Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Evolutionary grammar-based fuzzing

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von: Martin Eberlein
geboren am: 20.08.1995
geboren in: Nürnberg

Gutachter/innen: Prof. Dr. Lars Grunske
Prof. Dr. Timo Kehrer

eingereicht am: ..... verteidigt am: .....

Contents

Abstract

The software testing technique *fuzzing* has shown great success in finding defects and vulnerabilities. A *fuzzer* provides randomly generated inputs to a targeted software to expose erroneous behavior. To efficiently detect defects and bugs, generated inputs should be conforming to the basic structural semantics of the input format. This thesis presents an evolutionary grammar-based fuzzing approach to generate test cases, exposing erroneous behavior in parsers and interpreters. To reduce the time of generating non-conforming input files, the *fuzzer* uses probabilistic grammars to generate syntactically correct inputs. The *fuzzer* uses an evolutionary optimization approach to generate "complex" and "interesting" individuals, that may be more likely to trigger exceptional behavior. The evaluation of this evolutionary *fuzzer* shows the effectiveness of this approach in detecting defects and crashes. Applied to real-world applications on three common input formats (JSON, JavaScript, CSS), the approach was able to expose 13 unique defects.

1 Introduction

1.1 The problem and its setting

Software security vulnerabilities can be extremely costly ranging from 250,000$ to 300,000 US$ per security breach [1]. Hunting down those issues and deploying more secure software is highly anticipated and has therefore been subject of recent research. Modern software security testing uses a variety of different approaches and techniques, but can be extremely complex and time consuming. One way to reduce the effort is to automatically generate input samples to expose bugs and unwanted behavior. Recently fuzzing, a software testing technique, has shown great success in finding vulnerabilities and erroneous behavior in a variety of different programs and software [2], [3]. A *fuzzer* generates random input data and enhances or mutates them according to certain criteria, pointing at potential defects or software vulnerabilities.

Modern browsers combine a wide variety of interconnected components, using different interpreter and languages like JavaScript, Java, CSS or JSON to deliver the wanted results to the user. This makes web browsers extremely prone to hackers exploiting the growing set of embedded parsers and interpreters to lunch malicious attacks. Hallaraker at al. [4] have shown that in particular the JavaScript interpreter, which is used to enhance the client side display of web pages, is responsible for high level security issues, allowing attackers to steal users' credentials and lure users into divulging sensitive information. Unfortunately, due to the steady increasing complexity, interpreters become increasingly hard to test and verify.

Greybox fuzzer use light weight instrumentation of a program rather than extensive program analysis. This reduces the overhead significantly and makes greybox fuzzer extremely sufficient vulnerability detection tools. Nevertheless, greybox fuzzer still struggle to create semantically and syntactically correct inputs. The lack of the structural input awareness is considered to be the main limitation. Since greybox fuzzer usually apply mutation on the bit level representation of an input, it is hard to keep a high level, syntactically correct structure. Yet, to detect vulnerabilities deep inside programs, complex input files are needed.

To resolve this issue, this thesis presents an evolutionary grammar-based *fuzzing tool* to detect defects and unwanted behavior in parsers and interpreters. The described *fuzzer* builds on the recent work of Pavese et al. [5] and extends their proposed ideas on input generation with an evolutionary optimization approach. With the work of Pavese et al. as the foundation and the usage of probabilistic grammars, the fuzzer is able to generate syntactically correct inputs. Further, the presented approach of an evolutionary process is able to generate "interesting" and "complex" input files (e.g. *nested loops*). Utilizing the probabilistic grammar to generate new populations allows for good guiding properties. By selecting the most promising inputs of a population and by learning the probabilistic grammar accordingly, essentially favoring specific production rules from the previously population, this process allows the directed creation of inputs towards specific features.

1.2 Goal and structure of this thesis

The goal of this thesis is to construct an evolutionary grammar-based fuzzing tool to test parsers. This testing tool combines the ideas of probabilistic grammars and evolutionary algorithms to generate test cases that trigger defects and unwanted behavior. Furthermore, the *fuzzer* aims the to be language and grammar independent to appeal to a broader testing community.

The remainder of this thesis is structured as follows. First, the related work is discussed in Section 2. Then, in Section 3, some preliminaries are introduced and a brief background on the topics that from the basis of this thesis is given, introducing the ideas behind *fuzzing*, *evolutionary algorithms* and *(probabilistic) grammars*. In Section 4, the approach of building an evolutionary grammar-based fuzzing tool is presented, along with its implementation in Section 5. Section 6 shows the results of the evaluation, followed by a discussion of the presented approach (Section 7). Finally, this thesis is concluded in Section 8.

2 Related work

This section discusses the related work and the current *state-of-the-art* with respect to the goals of this thesis and specifically focuses on the currently available approaches for testing and finding vulnerabilities in parsers with grammar-based fuzzing techniques.

Fuzzing was originally introduced by a small research team by Miller et al. [6], systematically testing utility programs running on UNIX. By feeding a program randomly generated characters, Miller et al. were able to crash many of the tested utility programs. Researchers quickly adapted the idea of fuzzing and used it as a brute force tool to detect several security vulnerabilities. Since then, the technique has been used in a wide range of fields such as test case generation, protocol testing, file format testing or testing neural networks [7], [8].

Greybox fuzzing, a variation of traditional fuzzing, is considered to be the state of the art and most effective method for fault and vulnerability detection [2], [3]. A greybox fuzzer uses instrumentation to observe and determine whether an input improved a certain criteria, e.g code coverage. If the input yields new coverage or interesting improvement it will be mutated and modified and will be added again to the fuzzer's queue. Some of the already widely used fuzzing tools, *AFL* [2] and *libfuzzer* [3], have proven their capability of detecting unwanted behavior and discovered hundreds of high-impact security vulnerabilities. With *AFLSmart*, Pham et al. [9] introduced a coverage-based greybox fuzzer, which uses the ideas of smart greybox fuzzing, leveraging a high level structural representation of the input corpus to generate new input data. Pham et al. also introduced a novel power schedule, allowing the fuzzer to spend more time generating input data that are more likely to be valid and pass through the parsing stage.

Csmith, a language specific random test case generator by Yang et al. [10] is able to detect bugs in C compiler. *Csmith* derives valid C programs by randomly using production rules from the build-in C grammar. Yang et al. further introduced fixed semantic rules to the input generation, which allow or disallow certain production rules depending on the context, binding the fuzzer even further to the specific C language.

With *jsfunfuzz*, Ruddersman et al. [11] presented a language specific blackbox fuzzing tool for the JavaScript engine, which has shown to be very effective in detecting vulnerabilities and bugs, finding over 1000 defects in the Mozilla JavaScript interpreter Spidermonkey.

In 2012, Holler et al. [12] presented a language independent blackbox fuzzing tool called *LangFuzz*, which uses mutation to learn and reuse semantic context. Given a grammar and a collection of seed inputs, *LangFuzz* separates each input into fragments. Then, *LangFuzz* uses these fragments to generate new inputs by mutating fragments in a given seed. These fragments can either be deleted or substituted with similar fragments. This allows *LangFuzz* to preserve the semantic context and generate syntacticly and semantically valid input samples. *LangFuzz* is considered to be the state-of-the-art grammar-based fuzzing tool.

With *IFuzzer*, Veggalam et al. [13] introduced a genetic algorithm that uses code

fragments to iteratively build and extend individual test cases to test and target the JavaScript interpreter Spidermonkey. Similar to the approach described in this thesis, *IFuzzer* uses the properties of evolutionary algorithms to generate "fit" individuals. It uses a *cyclomatic complexity* metric to guide the evolutionary process. *IFuzzer* and the approach described in this thesis, mainly differ in their representations of individuals and their generation processes to derive test inputs. Additionally, *IFuzzer* is not language independent and only targets JavaScript interpreters, whereas this thesis aims to deploy a language independent framework to apply to a broader range of parsers and interpreters.

Grammar-based fuzzer usually can be distinguished into two groups, the generative and the mutation approach. The generative approach tires to randomly generate input from scratch, using certain constrains and rules. The mutation approach however, tries to mutate and modify existing inputs to derive new input data. *jsfunfuzz* and *Csmith* both use the generative approach, but due to the random test case generation often fail to produce syntactic preserving inputs. This leads to lot of time in the fuzzing campaign being wasted on the generation of syntactically invalid inputs. On the other hand, *LangFuzz* and *IFuzzer* use the mutation process to alter already valid input samples. The recently published *fuzzing survey* by Manès et al. [14], aims to precisely define what fuzzing is, to characterize various fuzzers and to systematically categorize fuzzers based on their features. Manès et al. also maintain a repository of an up-to-date genealogy database[1] of fuzzers and relevant papers.

Recently, Pavese et al. [5] presented an approach to generate test inputs using a grammar and a set of input seeds. By using the input seeds to obtain a probabilistic grammar, Pavese et al. were able to generate similar inputs to the initial inputs, or by inverting the probabilistic grammar, generate dissimilar inputs. Similar input samples can be very useful, for instance when learning from failure inducing samples, while dissimilar inputs can be very useful for testing less common, and therefore less evaluated parts of the program. This thesis will use the ideas on input generation presented by Pavese et al. and extends them with an evolutionary optimization approach. This input generation process builds the foundation of the evolutionary grammar-based fuzzer.

[1] https://github.com/SoftSec-KAIST/Fuzzing-Survey

3 Background

This section starts with setting the preliminaries, followed by the introduction of the topics that form the basis of this thesis, namely *fuzzing*, *evolutionary algorithms* and *(probabilistic) grammars*.

3.1 Preliminaries

The following terms are often used throughout this thesis and may have different connotations in other research papers. These are orientated on Holler et al. [12] and Pavese et al. [5].

Defect: This thesis focuses on defects and erroneous behavior, that will terminate or crash a parser or interpreter, for instance an assertion violation or a crash due to false memory allocation. All other bugs and defects creating false output will be dismissed. Since detecting false output often requires extensive knowledge about the program, analyzing seemingly correct output will therefore be out of the scope of this thesis.

Parser: This thesis aims to detect defects and unwanted behavior in parsers, which analyze the syntax of an input, conforming to the rules of a language. An interpreter is a software that receives structured inputs or programs and executes it. Most of these interpreters consist of lexer and parser stages, rejecting malformed inputs without even executing it. Since interpreters are often comprised of parsers, this thesis will, for reasons of simplicity, only refer to them as *parsers*. Nonetheless, the approaches described in this thesis can equally applied to interpreters.

Grammar: Throughout this thesis, the term grammar refers to context-free grammars. A further definition is given in Section 3.4.

3.2 Fuzzing

Recently, fuzzing has been recognized as one of the most effective methods for fault detection in computer programs. Fuzzing is a software testing technique to detect incorrect behavior or disfunction by randomizing the input to expose corner cases that have not been properly dealt with. Some of the already widely used fuzzing tools are AFL [2] and libfuzzer [3]. Modern fuzzer are able to generate input from scratch or use previous seeds to mutate or modify them. For instance, AFL mutates an input by flipping random bits, deleting or swapping values, and altering data. This method allows a wide range of applicability, not requiring a specific input model. Furthermore, a fuzzer can be considered a white-, grey-, or blackbox fuzzer, depending on the awareness of the program structure.

To further enhance the input generation, the mutation and modifying process can be guided into certain directions and corner cases, referred to as *coverage guided fuzzing*. A

fuzzer is considered to be more effective by achieving a higher degree of code coverage. By guiding the fuzzer into areas of the program that lack code coverage through testing, it increases the chance of finding bugs and unexpected behavior. Coverage guided fuzzing tools (CFG) usually consist of an input mutation generator, feedback guidance, and a fuzzing strategy. By recognizing interesting and vulnerable parts of the program, the feedback guidance provides adjustments to the fuzzing strategy and input generation, improving the effectiveness of the tool significantly [2].

A blackbox fuzzer, unaware of the internal structure of the program, can typically only apply random inputs to the program, only allowing to detect shallow or obvious errors. Nevertheless, since the computational time to modify the input can be omitted, the blackbox fuzzer is able to execute a lagre amount of inputs, making it very efficient. Opposing to that, a whitebox fuzzer uses the internal structure to progressively increase the code coverage. A whitebox fuzzer is therefore able to expose errors that would otherwise be hard to detect. However, the time of analyzing the program can be very time consuming. A greybox fuzzer combines both techniques and is only aware of the rough outline of the program structure. The fuzzer monitors and measures the level of the programs performance, allowing it to know if an input increased the code coverage.

3.3 Evolutionary algorithms

This thesis aims to construct an evolutionary grammar-based fuzzing tool, which makes use of the evolutionary optimization process. Evolutionary algorithms use the mechanics inspired by the biological evolution and try to solve problems by optimizing individual solutions, following Darwin's theory of evolution. Typical mechanics, also known as *genetic operations*, are *reproduction*, *recombination or crossover*, *mutation* and *natural selection* of individual solutions. By introducing a *fitness function*, the performance of the individual solutions can be analyzed and the quality of a solution can be determined. The best performing individuals of a population may then be selected to build the population of the next generation. By additionally applying the genetic operations, individual solutions can be further "evolved". This evolutionary process repeats until the best solution has been found. Although many variants of evolutionary algorithm exist, *genetic algorithm* being the most popular, most variants only differ in their representation of individuals. A commonly used procedure to derive a new population of individuals may be comprised of the following steps: (*i*) randomly select an initial starting population; (*ii*) analyze the fitness of each individual in the current population; (*iii*) select the *fittest* individuals for reproduction; (*iv*) breed a new population of offspring, created with mutation and crossover; (*v*) replace the old population with the new population; (*vi*) repeat steps *ii*-*v* until e.g. the optimal solution has been found.

$$
\begin{aligned}
&Expr \rightarrow Term \mid Expr \text{ “+“ } Term \\
&\quad \mid Expr \text{ “-“ } Term; \\
&Term \rightarrow \ Term \text{ “/“ } Factor \mid \ Term \text{ “*“ } Factor \\
&\quad \mid \ Factor; \\
&Factor \rightarrow \ \text{ “+“ } Factor \mid \ \text{ “-“ } Factor \\
&\quad \mid \ \text{ “(“ } Expr \text{ “)“ } \mid \ int; \\
&Int \rightarrow \ Digit \mid \ Digit \ Int; \\
&Digit \rightarrow \ \text{“0“} \mid \ \text{“1“} \mid \ \text{“2“} \mid \ \text{“3“} \mid \ \text{“4“} \mid \ \text{“5“} \mid \ \text{“6“} \\
&\quad \mid \ \text{“7“} \mid \ \text{“8“} \mid \ \text{“9“};
\end{aligned}
$$

Figure 1: Grammar G, which allows the production of arithmetic expressions.

3.4 Grammars

Context-free grammars are a well studied field of theoretical computer science, compiler design, and linguistics [15]. Context-free grammars are used to describe programming languages and can be used to automatically generate parser-programs in compilers.

A *context free grammar* is a *four tuple (N,T,P,S_0)*, where N is the set of *non-terminals*, T the set of *terminals*, P the set of *productions rules* with $P : N \rightarrow (N \cup T)$ and $S_0 \in N$ the initial starting symbol. Production rules are used to expand a *non-terminal* $S \in N$ to one of its n alternatives A_i.

$$S \rightarrow A_1 \mid A_2 \mid A_3 \mid ... \mid A_n \tag{1}$$

To demonstrate the usage of a *context-free grammar*, lets assume a simple example grammar, that allows the production of arithmetic expressions. This original grammar is taken from Pavese et al. [5] and will be further illustrated and extended in the following sections to demonstrate the mechanics of the evolutionary grammar-based fuzzer.

Figure 1 shows the production rules P of grammar G, with the *non-terminals* $N = \{Expr, Term, Factor, Int, Digit\}$, the *terminals* $T = \{0, 1, 2, .., 9, +, -, *, /, (,)\}$ and $S_0 = \{Expr\}$ as the starting symbol. By following the specified rules, this grammar can now be used to generate arithmetic expressions or can be used to verify if an input is part of the language that this grammar portrays.

A *probabilistic grammar* is a context-free grammar, augmented with probabilities p_i for each of the n alternatives expansions of a non-terminal S.

$$S \rightarrow p_1 A_1 \mid p_2 A_2 \mid p_3 A_3 \mid ... \mid p_n A_n \tag{2}$$

Exemplary for the grammar G, the production rule of the non-terminal *Expr* is

augmented with the following probabilities:

$$\begin{aligned} Expr \rightarrow\ & 45\%\ Term \mid 35\%\ Expr\ \text{“+“}\ Term \\ & \mid 20\%\ Expr\ \text{“-“}\ Term; \end{aligned} \tag{3}$$

If we use these probabilities to generate arithmetic expressions, 45% of the time the non-terminal $Expr$ would be expanded to $Term$, 35% of the time expanded to $Expr$ “+“ $Term$, and 20% of the time expanded to $Expr$ “-“ $Term$.

These probabilities can also be learned from a set of inputs, displaying the distribution of production rules required to derive these inputs. Pavese et al. [5] show that with this probabilistic grammar, similar input to those, from which the probabilities have been derived, can be generated. The procedure of learning a probabilistic grammar from a set of inputs and the generation of similar inputs according to that probabilistic grammar, builds the foundation of this thesis. This mechanic is used to drive the evolutionary process of generating a new population similar to the previous generation.

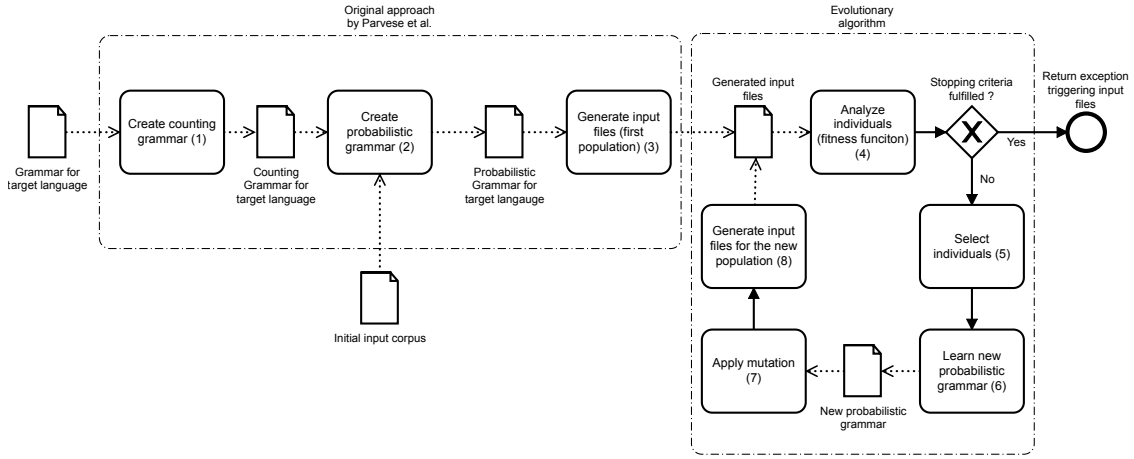


Figure 2: Overview of the approach extending the original approach by Pavese et al.

4 Approach

In this section, I will present my approach of building a language independent evolutionary grammar-based fuzzing tool to detect defects and unwanted behavior. I start with giving a brief overview of my evolutionary grammar-based approach, before going into more detail on the most important individual components.

4.1 Overview

The key concept of this research project is to build on the work of Pavese et al., described in their paper "Inputs from hell" [5], and to extend it to an evolutionary fuzzing tool. Figure 2 gives an overview of my approach extending the original approach by Pavese et al. The main idea is to use their ideas on test input generation and to combine it with an evolutionary optimization approach, to generate defect inducing test cases.

The evolutionary fuzzing tool starts, like the original approach, with a grammar of the target language. The first step is to obtain a counting grammar that allows us to measure how frequently individual alternatives of a production rule occur during parsing in each production context *(Activity (1))*. With the language specific counting grammar and an initial input corpus, the appropriate probabilities are determined for the probabilistic grammar *(Activity (2))*. Using this derived probabilistic grammar to generate the first population of inputs, marks the beginning of the evolutionary process *(Activity (3))*. The generated inputs are, as described by Pavese et al., either very similar or, by inverting the probabilities of the probabilistic grammar, very dissimilar to the initial input corpus.

After the first population of input files has been generated, the *fuzzer* will now iteratively build a new population, by learning a new probabilistic grammar from the most interesting individuals or from those that yielded the most improvement of a certain criteria. This process continues until a certain stopping criterion is fulfilled. As already implied, by retaining interesting individuals and by "learning" their *features*,

the *fuzzer* is combined with an evolutionary optimization approach. Evolutionary optimization strategies and approaches usually try to solve a problem by "evolving" already existing, but not perfect solutions. Analyzing the performance of individuals *(Activity (4))* allows us to rank them according to a metric *(fitness)*. The best performing and most interesting individuals will be selected *(Activity (5))* and are used to learn the new probabilistic grammar, distributing the probabilities of the production context of the best individuals *(Activity (6))*. This grammar is then "evolved" by mutation, mimicking the natural evolution process even further *(Activity (7))*. In *Activity (8)*, a new generation of individuals will be generated, building up the population for the next iteration. This iteration cycle may stop when: (i) it computes n generations, where n is the maximum number of generations that is allowed; or (ii) no new defects were found after x generations. Overall, my approach aims to find as many defects and errors in a targeted parser or interpreter, by generating inputs to trigger unwanted behavior.

In the following sections, I will explain my approach in more detail, focusing on the *Input generation*, the *Fitness function*, the *Selection strategy* and the *Mutation* to evolve the grammar. To further illustrate my approach, I will explain individual components and processes exemplary with the grammar G shown in Section 3.4.

4.2 Generation and learning of individuals

In this section, I will focus on the *initial input corpus*, the *learning of the probabilistic grammars* and the *generation of a new population*, represented in Fig. 2 by the *Activities (2),(3),(6) and (8)*. Since these activities are already developed and covered by Pavese et al., I will focus on how these components influence my approach as well as highlight problems that arise from them.

4.2.1 Initial input corpus

After constructing the counting grammar to the corresponding target grammar, the first probabilistic grammar is learned from the initial input corpus *(Activity (2))*. This initial input corpus contains a couple of preselected input files from which the probabilistic grammar should be learned. Since the *fuzzer* generates, with the help of the probabilistic grammar, similar inputs and individuals, the selection of the initial input corpus may strongly influence the performance and the outcome of the algorithm. On the one hand, if the initial input corpus is already comprised of input files containing interesting behavior, it is likely that this behavior will also show early on in the first populations. On the other hand, if the initial input corpus is comprised of inputs that only cover a small section of all possible production rules of the grammar, it may take an unnecessary long time and many iterations until the rest of the grammar is explored and discovered.

Defect and error inducing inputs can therefore boost the performance of the algorithm early on and should be selected carefully [5]. Nevertheless, the specification of an initial corpus is not necessarily required, if there is no initial input corpus available or if the parser should be tested without any influence of initial input files. Without an initial

$$
\begin{aligned}
&Expr \rightarrow 66.7\%\ Term \mid 33.3\%\ Expr + Term \\
&\qquad \mid 0\%\ Expr \text{ - } Term; \\
&Term \rightarrow 0\%\ Term \ /\ Factor \mid 25\%\ Term * Factor \\
&\qquad \mid 75\%\ Factor; \\
&Factor \rightarrow 0\%\ \ + Factor \mid 0\%\ \ \text{- } Factor \\
&\qquad \mid 25\%\ \ (\ Expr\ )\ \mid 75\%\ int; \\
&Int \rightarrow 100\%\ Digit \mid 0\%\ Digit\ Int; \\
&Digit \rightarrow 0\%\ 0 \mid 33.3\%\ 1 \mid 33.3\%\ 2 \mid 33.3\%\ 3 \\
&\qquad \mid 0\%\ 4 \mid 0\%\ 5 \mid 0\%\ 6 \\
&\qquad \mid 0\%\ 7 \mid 0\%\ 8 \mid 0\%\ 9;
\end{aligned}
$$

Figure 3: First probabilistic grammar G^1_{prob} learned from the input $I = 1 + (2 * 3)$

corpus, the tool simply assumes a normal distribution among all production rules and its alternative expansions and generates the first initial input corpus accordingly. This may not be too effective, but allows the *fuzzer* to function without the specification of an initial input corpus.

Let's consider the grammar G from Section 3.4 *(see page 10)* and the initial input $I = 1 + (2 * 3)$. To obtain the first probabilistic grammar G^1_{prob}, we need to use the grammar G to parse input I and determine the distribution of the taken expansions. Figure 3 shows the derived probabilistic grammar G^1_{prob} learned from the input I, which will be used to generate the first population. Notice that, since input I doesn't contain any substitution and division, the probabilities for the expansions of the *non-terminals* $Expr$ and $Term$ to ($Expr$ "-" $Term$) and ($Term$ "/" $Factor$) respectively, are set to 0%. A further explanation of this example can be found in this paper [5]. This example clearly shows the importance of choosing a "good" initial input corpus. Iteratively learning from this probabilistic grammar and generating new inputs accordingly, we are, without the usage of any other methods, like *mutation*, not able to explore the full potential of the grammar G.

4.2.2 Input generation

After the first probabilistic grammar has been derived, the first population of individuals is generated *(Activity(3))*. To generate the first input files, I use the method and tool provided by Pavese et al. [5]. Their described tool is able to generate input files according to a probabilistic grammar. Figure 4 shows the input files generated according to the probabilistic grammar G^1_{prob}, which only contain the same digits and operators as the initial input I. The generation process of these input files is rather simple, since it reduces to traversing the grammar according to the probabilities. However, this procedure runs the risk of probabilistically choosing production rules that lead to an excessively large input file, or by triggering a recursion with no base case, running the

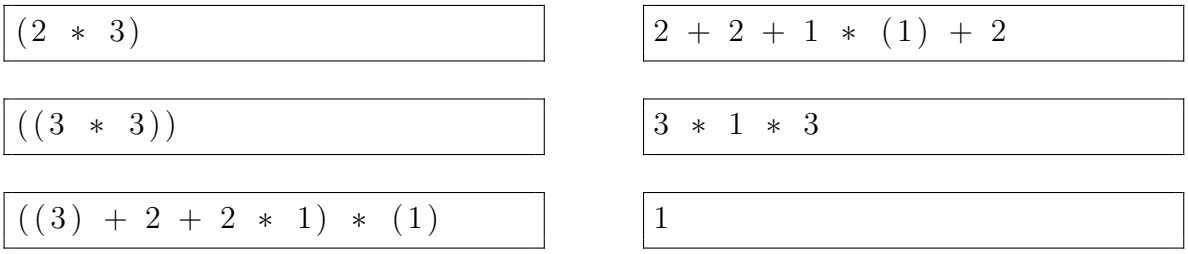


Figure 4: Inputs files generated according to the probabilistic grammar G^1_{prob} shown in Fig. 3.

risk of never terminating. To resolve this issue, Pavese et al. suggest setting a threshold, which when exceeded, production rules are no longer chosen probabilistically, but rather constrained to those expansions that generate the shortest possible expansion tree. This idea ensures both the termination of the generation procedure, as well as trying to keep the input file size close to the threshold parameter. However, restricting the *derivation tree* to a threshold can be a major problem for certain grammars. Resulting inputs often contain only single letters or worse, are totally empty.

After the first population of individuals has been generated, the *fuzzer* will iteratively learn a new probabilistic grammar and will build a new population accordingly *(Activities (6) and (8))*. The process of learning the probabilistic grammar from a set of inputs and the generation of inputs according to a probabilistic grammar will stay the same throughout all iterations.

4.3 Analysis of individuals

In this Section, I will focus on the analysis of the individuals, represented in Fig 2 by the *Activity (4)*. I will motivate the necessity of a metric guiding the iteration process and will present three in "complexity" increasing fitness functions that build on each other.

4.3.1 Motivation: Genetic drift

One of the properties of evolutionary algorithms and approaches is the genetic drift, also known as the Sewall Wright effect [16]. It describes the change of frequencies of an existing *gene*, in our case the frequencies and distribution of alternatives of a production rule, in an entire population due to random selection of individuals. On the one hand, genetic drift may cause gene variants to disappear completely and thereby reduce genetic variation, on the other hand, genetic drift may also cause initially rare *genes* to become more frequent and even fixed. This behavior can be directly observed in my approach. Consider the probabilistic grammar G^1_{prob} previously shown in Figure 3 and the generated inputs shown in Figure 4. Now, randomly selecting a subset s of these inputs and learning a new probabilistic grammar G^2_{prob} for the second iteration will result in a grammar with a slightly different distribution of the probabilities.

Figure 5 shows that, for instance, the production rule of $Expr$ has changed, raising the likelihood of expanding $Expr$ to the alternative $Term$. This happens because we initially started with a high probability for the expansion $Term$. Randomly selecting the subset s resulted in the raise of the expansion $Term$ and the repression of the other alternatives. Continuing the process of generating inputs according to the probabilistic grammar and iteratively learning a new probabilistic grammar G^n_{prob} from a subset s, resulted, after only nine generations, in the complete repression of all the other

$$
\begin{aligned}
&Expr \rightarrow \textbf{73\%}\ Term \mid \textbf{27\%}\ Expr + Term \\
&\qquad \mid 0\%\ Expr \text{ - } Term; \\
&Term \rightarrow 0\%\ Term\ /\ Factor \mid \textbf{10\%}\ Term * Factor \\
&\qquad \mid \textbf{90\%}\ Factor; \\
&Factor \rightarrow 0\%\ \ + Factor \mid 0\%\ \ \text{-}\ Factor \\
&\qquad \mid \textbf{13\%}\ \ (\ Expr\ ) \mid \textbf{87\%}\ Int; \\
&Int \rightarrow 100\%\ Digit \mid 0\%\ Digit\ Int; \\
&Digit \rightarrow 0\%\ 0 \mid 33.3\%\ 1 \mid 33.3\%\ 2 \mid 33.3\%\ 3 \\
&\qquad \mid 0\%\ 4 \mid 0\%\ 5 \mid 0\%\ 6 \\
&\qquad \mid 0\%\ 7 \mid 0\%\ 8 \mid 0\%\ 9;
\end{aligned}
$$

Figure 5: Probabilistic grammar G^2_{prob} learned from a randomly selected subset s of inputs, shown in Fig. 4.

$$
\begin{aligned}
&Expr \rightarrow \mathbf{100\%}\ Term \mid \mathbf{0\%}\ Expr + Term \\
&\quad \mid 0\%\ Expr \text{ - } Term; \\
&Term \rightarrow 0\%\ Term\ /\ Factor \mid \mathbf{0\%}\ Term * Factor \\
&\quad \mid \mathbf{100\%}\ Factor; \\
&Factor \rightarrow 0\%\ \ + Factor \mid 0\%\ \ \text{-}\ Factor \\
&\quad \mid \mathbf{0\%}\ \ (\ Expr\ )\ \mid \mathbf{100\%}\ Int; \\
&Int \rightarrow 100\%\ Digit \mid 0\%\ Digit\ Int; \\
&Digit \rightarrow 0\%\ 0 \mid 33.3\%\ 1 \mid 33.3\%\ 2 \mid 33.3\%\ 3 \\
&\quad \mid 0\%\ 4 \mid 0\%\ 5 \mid 0\%\ 6 \\
&\quad \mid 0\%\ 7 \mid 0\%\ 8 \mid 0\%\ 9;
\end{aligned}
$$

Figure 6: Probabilistic grammar G^9_{prob} after iteratively learning a new probabilistic grammar from a randomly selected subset s.

expansions. Figure 6 shows the resulting grammar G^9_{prob}. Since the expansion of $Expr$ to the *non-termial* $Term$ and the further expansion to $Factor$ is used to generate a digit, the generated individuals, according to the grammar G^9_{prob}, will result in inputs only containing a single digit. This behavior can be observed with most grammars and their input generation. However, since I am aiming to create "complex" and "interesting" input files and test cases, a method is needed to counteract the effects of the genetic drift. This example clearly illustrates the necessity of a metric to guide the learning process, specifically the necessity of a fitness function.

4.3.2 Naive fitness function *(file length)*

To detect defects and unwanted behavior, I am particularly interested in "complex" and "interesting" individuals. However, as the example in Section 4.3.1 shows, an additional method is needed to counteract the effects of the genetic drift. In my approach I use a fitness function to analyze the *performance* of the individuals, based on which they are selected for the next learning and iteration process. Since the fitness function directly influences future generations, it is considered to be one of the most important components of an evolutionary algorithm, guiding the process of my approach towards the goal. Analyzing the individuals with a fitness function allows us to rank the performance of all individuals. The *fitter* an individual the higher are its chances of *surviving* and getting selected for the next iteration *(survival of the fittest)*. With the goal of detecting defects and unwanted behavior, I am specifically focusing on structural aspects of the input files. For instance, complex structures, like nested loops, have a higher tendency to create uncommon behavior, thus, in order to reach the goal, making complex inputs more interesting than less complex inputs. Finding a suitable fitness function, ranking and selecting the individual inputs for the next iteration is therefore vital to the effectiveness of my approach. Since the target language is given as input,

I am only able to make little assumptions about the complexity of input files in the context of the grammar. These assumptions rely on the *file length*, the *number of used expansions* to derive an input file, or on the structure of a *derivation or parse tree*.

Similar to *IFuzzer*, developed by Veggalam et al. [13], my fitness function can be separated into two components: the feedback score and the structural score of an input file x. The feedback score describes the feedback from the parser when executing an input, including execution timeouts, errors and crashes. The structural score describes the structural aspects of the input file, favoring "complex" inputs. The fitness of an individual x can therefore be calculated as the sum of its feedback score and its structural score. The goal is to maximize the fitness for each individual.

$$fitness(x) = score_{feedback}(x) + score_{structure}(x) \tag{4}$$

The calculation of the feedback score is rather simple, since it reduces to executing the input x and observing if any exceptions were triggered, whereas the calculation of the structural score is much more difficult. In the following sections, I will present three in "complexity" increasing approaches that build on each other in order to rank the structure of an input.

File length

With the intension of detecting defects and exploring the grammar to trigger unwanted behavior of a parser, the aim is to use the grammar and its production rules to create *effective* test inputs. But, as described in Section 4.3.1, random input selection resulted for almost all grammars, due to the genetic drift, in the repression and loss of most (*interesting*) expansions. To counteract this effect and to resolve the issue of the decreasing variety of selectable expansions and the slowly decreasing input length, the simplest idea for the structural score is to measure the fitness of an individual x according to its input length.

$$score_{structure}(x) = length(x) \tag{5}$$

Comparing individuals according to their file length, will rank individuals with a greater file length higher than individuals only containing a single or fewer digit(s). The intension is that with more produced digits, the number of chosen productions increases as well, keeping more productions and more selectable expansions in the grammar.

To show the effectiveness of the fitness function using this structure score, consider the grammar G^1_{prob} (Fig 3, page 14) and the generated input files (Fig 4, page 15). If we now iteratively select the longest input files to learn a new probabilistic grammar and generate a new population accordingly, we observe that the expansion from the *non-terminal* $Expr \rightarrow Term$ does not increase. We rather observe that the probability for selecting Term is actually decreasing, since directly choosing this expansion will result in shorter input files. With this structural score the fitness function favors

$Expr \rightarrow 66.7\%\ Term \mid 33.3\%\ Expr + Term$
$\quad \mid 0\%\ Expr$ - $Term$;
$Term \rightarrow 0\%\ Term\ /\ Factor \mid 25\%\ Term * Factor$
$\quad \mid 75\%\ Factor$;
$Factor \rightarrow 0\%\ \ + Factor \mid 0\%\ \ - Factor$
$\quad \mid 25\%\ \ (\ Expr\ )\ \mid 75\%\ int$;
$Int \rightarrow 66.7\%\ Digit \mid 0\%\ Digit\ Int$;
$\quad \mid 33.3\%$ **Boolean**
Boolean $\rightarrow 33.3\%$ "$true$" | 33.3% "$false$"
$\quad \mid 33.3\%$ "$null$";
$Digit \rightarrow 0\%\ 0 \mid 33.3\%\ 1 \mid 33.3\%\ 2 \mid 33.3\%\ 3$
$\quad \mid 0\%\ 4 \mid 0\%\ 5 \mid 0\%\ 6$
$\quad \mid 0\%\ 7 \mid 0\%\ 8 \mid 0\%\ 9$;

Figure 7: Extended probabilistic grammar G_{extend}.

expansions that can potentially achieve a longer file, directly countering the effects of the genetic drift.

Nonetheless, if the language, like *JSON* or *JavaScript*, contains words of fixed length, for instance boolean expressions like "true", "false" or "null", which contribute to the file length much more than single letters, the fitness function will eventually favor the expansions that result in these words. This often leads to input files only containing the longest words, decreasing the variety and the selectable expansions once more. To further illustrate this behavior, let's again consider the grammar G^1_{prob} (Fig 3, page 14), but this time, we extend it with an additional production rule $Boolean \rightarrow$ "$true$" | "$false$" | "$null$". Additionally, we add the *non-terminal Boolean* as a selectable expansion alternative of the *non-terminal Int*. Figure 7 shows the constructed grammar G_{extend} enriched with exemplary probabilities. If we now run the process of iteratively generating input files, learning a new probabilistic grammar and selecting the individuals according to their file length, the generated inputs will eventually look like the following:

```
false + false + (false)
```

```
false
```

```
false * (false+ false)
```

```
false * (false + true)
```

```
false + false + false
```

```
false * false
```

Figure 8: Inputs generated after ten iterations with the probabilistic grammar G_{extend} and the *naive fitness function (file length)*.

After only ten generations, all generated input files in Figure 8 mainly consist of the boolean expression $false$. This behavior occurs because an expansion of the *non-terminal Int* to one of the boolean expressions, in particular to the boolean expression $false$, results in longer files than expanding it to a single digit.

Ranking input files according to their file length works very well with simple grammars and languages where every production rule and their expansions can equally contribute to the length of the file. With the help of this structure score, the fitness function is also able to counteract the effects of genetic drift, keeping a variety of different operators in the grammar. However, for many grammars and languages, this structure score is not sufficient in order to generate "complex" and "interesting" individuals.

4.3.3 Improved fitness function *(Number of expansions)*

To achieve a higher variety and to rank input files without the biased assumption on their file length, I chose to rank the input files according to the number of used productions to derive an input. This again follows the assumption that with more used productions and more taken expansions the input file is more likely to trigger uncommon behavior. Without the need to produce output or code, expansions are favored not for the capability to create long files but rather favored for their capability of producing more expansions. Nevertheless, evaluation and testing have shown that it is still useful if the generated input also contains a minimum amount of generated output or code. To not restrict the number of expansions too much, I use a ratio describing the number of expansions to the file length.

With this improved structure score I want to punish expansions that produce long words and favor inputs that had many letters produced by many expansions, increasing the overall amount of used production rules. In order to derive the improved structure score for each individual, the number of total expansions μ, the length of the file and a predefined constant λ are needed. The constant λ functions as a ratio parameter which can differ from language to language. It is needed to determine and to adjust the expected ratio of the number of expansions to the length of the file. The improved structure score for an individual x is calculated with the following formula given by (6) and (7):

$$ratio(x) = \frac{\mu(x)}{\lambda \times length(x)} \tag{6}$$

$$score_{structure}(x) = \mu(x) \times ratio(x) \tag{7}$$

We start with determining the ratio of the used expansions to the file length in Equation 6. Then, in Equation 7 the ratio is multiplied with total number of used expansions. This way, if the ratio is not sufficient, the number of expansions, with which I am measuring the structure of an individual, is decreased. If the ratio is "good", the structure score of an individual x is increased.

```
3 + null + 2 + (1)
```

```
(false) + 0 + (2 * 3) +2
```

```
false * (3 + true)
```

```
false * (null + 1)
```

```
3 + 1 + 1
```

```
2 + true + (null) + true
```

Figure 9: Inputs generated after ten iterations with the probabilistic grammar G_{extend} and the *improved fitness function*.

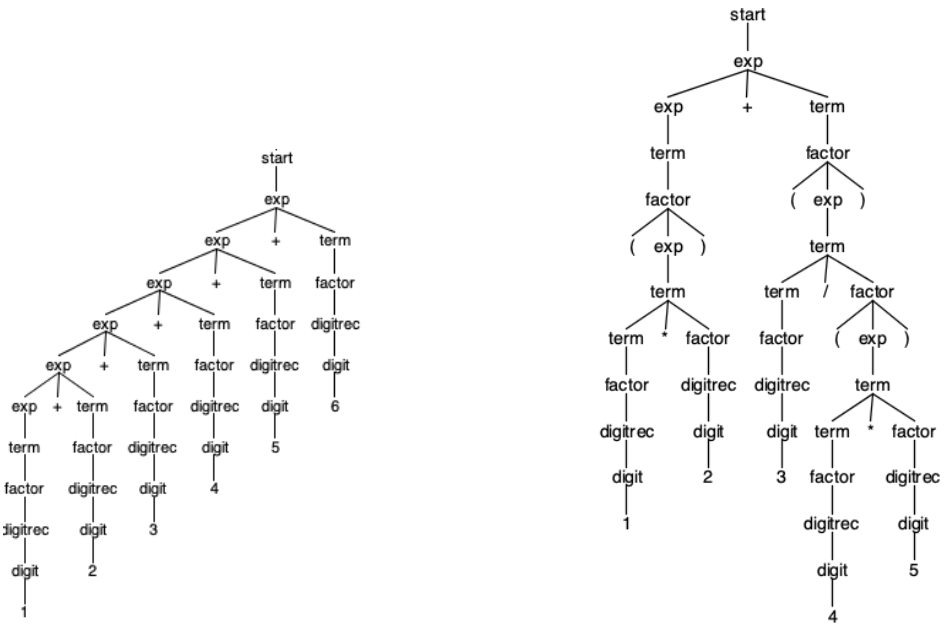
With this *improved* structure score the fitness function from Equation 4 will rank individuals higher which have more expansions as well as individuals that have a better expansion - length ratio. Figure 9 shows the input files generated after ten iterations, with the evolutionary process iteratively selecting the fittest individuals according to the improved structure score. These input files are much more diverse than those generated previously (Fig. 8).

Using this *improved* structure score as the structural component of the fitness function (Eq. 4), resulted in diverse and "interesting" input files. Further, evaluation and testing confirm the effectiveness of this fitness function, successfully guiding the evolutionary process towards defects and unwanted behavior. However, with the guidance of this fitness function, generated input files often lacked "complexity". Particularly for the language *JavaScript*, the fuzzer was only partially able to generate complex input files that contained *nested loops* or *iteration statements*. Although the *improved fitness function* already proved itself as a sufficient metric, I further investigated the usage of a slightly more sophisticated structure score.

4.3.4 Sophisticated fitness function *(Derivation tree)*

The goal of my approach is to generate uncommon code structures to detect defects and trigger exceptions. The *improved fitness function*, defined in Section 4.3.3, already considers the number of total productions and expansions needed and preserves a specific ratio of the number of expansions to the length of a file. Nevertheless, this metric lacks the ability to make any assumptions on the complex structure of the input file. The improved structure score considers every taken expansion equally, since any produced expansion can contribute equally the overall score. But, since I thrive to create complex structures, a different metric is needed. Although I am not able to make any assumptions about the grammar or language, the derived *derivation tree*, also called *parse tree*, of an input file can hint at complex structures and can be an indication for complex input files.

To illustrate this idea, let's assume the two inputs $I = 1 + 2 + 3 + 4 + 5 + 6$ and $J = (1 * 2) + (3 * (4/5))$. Both inputs have an equal amount of produced digits, and have a similar amount of used production rules and expansions. But, input J shows,



(a) Derivation tree of $1+2+3+4+5+6$. (b) Derivation tree of $(1*2)+(3/(4*5))$.

Figure 10: Two parse tress of different complexity.

with its nested terms, a much more complex behavior than the simple addition and sum of the digits from 1 to 6, and is therefore more desirable than input I. If we now compare the two derived derivation trees, shown in Figure 10, we can clearly identify a difference. In the derivation tree induced by the input I, we can see the concatenation of long sequences leading from either $Expr$ or $Term$ to a digit. In the derivation tree of input J we observe the same long sequences in the lower part of the tree. But as we can see, *non-terminals* like $Factor$ or $Expr$, which are responsible for the nested terms and fan out to many other *non terminals*, occur much deeper inside the tree.

Summarizing, I am looking for expansions that have a high fan out to other expansions, and occur deep inside the derivation tree. To give more weight to production rules that expand to more complex structures the fuzzer will use a derivation tree τ to calculate the structural score of an individual. Since a node inside the derivation tree describes a *non terminal* and the path from one node to another describes an expansion, the structural fitness score for τ can be derived by calculating the degree of each node v inside the tree. To give expansions with a high fan out on deeper levels more weight, the degree is further raised to the power of the level h at which the *non-terminal* occurred.

$$p(v) = deg(v)^h \tag{8}$$

$$score_{structure}(x) = \sum_{\forall v \in V} p(v) \tag{9}$$

After the derivation tree τ of an individual x has been derived, we calculate for all nodes v inside τ the value of $p(v)$ (Eq. 8). The values for all $p(v)$ sum up to the

structure score of an individual x (Eq. 9).

If we now compare this structure score to the previously defined structure score from Section 4.3.3, we see that previously both inputs I and J have been ranked similar. But with the new metric, we calculate a $score_{structure}$ for I of $10,591$ and a $score_{structure}$ for J of $96,347$, clearly separating them from each other and ranking J much higher than I. With this sophisticated structure score as the structural component of the fitness function (Eq. 4), the evolutionary process is able to generate interesting input files and is more likely to bring forth "complex" individuals.

Summarizing this subsection, I discussed the necessity of a metric to guide the evolutionary process along with presenting three metrics to analyze individuals. Without a suitable fitness function, the genetic drift is responsible for the repression of almost all selectable alternatives of a production rule. Since I am aiming to find defects and trigger unwanted behavior, I am specifically interested in "complex" individuals. All three metrics have shown to successfully counteract the effects of genetic drift, the latter also able to create most complex individuals. Due to its capability of generating more complex individuals, I am using the fitness function with the sophisticated structure score as the metric to analyze the individuals.

4.4 Selection strategy

The selection strategy, represented in Fig. 2 (see page 12) by the *Activity (5)*, is essential to the performance of evolutionary algorithms and specifies which of the analyzed individuals will be selected for the next generation. The approach of learning a probabilistic grammar and the resulting generation of offspring is quite different, in terms of the representation of individuals, compared to typical evolutionary algorithms. However, many established selection strategies can still be applied.

Individuals that have been selected will build the base for the next generation. As these individuals are the ones whose genes are inherited by the next generation, it is desirable that the selected subset is comprised of *good* individuals. A selection process is simply a mechanism that favors better performing individuals. The *selection pressure* [17] is the degree to which the better individuals are favored: the higher the selection pressure, the more the better individuals are favored. This selection pressure drives the evolutionary algorithm and largely determines the convergence rate, with a higher selection pressure resulting in a higher convergence rate. However, if the selection pressure is too high, there is an increased chance of the approach prematurely converging to an incorrect solution. If the selection pressure is too low, the algorithm might take unnecessarily long to find *good* solutions.

To provide a balanced selection pressure, the fuzzer uses elitism [18] and tournament selection [17]. Elitism means that a percentage of the fittest individuals in the current generation is copied directly into the next generation. Tournament selection is a method of selecting an individual from a population of individuals. It involves running several *tournaments* among s randomly selected competitors, with s being the tournament size. The winner of each tournament is the individual with the highest fitness and is selected for the next iteration.

After all individuals for the next generation have been selected, a new probabilistic grammar is learned. An individual's influence and contribution is therefore limited by the total number of selected individuals. This does introduce a bias that will be discussed in Section 7.

4.5 Grammar evolution

Genetic drift may cause gene variants to disappear completely and thereby reduce genetic variation. To maintain a genetic diversity from one generation to the next, I will additionally apply mutation, analogous to biological mutation, represented in Figure 2 by *Activity (7) (see page 12)*. The purpose of mutation is to introduce diversity into the population and to avoid local optima to explore other sections of the environment or the *search space*. It also allows the algorithm to discover features that have previously not been explored and allows a population to recover from being stuck in a corner case.

Differently compared to typical evolutionary algorithms the fuzzer does not apply the mutation directly to the individuals, but rather applies it to the learned probabilistic grammar. The nature of this approach and the idea of learning and generating inputs iteratively already implies an indirect mutation process, since the generated offspring

only follow probabilistic rules, making other genetic operators like *crossover* obsolete. Generating a new population of individuals from the previously learned grammar ensures that the next generation of individuals is similar, but it will never be identical to the previous generation. To allow the exploration of features and genes that have previously not been explored or have been lost during the evolutionary process, an additional mutation function is needed. Since my approach differs compared to other evolutionary algorithms, regarding the representation of individuals, I am not able to apply typical mutation operators, like *bit flipping*. However, mutating the probabilistic grammar allows to alter the probabilities of individual production rules. To apply the mutation to a probabilistic grammar, a random production rule is chosen and then the probabilities p_i of the individual alternatives A_i are recalculated. To ensure that $\sum_{i=1}^{n} p_i = 1$ still holds the newly derived probabilities are normalized. The formula for recalculating the probabilities can be described as follows: After a production rule with n different alternatives has been selected, the probabilities p_i for each alternative A_i are recalculated by selecting a random number r_i between 0 (exclusive) and 1 (inclusive) and normalizing it with the sum of all n random numbers.

$$p_i = \frac{r_i}{\sum_{i=1}^{n} r_i} \tag{10}$$

Figure 11 shows the mutation of the grammar G^3_{extend}. The mutation is applied to the production rule of the *non-terminal* $Term$, changing the probabilities for expanding $Term$ to its alternatives. The rate μ at which mutation occurs is adjustable, but should be kept rather low. Otherwise the generation of the new population will be too random.

$Expr \rightarrow$ 66.7% $Term$ | 33.3% $Expr$ + $Term$	$Expr \rightarrow$ 66.7% $Term$ | 33.3% $Expr$ + $Term$
| 0% $Expr$ - $Term$;	| 0% $Expr$ - $Term$;
Term $\rightarrow$ **0%** $Term$ / $Factor$ |	**Term** $\rightarrow$ **32%** $Term$ / $Factor$ |
25% $Term$ * $Factor$ | **75%** $Factor$;	**24%** $Term$ * $Factor$ | **44%** $Factor$;
$Factor \rightarrow$ 0% + $Factor$ | 0% - $Factor$	$Factor \rightarrow$ 0% + $Factor$ | 0% - $Factor$
| 25% ($Expr$) | 75% int;	| 25% ($Expr$) | 75% int;
$Int \rightarrow$ 66.7% $Digit$ | 0% $Digit$ Int;	$Int \rightarrow$ 66.7% $Digit$ | 0% $Digit$ Int;
| 33.3% $Boolean$	| 33.3% $Boolean$
$Boolean \rightarrow$ 33.3% "$true$" | 33.3% "$false$"	$Boolean \rightarrow$ 33.3% "$true$" | 33.3% "$false$"
| 33.3% "$null$";	| 33.3% "$null$";
$Digit \rightarrow$ 0% 0 | 33.3% 1 | 33.3% 2 | 33.3% 3	$Digit \rightarrow$ 0% 0 | 33.3% 1 | 33.3% 2 | 33.3% 3
| 0% 4 | 0% 5 | 0% 6	| 0% 4 | 0% 5 | 0% 6
| 0% 7 | 0% 8 | 0% 9;	| 0% 7 | 0% 8 | 0% 9;

Figure 11: Grammar G^3_{extend} before and after the mutation of the production rule of $Term$.

In this section, I presented my approach of building an evolutionary grammar-based fuzzing tool to detect defects in parsers. I started with giving an overview of the evolutionary process extending the original approach by Pavese et al.[5], before explaining the individual components in more detail. I covered the *generation and the learning of input files*, around which the evolutionary process is built. In order to effectively generate interesting and complex individuals, I presented three fitness functions to *analyze individuals* and to guide the evolutionary process. Additionally, I discussed the *selection strategy*, according to which the best performing individuals are selected for the next iteration. Lastly, I stated the necessity of a *mutation function* that allows for the further exploration of the target grammar. In the next sections, I will demonstrate and evaluate my approach.

5 Implementation

I realized the evolutionary grammar-based fuzzer as a *proof-of-concept* based on the process and methods described in Section 4. The fuzzer works as described in the overview diagram (Fig. 2, page 12) and requires a context free grammar and an initial input corpus. This input corpus can be of any size, but should be, as mentioned in Section 4.2.1, comprised of "good" inputs covering a fair amount of the in the context free grammar specified production rules. Additionally, I will add a *base-file*, only composed of the alphabet (a-z), to each generation, in order to keep a minimum set of the letters in the probabilistic grammar. To learn each probabilistic grammar and to generate new individuals, I use the tool provided by Pavese et al., which makes use of the well-known parser generator framework ANTLR [19]. Further, the fitness function, described in Section 4.3.4, makes use of the derivation tree returned by the parser generated by the ANTLR parser generator. After the first grammar has been learned and the first population is generated, the tool iteratively learns a new probabilistic grammar and generates a new generation accordingly. Like most evolutionary algorithms, many individual parameters can be adjusted. The following is a non-exhaustive list of parameters:

- number of generations n
- population size p
- threshold, to limit the input generation t (described in Section 4.2.2)
- mutation rate μ
- elitism e
- number of tournaments t_n
- size of tournaments t_{size}

For my evaluation I use the best combination based on observations, made during a fixed profiling period, but further fine tuning of all these parameters should be possible.

6 Experimental evaluation

In this section, I evaluate the effectiveness of my approach by performing experimentation on real-world applications. I compare my approach to the original approach by Pavese et al [5] and ask the following research questions:

- **RQ1** Can my approach achieve a higher code coverage than the original approach?
- **RQ2** Can my approach trigger more exception types than the original approach?

To answer these questions, I conducted an empirical study to compare my approach to the original input generation tool. All the experiments were performed on a standalone machine with a configuration of Quad-Core 3.4 Ghz Intel i5 CPU and 8 GB RAM.

6.1 Evaluation setup

6.1.1 Code coverage metric

To evaluate and compare my approach to the original approach I analyzed the achieved coverage. Code coverage [20] is a metric counting the unique lines of code of the targeted parser that have been executed during a test. This metric is a commonly used indicator to measure the effectiveness of a test as well as being an indicator for well tested code. The idea behind code coverage is that the higher the coverage and therefore the number of executed lines of code the more trustworthy the code becomes. A program with a high code coverage, measured as a percentage, has had more of its source code executed during testing, which suggests it has a lower chance of containing undetected software bugs compared to a program with low code coverage. Although several variants, like *branch coverage* or *instruction coverage* [21], exist, I used the line coverage metric to answer **RQ1**.

6.1.2 Subjects

To examine the effectiveness of my approach, different programming languages and different parsers are tested, since the performance of both approaches can highly depend on the complexity of the language and the complexity of the parser. A complex language and its grammar usually result in more complex parsers and interpreters, which are naturally more prone to defects and more difficult to test. To counteract this fact, I am focusing on three, in complexity varying grammars and their parser, namely *JSON*, *JavaScript* and *CSS3*. The parsers for these grammars are widely used in browsers and web applications. The subjects along with their grammars are listed in Table 1.

6.1.3 Research protocol

To test my approach and to compare it to the original approach I generated multiple test runs. For each subject both approaches run ten experiments, each containing 10,000 generated input files. The initial input corpus was comprised of randomly selected input

Input language	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
	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
JavaScript	Rhino	1.7.7	4873	100,234
CSS3	cssValidator	1.0.4	7774	120,838

Table 1: Test subjects

files crawled from Github. For each of the test runs, my approach ran for 100 generations with each population containing 100 individuals, adding up to 10,000 generated input files. The elitism rate was set to 5% and for each generation ten tournaments of size ten were held. The mutation rate was set to 1.0[2]. Over all ten test runs, 100,000 inputs were generated per subject. For the original approach a probabilistic grammar was learned from the initial input corpus and for each test run 10,000 input files were generated. Over all ten test runs, 100,000 inputs were generated per subject.

[2]This number indicates that one production rule per iteration is mutated.

6.2 Experimental results

Figure 12 to Figure 21 show a representative selection of the results. For each subject, I constructed a chart representing the line coverage achieved by my approach compared to the original approach. The horizontal axis represents the achieved line coverage and the vertical axis represents the increasing number of generated input files.

RQ1: Can my approach achieve a higher code coverage than the original approach?

To answer **RQ1**, I compare the executed lines covered by both approaches. In particular, I am investigating if my approach achieves at least the same percentage of line coverage than the original approach. Figure 12 to Figure 19 show the JSON parsers and Figure 20 and Figure 21 show the JavaScript and CSS3 parser respectively.

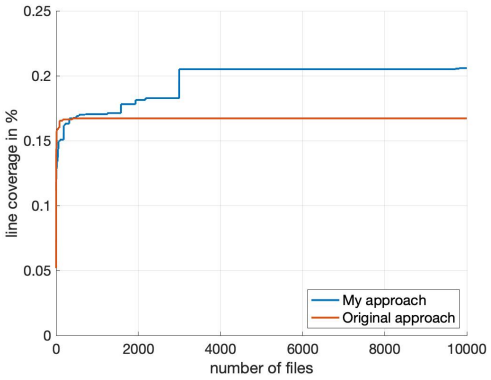


Figure 12: Line coverage of the ARGO parser.

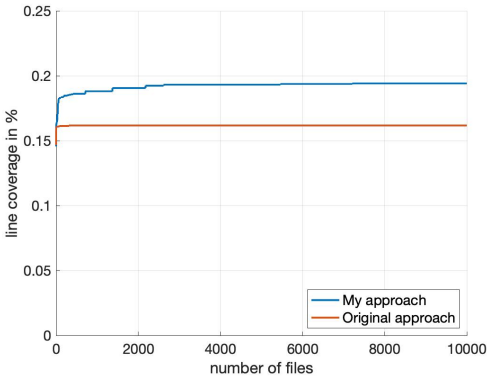


Figure 13: Line coverage of the Genson parser.

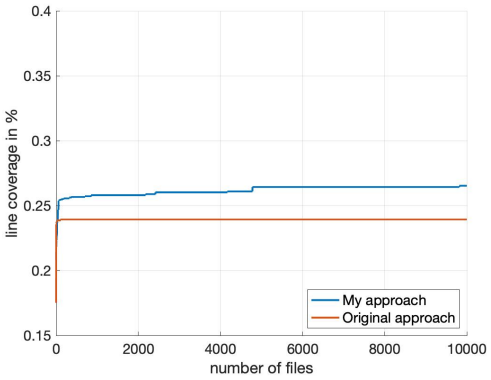


Figure 14: Line coverage of the Gson parser.

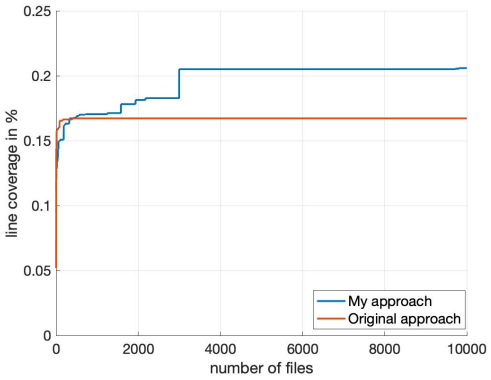


Figure 15: Line coverage of the JSON-Java parser.

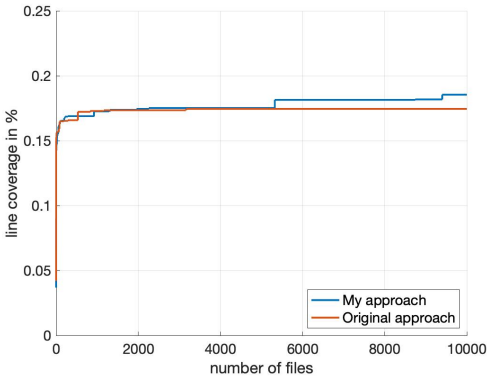


Figure 16: Line coverage of the jsonTo-Java parser.

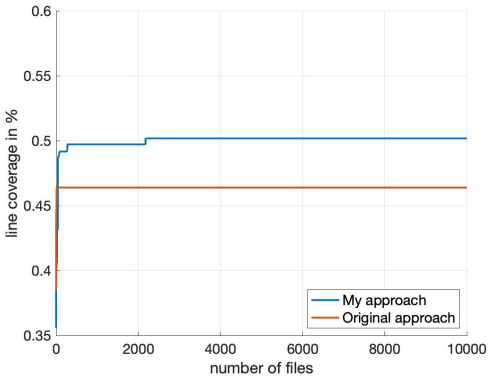


Figure 17: Line coverage of the minimalJSON parser.

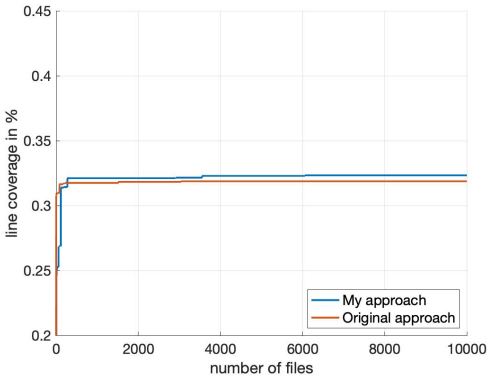


Figure 18: Line coverage of the Pojo parser.

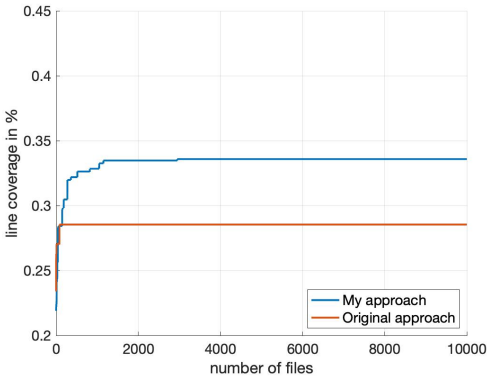


Figure 19: Line coverage of the json-simple parser.

The visual evaluation of the Figure 12 to Figure 21 shows that my approach was able to achieve at least the same percentage of line coverage than the original approach. Further, my approach was able to increase the line coverage for all subjects. For the language JSON my approach was able to improve the line coverage up to 21% (Genson, Fig. 13), for the language JavaScript up to 54 % (Rhino, Fig. 20) and for the language CSS3 up to 20 % (cssValidator, Fig. 21).

Further investigating the figures 12 to 19 shows that for almost all JSON parsers both approaches seem to hit a point after which no more line coverage is achieved. For the original approach this point seems to appear early on during the input generation. After only approximately 200 generated input files, no further improvement of the code coverage is achieved. The same applies to my approach, but the point after which no more major improvement is observed varies from parser to parser. This point lies between 2,100 (minimalJson, Fig. 17) and 5,000 (Gson, Fig. 14) generated input files.

Differently compared to the JSON parser, a specific point after which no more improvement of the line coverage occurs (within the 10,000 generated input files),

Subject	# Runs	LOC	My approach			Original approach			Mean increase
			max	mean	SD	max	mean	SD	
ARGO	10	8,265	49.71%	48.97%	0.62%	43.19%	43.19%	0%	13.39%
Genson	10	18,780	19.81%	19.30%	0.21%	16.17%	16.17%	0%	21.21%
Gson	10	25,172	27.07%	26.80%	0.18%	23.92%	23.92%	0%	12.05%
JSONJava	10	3,742	21.73%	19.71%	1.11%	16.72%	16.72%	0%	17.87%
jsonToJava	10	5,131	18.54%	18.22%	0.35%	17.51%	17.38%	0.10%	4.80%
minimalJSON	10	6,350	51.61%	51.32%	0.44%	46.38%	46.38%	0%	10.65%
Pojo	10	18,492	32.46%	32.31%	0.08%	31.88%	31.88%	0%	1.37%
json-simple	10	2,432	34.44%	34.29%	0.27%	28.54%	28.54%	0%	20.15%
Rhino	10	100,234	16.78%	15.68%	0.56%	10.75%	10.13%	0.37%	54.73%
cssValidator	10	120,838	8.67%	8.25%	0.22%	6.92%	6.83%	0.06%	20.80%

Table 2: Accumulated results over all test runs. For each subject and approach ten test runs were executed.

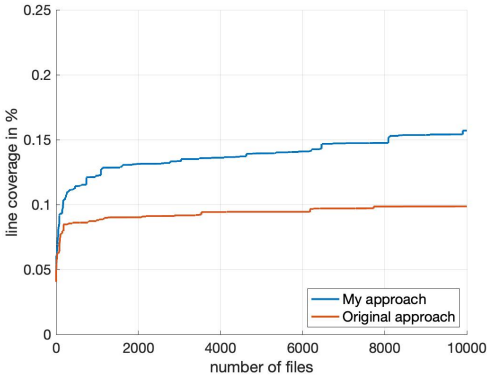


Figure 20: Line coverage of the Rhino parser.

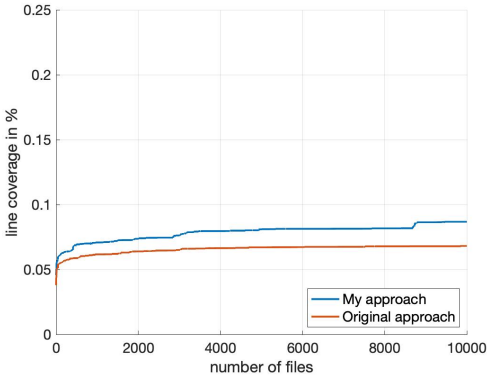


Figure 21: Line coverage of the cssValidator parser.

cannot be determined for the language JavaScript and the parser Rhino (Fig. 20). Both approaches manage to iteratively increase the line coverage over the generated input files.

Table 2 shows the accumulated results for each subject and over all test runs. For both approaches, the maximum and the mean line coverage as well as the standard deviation are shown, along with the mean increase of the line coverage of my approach compared to the original approach. The mean improvement of the line coverage ranges from 1.3 % to 54 %. Additionally, Table 2 hints, at least for the original approach (JSON), at the existence of a point, within the 10,000 generated input files per run, at which no more improvement is observed. Apart from the jsonToJava parser, all runs for each JSON parser achieved the same line coverage, resulting in a standard deviation of 0%.

Subject	# Runs	LOC	My approach median	Original approach median	U	p - value
ARGO	10	8,265	48.95%	43.19%	100	0.000063
Genson	10	18,780	19.66%	16.17%	100	0.000063
Gson	10	25,172	26.82%	23.92%	100	0.000063
JSONJava	10	3,742	19.11%	16.72%	100	0.000063
jsonToJava	10	5,131	18.34%	17.40%	94.5	0.000857
minimalJSON	10	6350	51.44%	46.38%	100	0.000060
Pojo	10	18,492	32.30%	31.88%	100	0.000060
json-simple	10	2,432	34.44%	28.54%	100	0.000048
Rhino	10	100,234	15.64%	10.05%	100	0.000182
cssValidator	10	120,838	8.13%	6.81%	100	0.000181

Table 3: Statistical analysis with the Mann-Whitney U test.

To support the visual evaluation, I also perform a statistical analysis on the achieved line coverage to increase the confidence in my conclusions. I perform a non parametric Mann-Whitney U test [22], [23] on the achieved code coverage of the original approach compared to my approach. The U test is a nonparametric test that evaluates if it is equally likely that a randomly selected value from one population will be less than or greater than a randomly selected value from the other population. I chose to perform the U test since both standard deviations are different and the number of ten test runs per subject is too low to make reliable statement with the Welch test. The statistical analysis confirms the results from the the visual inspection and can be seen in Table 3. The results show that two randomly selected runs from each approach, differ significantly for all subjects and are highlighted in blue. Exemplary, the median line coverage of my approach and the original approach for the jsonToJava parser were 18.34% and 17.40% respectively; the distribution of the two approaches differed significantly (Mann–Whitney U = 94.5, $N_{MyApproach} = N_{OriginalApproach} = 10$, $p < 0.05$ two-sided).

With the visual evaluation and the statistical analysis I conclude that my approach is able to achieve a higher line coverage than the original approach by Pavese et al.

RQ2: Can my approach trigger more exception types than the original approach?

To answer **RQ2**, I compare the number of times a unique exception type has been triggered. Table 4 shows the thrown exception types per subject and input language. If neither approach was able to trigger an exception, the subject is not included in the table. For the JsonJava, simple-json, minimal-json and cssValidator parsers no defects and exceptions have been found by both approaches. The ratios in column 4 and 5 relate to the number of test runs in which my approach and the original approach were able to trigger the exception.

Table 4 shows that my approach was able to detect all the exception types triggered by the original approach in each test run. Further, my approach was able to find seven

Input language	Subject	Exception types	My approach	Original approach
JSON	ARGO	argo.saj.InvalidSyntaxException	10 / 10	**0** / 10
	Genson	java.lang.NullPointerException	10 / 10	10 / 10
	Gson	java.lang.IllegalArgumentException	3 / 10	**0** / 10
	jsonToJava	org.json.JSONException	10 / 10	10 / 10
	jsonToJava	java.lang.IllegalArgumentException	10 / 10	10 / 10
	jsonToJava	java.lang.ArrayIndexOutOfBoundsException	10 / 10	10 / 10
	jsonToJava	java.lang.NumberFormatException	3 / 10	**0** / 10
	Pojo	java.lang.StringIndexOutOfBoundsException	10 / 10	10 / 10
	Pojo	java.lang.IllegalArgumentException	10 / 10	10 / 10
	Pojo	java.lang.NumberFormatException	8 / 10	**0** / 10
JavaScript	Rhino	java.lang.IllegalStateException	10 / 10	**0** / 10
	Rhino	java.util.concurrent.TimeoutException	10 / 10	**0** / 10
	Rhino	java.lang.ClassCastException	1 / 10	**0** / 10
		Total exception types	13	6

Table 4: Exception that have been triggered by my approach and the original approach during each test run.

exceptions that have not been triggered by the original approach. However, apart from the exception *argo.saj.InvalidSyntaxException*, found in the ARGO parser, the other six exceptions were not consistently found and were only triggered during some test runs.

Overall, 13 exception types in six subjects have been found during my experiments. Out of these 13 exception types, seven have not been triggered by the original approach. Figure 22 shows that the six exceptions triggered by the original approach were also found by my approach.

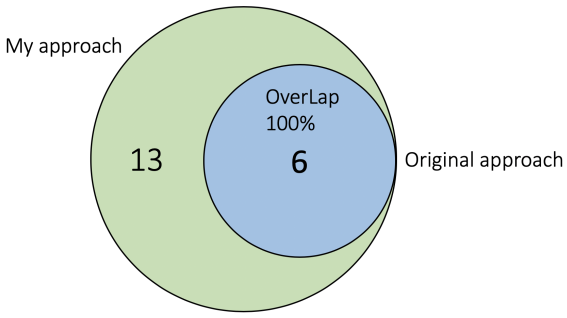


Figure 22: Number of total exceptions triggered by my approach (13) and the original approach (6).

6.3 Threats to validity

6.3.1 Internal validity

The biggest threat to the internal validity collides with the initial input corpus selection. Selecting a good and well distributed input corpus can be vital to validating the performance of the fuzzer. A "good" selected input corpus can boost the efficiency of the fuzzer severely, allowing it to find more defects or achieve a higher code coverage much faster. Testing the fuzzer multiple times on different initial input sets can increase the confidence in the results.

6.3.2 External validity

A threat to the external validity might be the selection of a grammar along with the matching parsers. An already well established and well tested parser will reduce the chance of finding any defect. The same applies to the complexity of a grammar. A complex grammar usually results in a more complex parsers and a complex parser is naturally more prone to defects than a simpler version. Due to the short amount of time, not every grammar and their parsers can be considered, nonetheless, selecting a specific language can have a major influence on the outcome of this study. Not neglecting this threat entirely, I concentrated on common languages ranging in complexity, namely JavaScript, CSS3 and JSON.

7 Discussion

This section further discusses the results of the evaluation and compares the evolutionary grammar-based fuzzer with the current *state-of-the-art* and other similar approaches.

The evaluation shows that my approach was able to significantly increase the code coverage and was able to find more exception types than the original approach. Comparing both approaches, the results show that the evolutionary *fuzzer* increased the line coverage for the languages JSON up to 20%, for JavaScript up to 54% and for CSS3 up to 20%. However, even though the *fuzzer* was able to increase the line coverage, the overall line coverage of the tested parsers is still arguably low. For the *Rhino* parser, for which my approach was able to achieve the highest increase, the overall line coverage lies only around 15.7%, leaving most of the code unexecuted. Despite the low line coverage, the fuzzer was still able to detect three exception types, hinting at defects and bugs.

Arguably one of the biggest threats to validity is the rather low experimentation size of only ten runs per subject and approach. The decision to limit the number of runs to ten, was influenced by the high execution time of the evolutionary fuzzer. The high time consumption is typical for evolutionary algorithms, with this approach being no exception. This is usually attributed to the analysis of each individual solution, in my case the analysis of each input file. Deriving the parse tree, returned by the ANTLR parser generator, and calculating the fitness for each individual can be considered as the bottleneck of this evolutionary process.

The low overall line coverage for almost all parsers may be due to the reason that although all generated input files were syntactically correct, they often violated the semantics of the grammar (e.g. in JavaScript, using an identifier that is never declared is considered to be a runtime error [12]). The state-of-the-art grammar-based fuzzing tool *LangFuzz* [12] solves this issue by replacing all undeclared identifiers with identifiers that occur somewhere in the rest of the program, reducing the risk of generating semantically incorrect inputs. Since replacing identifiers requires more knowledge of the targeted language, this method lies out of the scope of this thesis.

The main idea to build on the work of Pavese et al. and to extend it to an evolutionary fuzzing tool is that with the usage of the grammar, all generated input files are syntactically correct derived. This reduces the time spent generating syntactically incorrect inputs. Further, using the probabilistic grammar allows the fuzzer to minimize the *search space* of all possible inputs, and lets the fuzzer concentrate on the generation of "interesting" inputs. Additionally, by altering the individual probabilities, the probabilistic grammar guides the generation process. However, these probabilities can not be freely chosen. Directly altering the probabilities runs the risk of creating a probabilistic grammar that creates extensively large input files. Since the fuzzer is not aware of these conditions, it is not able to counteract. Pavese et al. define a threshold to omit this behavior, but as soon as the input generation reaches this threshold, the input generation is out of the control of the fuzzer and does not follow the probabilistic grammar anymore. At this point the fuzzer is no longer able to influence

the input generation. This is also the reason why the fuzzer does not make use of the defined *inverted probabilistic grammar* [5]. Generated inputs files did not always follow the specified inverted probabilistic grammar, simply because most of the time, the generation process was not able to terminate.

The learning of the probabilistic grammar from the best performing individuals, and the usage of that probabilistic grammar to generate a new population differs from typical evolutionary algorithms. In particular, genetic algorithms use *crossover mechanics* [24] to derive a new population of offsprings. This allows individuals to directly pass on their genes to the new generation. With this approach the fuzzer is only able to achieve this indirectly. Since only one probabilistic grammar is learned, all *genes* are combined and are influencing each other. The influence of an individual on the overall *gene pool* is therefore strongly connected to the total number of selected individuals from which the probabilistic grammar is learned.

Nevertheless, the evolutionary grammar-based fuzzing tool has proven that it is capable of successfully finding real-world defects in parsers.

8 Conclusion and future work

This thesis proposed an evolutionary grammar-based fuzzing tool to expose exceptional behavior in parsers and interpreters and evaluated the approach on real-world applications. The presented *fuzzer* uses the concepts of generating similar input files according to a probabilistic grammar, and extends these ideas to an evolutionary optimization approach. This generation-based process generates inputs along with the learning of new probabilistic grammars to create syntactically correct test cases. With the introduction of a fitness function and a selecting strategy, the fuzzer is able to generate structural complex input files that trigger defects and unwanted behavior. The introduced mutation of grammars maintains genetic diversity and allows the fuzzer to discover features that have previously not been explored.

The main contribution of this thesis, is the elaboration of difficulties when combining the generation of input files according to a probabilistic grammar with an evolutionary algorithm. The main challenge comes from countering the effects of the genetic drift to generate structural uncommon and interesting input files.

To evaluate the proposed approach, the evolutionary fuzzer was compared to the original input generation tool [5]. This thesis examined and compared both approaches on their capabilities of achieving a certain degree of line coverage, as well as examined their capabilities of detecting defects. The results show that the *fuzzer* was able to increase the line coverage for all tested subjects up to 50% (JavaScript, Rhino). Furthermore, the *fuzzer* was able to trigger and identify 13 different exception types in six parsers, from which seven exception types have not been triggered by the original approach.

Future work should include the further evaluation of the fuzzer, in particular the conducting of additional test runs to increase the confidence in the examined results and the observation of the performance of the fuzzer over a longer period of time. The evolutionary fuzzer should also be compared to the state-of-the-art to confirm its practicality. Additionally, future work should concentrate on reducing the time needed to calculate the fitness for each individual as well as the further improvement the fitness function. Combining the ideas of probabilistic input generation and the state-of-the-art fragmentation to create syntactically correct substructures is also worth more investigation.

9 References

[1] R. Richardson, "CSI Computer Crime & Security Survey," p. 31, 2008.

[2] M. Zalewski, "American fuzzing lop (afl)," 2018.

[3] "libFuzzer: A library for coverage-guided fuzz testing," 2018.

[4] O. Hallaraker and G. Vigna, "Detecting malicious JavaScript code in Mozilla," in *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pp. 85–94, June 2005.

[5] E. Pavese, E. Soremekun, N. Havrikov, L. Grunske, and A. Zeller, "Inputs from Hell: Generating Uncommon Inputs from Common Samples," *arXiv:1812.07525 [cs]*, Dec. 2018. arXiv: 1812.07525.

[6] B. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities.," *Commun. ACM*, vol. 33, pp. 32–44, Dec. 1990.

[7] A. Odena and I. Goodfellow, "TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing," *arXiv:1807.10875 [cs, stat]*, July 2018. arXiv: 1807.10875.

[8] X. Xie, L. Ma, F. Juefei-Xu, H. Chen, M. Xue, B. Li, Y. Liu, J. Zhao, J. Yin, and S. See, "DeepHunter: Hunting Deep Neural Network Defects via Coverage-Guided Fuzzing," *arXiv:1809.01266 [cs]*, Sept. 2018. arXiv: 1809.01266.

[9] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury, "Smart Greybox Fuzzing," *arXiv:1811.09447 [cs]*, Nov. 2018. arXiv: 1811.09447.

[10] 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*, PLDI '11, (New York, NY, USA), pp. 283–294, ACM, 2011. event-place: San Jose, California, USA.

[11] "Jesse Ruderman » Introducing jsfunfuzz."

[12] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, (Bellevue, WA), p. 38, USENIX Association, Aug. 2012.

[13] S. Veggalam, S. Rawat, I. Haller, and H. Bos, "IFuzzer: An Evolutionary Interpreter Fuzzer Using Genetic Programming," in *Computer Security – ESORICS 2016* (I. Askoxylakis, S. Ioannidis, S. Katsikas, and C. Meadows, eds.), Lecture Notes in Computer Science, (Cham), pp. 581–601, Springer International Publishing, 2016.

[14] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The Art, Science, and Engineering of Fuzzing: A Survey," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[15] J. E. Hopcroft, R. Motwani, and J. D. Ullman, "Introduction to automata theory, languages, and computation, 2nd edition," *ACM SIGACT News*, vol. 32, pp. 60–65, Mar. 2001.

[16] S. Wright, "The Evolution of Dominance," *The American Naturalist*, vol. 63, pp. 556–561, Nov. 1929.

[17] B. L. Miller, B. L. Miller, D. E. Goldberg, and D. E. Goldberg, "Genetic Algorithms, Tournament Selection, and the Effects of Noise," *Complex Systems*, vol. 9, pp. 193–212, 1995.

[18] H. Du, Z. Wang, W. Zhan, and J. Guo, "Elitism and Distance Strategy for Selection of Evolutionary Algorithms," *IEEE Access*, vol. 6, pp. 44531–44541, 2018.

[19] "The Definitive ANTLR 4 Reference by Terence Parr | The Pragmatic Bookshelf."

[20] J. C. Miller and C. J. Maloney, "Systematic mistake analysis of digital computer programs," *Communications of the ACM*, vol. 6, pp. 58–63, Feb. 1963.

[21] H. Kelly J., V. Dan S., C. John J., and R. Leanna K., "A practical tutorial on modified condition/decision coverage," 2001.

[22] H. B. Mann and D. R. Whitney, "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other," *The Annals of Mathematical Statistics*, vol. 18, pp. 50–60, Mar. 1947.

[23] A. Arcuri and L. Briand, "A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, pp. 219–250, May 2014.

[24] P. W. Poon and J. N. Carter, "Genetic algorithm crossover operators for ordering applications," *Computers & Operations Research*, vol. 22, pp. 135–147, Jan. 1995.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 25. März, 2020 .....