov[2], Lars Grunske[1], and Andreas Zeller[2]

[1] Humboldt-Universität zu Berlin, Berlin, Germany

{pavesees, grunske}@informatik.hu-berlin.de

[2] CISPA / Saarland University, Saarbrücken, Germany

{ezekiel.soremekun, nikolas.havrikov, zeller}@cispa.saarland

(Dated December 20, 2018)

Abstract Generating structured input files to test programs can be performed by techniques that produce them from a grammar that serves as the specification for syntactically correct input files. Two interesting scenarios then arise for effective testing. In the first scenario, software engineers would like to generate inputs that are as *similar* as possible to the inputs in common usage of the program, to test the reliability of the program. More interesting is the second scenario where inputs should be as *dissimilar* as possible from normal usage. This is useful for robustness testing and exploring yet uncovered behavior. To provide test cases for both scenarios, we leverage a context-free grammar to parse a set of sample input files that represent the program's common usage, and determine *probabilities* for individual grammar production as they occur during parsing the inputs. *Replicating* these probabilities during grammar-based test input generation, we obtain inputs that are close to the samples. *Inverting* these probabilities yields inputs that are strongly dissimilar to common inputs, yet still valid with respect to the grammar. Our evaluation on three common input formats (JSON, JavaScript, CSS) shows the effectiveness of these approaches in obtaining instances from both sets of inputs.

1 Introduction

During the process of software testing, software engineers typically look at satisfying two goals. First, ensuring that the software works well on *common* inputs, such that the software delivers its promise on the vast majority of cases (and for the vast majority of customers) that will be seen in typical operation. This is usually achieved by having a set of tests (manually written or generated) that covers this common behavior. Besides these *common* inputs, though, it is also advisable to test for *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 well understood [9].

The question that then arises is, how can engineers obtain such uncommon inputs? In this paper, we focus on the problem of generating uncommon (but otherwise syntactically correct and perfectly legal) inputs that are *unlikely to be seen in typical operation*. To this end, we assume the existence of a *context-free grammar* that describes the input language to a program, that is, it describes the set of its valid inputs. Using such a grammar, we can *parse* existing common 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 choices present in productions carry different likelihoods. Since these probabilities come from the common samples used for the quantification, this grammar describes the distribution of valid, but *common* inputs. The key idea is that now we can *invert* these probabilities in order to obtain a second probabilistic grammar. This *inverted* grammar, however, describes in turn the distribution of legal, but *uncommon* inputs. We call them "inputs from hell".

As an example of such "inputs from hell", consider Figure 1, listing two JavaScript inputs generated by focusing on uncommon features. Both of these snippets are valid JavaScript code, but cause the Mozilla Rhino JavaScript compiler to crash during interpretation. They make use of so-called *destructuring assignments*: in JavaScript, it is allowed to 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 not extensively found across JavaScript samples and tests. This is precisely why the aim of our approach is to generate these "inputs from hell".

```
const [c, y, y] = [];
var { a: {} = 'b' } = {};
```

Figure 1: Two inputs from hell that break Rhino 1.7.7.2

This paper makes the following contributions:

1. We show how to use context-free grammars to determine production probabilities from a given set of input samples.

2. We show how to 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. As a result our approach leverages the well-known concept of 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.

3. We show how to 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.

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 *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 3 shows the grammar G_p as extension of G with all probabilities as extracted from the derivation tree of I (Figure 2). 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* ")"`).

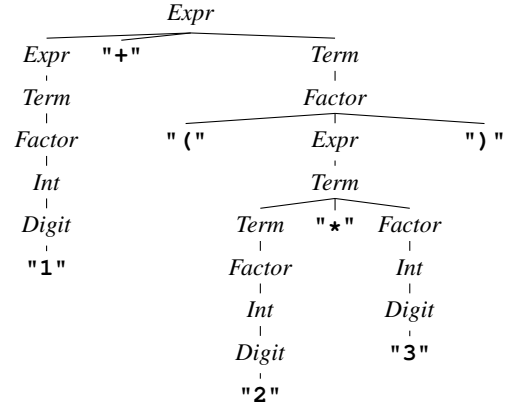Expressions using unary operators like `"+"` *Factor* and `"-"` *Factor* do not occur.

Figure 2: Derivation tree representing `"1 + (2 * 3)"`

If we use G_p from Figure 3 as a probabilistic production grammar, we obtain inputs according to these probabilities. As listed in Figure 4, 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.

```
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";
```

Figure 3: Probabilistic grammar G_p, expanding G

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

Figure 4: Inputs generated from G_p in Figure 3

Replicating "more of the same" features as found in sample inputs makes most sense if these studied inputs can be associated with errors. However if we, as in most cases, only have sample inputs that work just fine, we would typically be interested in inputs that are *different* from our samples. 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 thus "inverted" rule reads now:

Digit $\rightarrow$ 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 5.

Expr $\rightarrow$ 0% *Term* | 0% *Expr* `"+"` *Term*
    | 100% *Expr* `"-"` *Term*;
Term $\rightarrow$ 0% *Factor* | 0% *Term* `"*"` *Factor*
    | 100% *Term* `"/"` *Factor*;
Factor $\rightarrow$ 0% *Int* | 50% `"+"` *Factor*
    | 50% `"-"` *Factor* | 0% `"("` *Expr* `")"`;
Int $\rightarrow$ 100% *Digit Int* | 0% *Digit*;
Digit $\rightarrow$ 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"`;

Figure 5: Grammar $G_{p^{-1}}$ inverted from G_p in Figure 3

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, which we will detail further on.

If we use $G_{p^{-1}}$ as a production grammar—and avoiding infinite production—we obtain inputs as shown in Figure 6. 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. With both the sets of similar and dissimilar inputs, we can expect to have a good set of regression tests as well as a set exploiting less frequently used 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
```

Figure 6: Inputs generated from $G_{p^{-1}}$ from Figure 5

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) [26].

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, $P : V \rightarrow (V \cup T)^*$ 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:

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

In a *probabilistic grammar*, 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:

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

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

Here, the probabilities for `"b"` and `"c"` are not specified; we assume that the complement from `"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

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

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 now 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* [1], representing the structure

of a sentence according to G. As an example, consider Figure 2, 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 2, 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 $S \to A_1 | \ldots | A_n$ as

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

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 3, assigning probabilities to all alternatives.

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 "less of the same" approach promises to be useful if we accept that our samples are not able to cover all the possible behavior of the system under test, 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:

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

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 production 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 [37]. The main idea of this algorithm is to allow the expansion of not-yet-expanded productions, but all the 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. We discuss this issue later in Section 4.
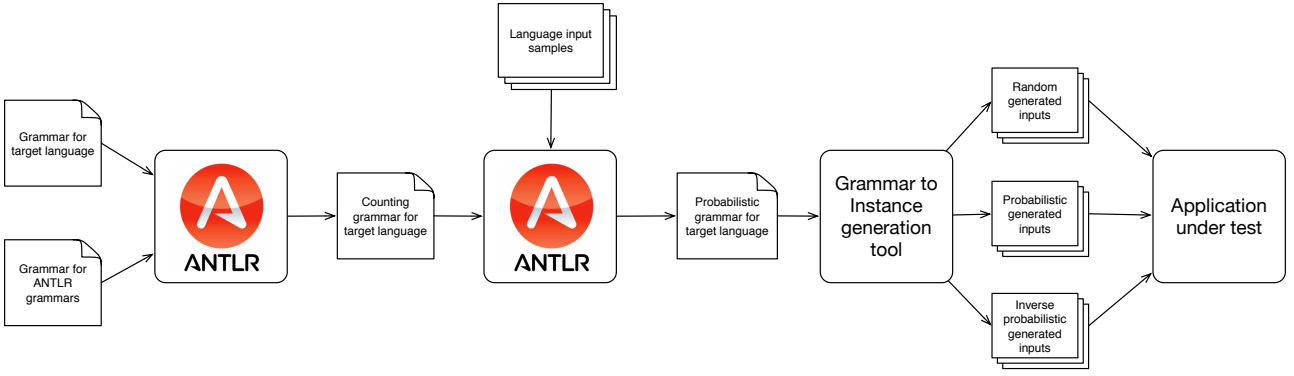
4

Figure 7: Workflow for the generation of *more of the same* and *less of the same*.

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 7.

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 [45].

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 to several case studies. In particular, we ask the following research questions:

> **(RQ1)** Can a learned grammar be used to generate inputs that resemble those that were employed during the grammar training? ("more of the same")

> **(RQ2)** 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? ("less of the same")

To answer the first two questions, 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 call frequency* as a measure of input similarity. We will define this measure later in this section, and we will

discuss its usefulness, as well as alternatives to it, when we discuss the threats to the validity of our validation approach.

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 two 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 as a result of the inversion process described in Section 3.

Expansion size control is carried out in order to avoid unbounded expansion as described in Section 3.

4.1.2 Research Protocol

In our evaluation, we generate test suites and measure the frequency of method calls they induce in our subject programs. We use the HPROF [42] profiler to accurately monitor the number of method calls for each input, since all subjects are implemented in Java. For each input language, the experimental protocol proceeds as follows:

a) We randomly select five files from a pool of thousands of sample files crawled from GitHub code repositories, and through our approach produce a probabilistic grammar out of them.

b) We feed the sampled input files into the subject program and record the frequency of method calls using HPROF [42].

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).

5

d) We test each subject program, by feeding the input files into the subject program and recording the frequency of method calls using HPROF [42].

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

4.1.3 Subject Programs

In order to validate our approach, we evaluated the technique 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 1 summarizes the subjects to be analyzed, their input format and the number of methods in each implementation.

Subject	# of methods	Input language
Argo	523	
Genson	1182	
Gson	793	
JSONJava	202	
Jackson	5378	JSON
JsonToJava	294	
MinimalJson	224	
Pojo	445	
json-simple	63	
Rhino	4873	JavaScript
cssValidator	7774	CSS3

Table 1: Subjects and input language for approach validation.

The initial, unquantified grammars for the input subjects were adapted from those in the repository of the well-known parser generator ANTLR [45][1]. 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 the artefact package submitted along this paper.

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 (dis)similarity [15, 51], 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 a measure of input similarity in terms of their method call frequency. We will say two inputs are similar if they trigger a similar distribution in the frequency with which the methods of the piece of

software under analysis are called. Of course, such a notion allows for a great variance drift if we were to compare only two inputs. Therefore, we perform this comparison on test suites as a whole to dampen the effect of this variance.

Using this proxy measure of method call frequency, we will aim at answering **(RQ1)** and **(RQ2)**. The first question will be answered satisfactorily if the distribution of call frequencies when running the subjects on *probabilistically* generated suites is similar to the frequency when running the software on the *training* samples. Likewise, the second question will be answered positively if the call frequency distributions for suites generated with the *inverse* approach are markedly different.

4.2 Experimental results

In the figures below ranging from Figure 8 to Figure 13, we show a representative selection of our results[2]. In this section, we describe the data depicted in these charts and offer our interpretations.

For each subject, two charts are constructed. In both charts, the horizontal axis (which is otherwise unlabelled) represents the set of methods in the subject, ordered by the *frequency of calls* in the experiment on *probabilistic inputs*. The chart at the top represents the *accumulated* call frequency for each strategy (calls in the samples, in probabilistic inputs and inverse probabilistic inputs), as we consider more and more methods. The chart at the bottom represents the absolute call frequency for each method. In each chart, the data series corresponding to the *sample runs* is depicted in **blue**, the series corresponding to the *probabilistic runs* in **green**, and the series for the *inverse probabilistic runs* in **orange**.

4.2.1 Research Question 1

In order to argue for a positive answer for **(RQ1)**, we need to compare the statistical distributions resulting from our strategies. Such a comparison is a notoriously difficult problem, with the common advice being to run a visual test [7]. We need to be able to see a pattern in frequency calls such that the accumulated curves for the *sample* runs and the *probabilistic* runs roughly match.[3] It can be seen that this match does hold in all JSON examples very closely, and to a further extent also by the Rhino JS interpreter and CSSValidator.

We also perform a statistical analysis on the distributions to increase the confidence in our conclusion. To this end we aim at performing 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,

---

[1]The original grammars can be found at `https://github.com/antlr/grammars-v4`.

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 in [13]—we first estimate the kernel density functions of the data distributions, which smoothens the estimated distribution. Then, we bootstrap and resample new data on the kernel density estimates, and perform the KS test on the bootstrapped data. Results are shown in Table 2, column (C). Results range from strong (in blue) to inconclusive (in orange and red) on relating the sample to the probabilistic data[4].

> *In almost subjects, the "more of the same" method call frequency distribution matches the distribution in the sample. In the other cases, results are mostly inconclusive.*

In the cases of most discrepancy, they can be explained looking at the absolute frequencies, looking for the spikes that cause the curves to mismatch. In the case of the Rhino JS interpreter, two spikes distinguish themselves clearly, with frequencies hovering around 24% and 39% of all calls—that is, they account for more than *half* of the total method calls on the sample. In looking at the data, it turns out that these spikes correspond to methods `org.mozilla.javascript.TokenStream.getChar` and `org.mozilla.javascript.TokenStream.<init>`. This is explained by the fact that the samples have real-world properties, such as sensibly-named (and therefore longer) variables and methods, whereas our approach tends to generate much shorter names. In the case of the CSSValidator subject, the situation is similar. The frequency chart shows three distinct spikes which correspond to methods `util.Utf8Properties.continueLine` (8%), `util.Utf8Properties.loadConversion` (19%) and `util.Utf8Properties.removeWhiteSpaces` (23%), which deal with utf-8 conversions. Again, the load on those methods is larger, as names are longer in real-world samples.

4.2.2 Research Question 2

In this case, we want to check if we see a *markedly different* accumulated frequency between the samples and the inputs generated by the inverse probabilistic approach. Again, it can easily be seen that in almost all charts this is the case, except for the CSSValidator subject.

> *With the exception of CSSValidator, the method call frequency distribution of "less of the same" is markedly different from the distribution in the sample.*

Most intriguing for this subject is the fact that the curves for the probabilistic and inverse-probabilistic generations

Figure 8: Call frequency analysis for Gson

Figure 9: Call frequency analysis for JSONJava

---

[4]In the case of the Jackson exemplars, frequencies for the sample calls are all close to zero, which makes the data inadequate for the KS test.
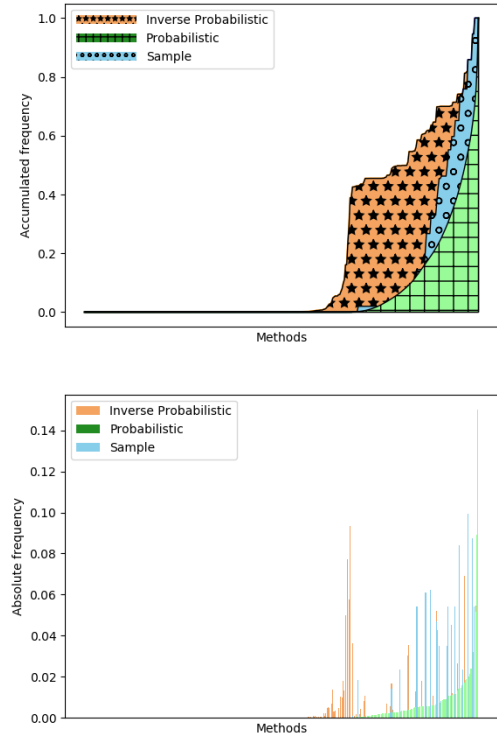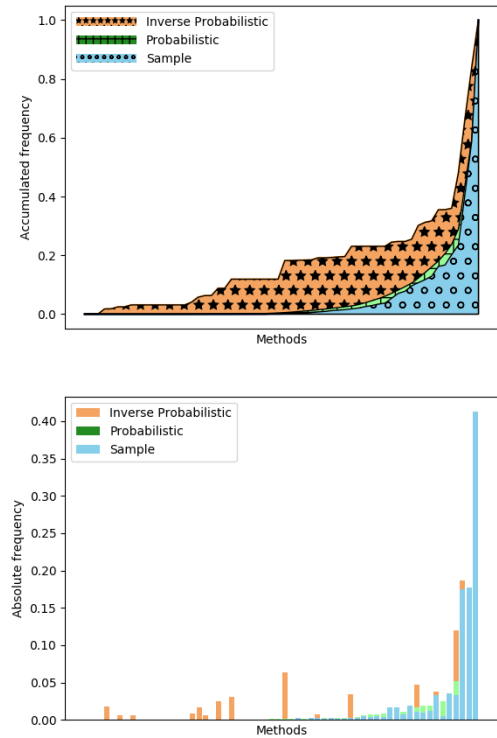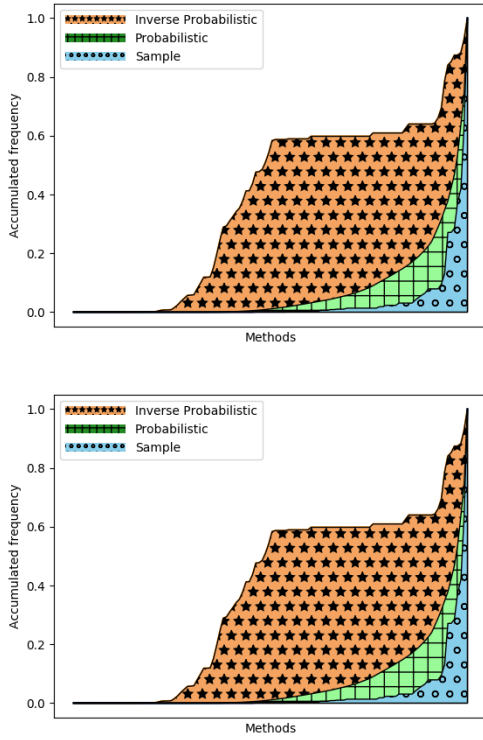
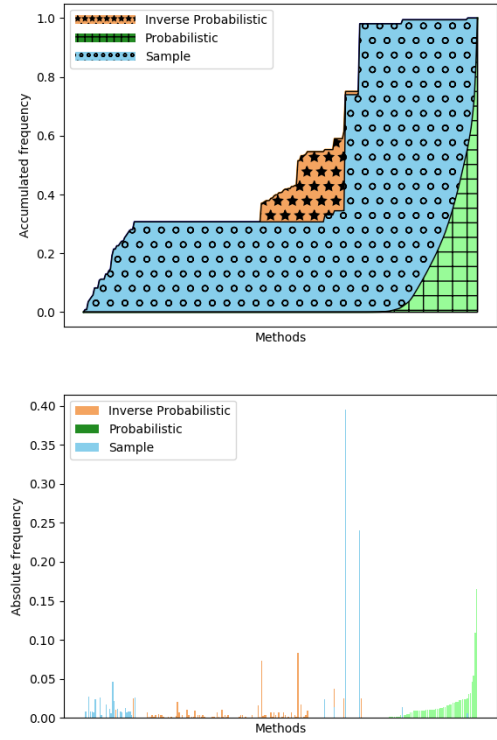Figure 10: Call frequency analysis for MinimalJson

Figure 12: Call frequency analysis for Rhino
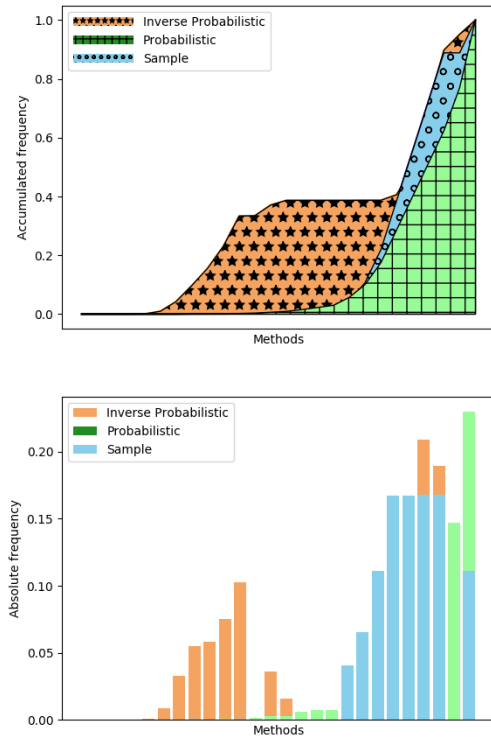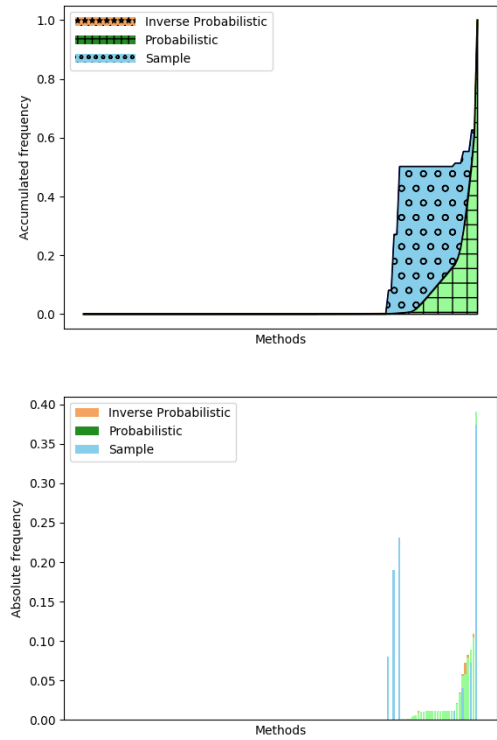

Figure 11: Call frequency analysis for json-simple

Figure 13: Call frequency analysis for CSSValidator

Subject	A	B	C	D
Argo	10	+3.32%	(0.37, 1.33e−6)	(0.57, 4.41e−15)
Genson	24	+8.16%	(0.13, 0.34)	(0.51, 3.70e−12)
Gson	21	+6.14%	(0.08, 0.89)	(0.43, 9.12e−9)
JSONJava	6	+10.00%	(0.15, 0.19)	(0.61, 3.28e−17)
Jackson	20	**+2.22%**	N/A	N/A
JsonToJava	11	+12.09%	(0.23, 8.21e−2)	(0.59, 3.96e−16)
MinimalJson	25	+20.49%	(0.36, 2.85e−6)	(0.58, 1.34e−15)
Pojo	18	+8.78%	(0.21, 0,02)	(0.53, 4.26e−13)
json-simple	8	**+30.77%**	(0.27, 1.03e−2)	(0.58, 1.34e−15)
Rhino	13	+5.22%	(0.70, 1.57e-22)	(0.51, 3.69e−12)
cssValidator	17	+11.88%	(0.49, 2.95e−11)	(0.48, 8.08e−11)

Table 2: (A): number of methods called by the inverse approach that are *never* called by the samples - (B): percentage increase of (A) w.r.t. sample - (C, D): smoothed bootstrapped Kolmogorov-Smirnov tests for distributions (C): sample vs. probabilistic; (D): sample vs. inverse (test statistic, p-value).

fit each other almost perfectly. An in-depth analysis of the learned probabilistic grammar, however, revealed that the probabilistic grammar is *almost uniform*, which explains why the inverted probabilistic grammar would look very much alike. The results for **(RQ1)** and **(RQ2)** on this subject suggest that the approach does not work very well when the samples induce an almost-uniform grammar; and that, apparently, the original CSS grammar and real world samples are such that they don't allow for much variety, therefore resulting in such an almost-uniform grammar.

> *A "less of the same" strategy works best if the element distributions in the sample input is non-uniform.*

Further evidence on the power of the "less of the same" approach is shown in Table 2. Column (A) shows the absolute number of methods that were frequently called in the "less of the same" inputs that were not called at all in the samples. Column (B) shows this data as a percentage of the methods not covered in the sample. In column (D) we perform the distribution fitting test between the sample call distribution and the inverse probabilistic one. All results show the distributions are markedly different with strong statistical evidence.

4.3 Threats to Validity

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[5]. 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.

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.

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 [2, 4, 60], 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. Indeed, we carried out coverage-based experiments on our probabilistically and inverse-probabilistically generated suites, and this metric turned out to be inadequate, as we did not find significant differences when looking at binary notions of coverage.

5   Related Work

Software Test Generation. 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. [2] on *symbolic code analysis* to solve the path conditions leading to uncovered code [55, 5, 8, 9, 28, 33, 12, 53], *search-based approaches*

---

[5]A Markov chain with multiple bottom (isolated) connected components will have no equilibrium distributions. It can be easily the case that a grammar has such a multiplicity of connected components.

to systematically evolve a population of inputs towards the desired goal [40, 16, 44, 39], random inputs to programs and functions [43, 41] or a combination of these techniques [19, 50, 6, 48, 54]. Additionally, machine learning techniques can also be applied to create test sequences [35, 52]. 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* [23]. Such producers are mainly used for testing compilers and interpreters: CSmith [58] produces syntactically correct C programs, and LANGFUZZ [25] uses a JavaScript grammar to parse, recombine, and mutate existing inputs while maintaining most of the syntactic validity. GENA [22, 21] 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 [18] 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 [11]. The concept has seen several interactions and generalizations with physics and statistics; we recommend the very nice article by Geman and Johnson [17] as an introduction. Probabilistic grammars are frequently used to analyze ambiguous data sequences—in computational linguistics [38] to analyze natural language, and in biochemistry [49] 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 [36] or processor instructions [10].

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 [32] allowed users to specify probabilities for individual production rules. This approach, in contrast to the one we present in this paper, does not use existing samples to learn or estimate probabilities. The test case generation [29, 30] and failure reproduction [31] approaches by Kifetew et al. combine probabilistic grammars with a search-based testing approach. The results [30] show that the combination produces a large percentage of correct inputs and, based on the fitness function, produces a high-branch coverage. However, due to the search-based nature of the approach, a large number of system evaluations to determine the fitness of the generated test cases are required. The approach by Poulding et al. [14, 47] uses a stochastic context-free grammar for statistical testing. The goal of this work is thus 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.

Mining Probabilities. Related to our work are approaches that mine grammar rules and probabilities from existing samples. Patra and Pradel [46] use a given parser to mine probabilities for subsequent fuzz testing and to reduce tree-based inputs for debugging [24]. 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 *grammar mining* approaches: GLADE [3] and Learn&Fuzz [20], which learn producers from large sets of input samples even without a given grammar.

All these approaches, however, share the problem of producing only "more of the same"—they can only focus on common features rather than uncommon features, creating a general "tension between conflicting learning and fuzzing goals" [20]. In contrast, our work can specifically focus on uncommon inputs—that is, the complement of what has been learned.

Like us, the Skyfire approach [56] 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 [59]. Here, favoring low probability rules is one of many heuristics applied besides low frequency, low complexity, or production limits. The tool requires, however, the specification of a context-dependent grammar. 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 [34], as exemplified using the above GLADE [3] and Learn&Fuzz [20] approaches, may assist in this task. The AUTOGRAM approach by Höschele and Zeller [27] mines human-readable input grammars exploiting structure and identifiers of a program processing the input, which makes it particularly promising.

6 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 [38] 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 [3, 20, 27, 34] 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 [57], for instance, could then identify input elements most likely associated with a failure. Generating "more of the same" 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`

References

[1] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[2] Saswat Anand et al. "An orchestrated survey of methodologies for automated software test case generation". In: *Journal of Systems and Software* 86.8 (2013), pp. 1978–2001.

[3] Osbert Bastani et al. "Synthesizing Program Input Grammars". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, 2017, pp. 95–110.

[4] Antonia 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*. Ed. by Lionel C. Briand and Alexander L. Wolf. IEEE Computer Society, 2007, pp. 85–103.

[5] Marcel Böhme et al. "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*. Ed. by Bhavani M. Thuraisingham et al. ACM, 2017, pp. 2329–2344.

[6] Marcel Böhme et al. "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*. Ed. by Bhavani M. Thuraisingham et al. ACM, 2017, pp. 2329–2344.

[7] Andreas Buja et al. "Statistical inference for exploratory data analysis and model diagnostics". In: *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 367.1906 (2009), pp. 4361–4383.

[8] Cristian Cadar, Daniel Dunbar, and Dawson 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*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 209–224.

[9] Cristian Cadar et al. "EXE: Automatically Generating Inputs of Death". In: *ACM Trans. Inf. Syst. Secur.* 12.2 (2008), 10:1–10:38.

[10] 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.

[11] Noam Chomsky. *Syntactic structures*. Mouton, 1957.

[12] Maria Christakis, Peter Müller, and Valentin 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*. Ed. by Laura K. Dillon, Willem Visser, and Laurie Williams. ACM, 2016, pp. 144–155.

[13] Yanqin Fan. "Testing the goodness of fit of a parametric density function by kernel method". In: *Econometric Theory* 10.2 (1994), pp. 316–356.

[14] Robert Feldt and Simon 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.

[15] Robert Feldt et al. "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.

[16] Gordon Fraser and Andrea 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*. ESEC/FSE '11. Szeged, Hungary: ACM, 2011, pp. 416–419.

[17] Stuart Geman and Mark Johnson. "Probabilistic Grammars and their Applications". In: *In International Encyclopedia of the Social & Behavioral Sciences. N.J. Smelser and P.B.* 2000, pp. 12075–12082.

[18] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. "Grammar-based Whitebox Fuzzing". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. Tucson, AZ, USA: ACM, 2008, pp. 206–215.

[19] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed Automated Random Testing". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. Chicago, IL, USA: ACM, 2005, pp. 213–223.

[20] Patrice Godefroid, Hila Peleg, and Rishabh Singh. "Learn&Fuzz: Machine Learning for Input Fuzzing". In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. ASE 2017. Urbana-Champaign, IL, USA: IEEE Press, 2017, pp. 50–59.

[21] Hai-Feng Guo and Zongyan Qiu. "A dynamic stochastic model for automatic grammar-based test generation". In: *Softw., Pract. Exper.* 45.11 (2015), pp. 1519–1547.

[22] Hai-Feng Guo and Zongyan 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*. Ed. by Hüsnü Yenigün, Cemal Yilmaz, and Andreas Ulrich. Vol. 8254. Lecture Notes in Computer Science. Springer, 2013, pp. 17–32.

[23] Kenneth V. Hanford. "Automatic generation of test cases". In: *IBM Systems Journal* 9.4 (1970), pp. 242–257.

[24] Satia Herfert, Jibesh Patra, and Michael 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*. Ed. by Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen. IEEE Computer Society, 2017, pp. 861–871.

[25] Christian Holler, Kim Herzig, and Andreas 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.

[26] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. "Introduction to automata theory, languages, and computation". In: *Acm Sigact News* 32.1 (2001), pp. 60–65.

[27] Matthias Höschele and Andreas Zeller. "Mining Input Grammars from Dynamic Taints". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. Singapore, Singapore: ACM, 2016, pp. 720–725.

[28] Sarfraz Khurshid, Corina S. Pasareanu, and Willem 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*. Ed. by Hubert Garavel and John Hatcliff. Vol. 2619. Lecture Notes in Computer Science. Springer, 2003, pp. 553–568.

[29] Fitsum Meshesha Kifetew, Roberto Tiella, and Paolo 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*. Ed. by Claire Le Goues and Shin Yoo. Vol. 8636. Lecture Notes in Computer Science. Springer, 2014, pp. 138–152.

[30] Fitsum Meshesha Kifetew, Roberto Tiella, and Paolo Tonella. "Generating valid grammar-based test inputs by means of genetic programming and annotated grammars". In: *Empirical Software Engineering* 22.2 (2017), pp. 928–961.

[31] Fitsum Meshesha Kifetew et al. "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.

[32] Ralf Lämmel and Wolfram Schulte. "Controllable Combinatorial Coverage in Grammar-Based Testing". In: *Testing of Communicating Systems*. Ed. by M. Ümit Uyar, Ali Y. Duale, and Mariusz A. Fecko. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 19–38.

[33] You Li et al. "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*. Ed. by Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes. ACM, 2013, pp. 19–32.

[34] Zhiqiang Lin and Xiangyu 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*. Ed. by Mary Jean Harrold and Gail C. Murphy. ACM, 2008, pp. 83–93.

[35] Peng Liu et al. "Automatic text input generation for mobile testing". In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017*. Ed. by Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard. IEEE, 2017, pp. 643–653.

[36] Tianqiang Liu et al. "Creating Consistent Scene Graphs Using a Probabilistic Grammar". In: *ACM Trans. Graph.* 33.6 (Nov. 2014), 211:1–211:12.

[37] S. Luke. "Two fast tree-creation algorithms for genetic programming". In: *IEEE Transactions on Evolutionary Computation* 4.3 (Sept. 2000), pp. 274–283.

[38] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. Cambridge, MA, USA: MIT Press, 1999.

[39] Ke Mao, Mark Harman, and Yue Jia. "Sapienz: multi-objective automated testing for Android applications". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016*. Ed. by Andreas Zeller and Abhik Roychoudhury. ACM, 2016, pp. 94–105.

[40] Phil 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*. ICSTW '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 153–163.

[41] Barton P. Miller, Louis Fredriksen, and Bryan So. "An Empirical Study of the Reliability of UNIX Utilities". In: *Commun. ACM* 33.12 (Dec. 1990), pp. 32–44.

[42] Kelly O'Hair. "HPROF: a Heap/CPU profiling tool in J2SE 5.0". In: *Sun Developer Network, Developer Technical Articles & Tips* 28 (2004).

[43] Carlos Pacheco et al. "Feedback-Directed Random Test Generation". In: *Proceedings of the 29th International Conference on Software Engineering*. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 75–84.

[44] A. Panichella, F. M. Kifetew, and P. Tonella. "Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets". In: *IEEE Transactions on Software Engineering* 44.2 (Feb. 2018), pp. 122–158.

[45] Terence Parr. *The Definitive ANTLR 4 Reference*. 2nd. Pragmatic Bookshelf, 2013.

[46] Jibesh Patra and Michael Pradel. *Learning to Fuzz: Application-Independent Fuzz Testing with Probabilistic, Generative Models of Input Data*. Tech. rep. TUD-CS-2016-14664. Technical University of Darmstadt, Nov. 2016.

[47] Simon M. Poulding et al. "The optimisation of stochastic grammars to enable cost-effective probabilistic structural testing". In: *Journal of Systems and Software* 103 (2015), pp. 296–310.

[48] Sanjay Rawat et al. "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.

[49] Yasubumi Sakakibara et al. "Stochastic context-free grammers for tRNA modeling". In: *Nucleic Acids Research* 22.23 (1994), pp. 5112–5120.

[50] Koushik Sen, Darko Marinov, and Gul 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*. ESEC/FSE-13. Lisbon, Portugal: ACM, 2005, pp. 263–272.

[51] Qingkai Shi et al. "Measuring the Diversity of a Test Set With Distance Entropy". In: *IEEE Trans. Reliability* 65.1 (2016), pp. 19–27.

[52] Ting Su et al. "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*. Ed. by Eric Bodden et al. ACM, 2017, pp. 245–256.

[53] Nikolai Tillmann and Jonathan 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*. Ed. by Bernhard Beckert and Reiner Hähnle. Vol. 4966. Lecture Notes in Computer Science. Springer, 2008, pp. 134–153.

[54] Luca Della Toffola, Cristian-Alexandru Staicu, and Michael Pradel. "Saying ʿHI!ʾ 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*. Ed. by Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen. IEEE Computer Society, 2017, pp. 44–49.

[55] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. "Test Input Generation with Java PathFinder". In: *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '04. Boston, Massachusetts, USA: ACM, 2004, pp. 97–107.

[56] Junjie Wang et al. "Skyfire: Data-Driven Seed Generation for Fuzzing". In: *2017 IEEE Symposium on Security and Privacy, SP 2017*. IEEE Computer Society, 2017, pp. 579–594.

[57] W Eric Wong et al. "A survey on software fault localization". In: *IEEE Transactions on Software Engineering* 42.8 (2016), pp. 707–740.

[58] Xuejun Yang et al. "Finding and Understanding Bugs in C Compilers". In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: ACM, 2011, pp. 283–294.

[59] Michal Zalewski. *American Fuzzy Lop*. `http://lcamtuf.coredump.cx/afl/`. Accessed: 2018-01-28. 2018.

[60] Hong Zhu, Patrick A. V. Hall, and John H. R. May. "Software Unit Test Coverage and Adequacy". In: *ACM Comput. Surv.* 29.4 (1997), pp. 366–427.