Fuzzing-based Grammar Inference

Hannes Sochor, Flavio Ferrarotti, and Daniela Kaufmann

Software Competence Center Hagenberg GmbH (SCCH), Hagenberg, Austria
{firstname.lastname}@scch.at

Abstract. In this paper we propose and suggest a novel approach for grammar inference that is based on grammar-based fuzzing. While executing a target program with random inputs, our method identifies the program input language as a human-readable context-free grammar. Our strategy, which integrates machine learning techniques with program analysis of call trees, uses a far smaller set of seed inputs than earlier work. As a further contribution we also combine the processes of grammar inference and grammar-based fuzzing to incorporate random sample information into our inference technique. Our evaluation shows that our technique is effective in practice and that the input languages of tested recursive-descending parser are correctly inferred.

Keywords: Fuzzing, Grammar-based Fuzzing, Software Testing, Grammar Inference, Grammar Learning, Program Analysis

1 Introduction

Software testing is one of the most important phases of the software lifecycle. This includes testing not only for functional correctness but also for safety and security. Finding bugs and security vulnerabilities presents a difficult task when facing complex software architectures. An integral part of software testing is fuzzing software with more or less random input and tracking how the software reacts. Having knowledge about the input structure of the software under test enables the fuzzer to generate more targeted inputs which significantly increases the chance to uncover bugs and vulnerabilities by reaching deeper program states.

As such the most successful fuzzers all come with some sort of model that describes the input structure of the target program. One of the most promising methods poses grammar-based fuzzing, where inputs are generated based on a context-free grammar which fully covers the so-called input language of a program. This makes it possible for the grammar-based fuzzer to produce inputs that are valid or near-valid, considerably raising its success rate. Although grammar-based fuzzing is a very successful method, in most cases such a precise description of the input language is not available.

The automation of learning input languages for a program, in our instance in the form of a synthesized context-free grammar, is still an issue in current grammar-based fuzzing techniques and is not completely resolved yet. With these capabilities, we would be able to apply grammar-based fuzzing to a wider range

of problems. Additionally it would be possible to utilize the inferred model for additional security analysis, such as comparing the inferred grammars of different implementations to determine whether they are equivalent.

While some current grammar-based fuzzing tools can be used more broadly but suffer from mistakes as a result, others are connected to specific programming paradigms or languages. In addition, state of the art grammar inference tools heavily depend on a good starting set of seed inputs to be able to correctly infer a grammar [8,10]. However, a good set of inputs is frequently not available, which leaves much room for improvement. Especially in the setting of security analysis that includes grammar-based fuzzing, a proper set of seeds is vital, as the inferred grammar has to be as accurate as possible.

In this paper we propose a novel automated method for grammar-based fuzzing, which automatically learns the grammar that is later used for fuzzing. In our technique we start from an incomplete seed grammar that is extracted from a small set of seed inputs. While fuzzing the target program, we actively learn and continuously enhance this grammar. These improvements in the grammar are based on information that we gain while executing our target program with randomly generated inputs and observing the response of the program. Our method makes use of a machine learning algorithm in combination with program analysis, more precisely, by extracting the call tree of some executed inputs. Our approach is generic and may be utilized regardless of the programming language of the target program because the learning process we use is black-box and the extraction of call trees is not based on a particular programming language. In addition, the input set needed for our approach is significantly smaller than in state of the art grammar inference tools [8,10]. The fundamental disadvantage of our approach is that it is restricted to recursive top-down parsers, which account for up to 80% of all parsers in use today [12]. Most grammatical inference tools also share this restriction, according to [8, 10].

Our experimental results show that our fuzzing-based grammar inference method enables us to learn a context-free grammar from tested recursive top-down parsers with the maximum possible accuracy in every case that we considered. Our technique accomplishes this in a relatively quick time while employing a simple program analysis technique. We further generate a human-readable grammar that may be applied to further security analysis.

The paper is structured as follows: We provide background information on formal definitions and learnability in language theory in Sect. 2. Our main contribution can be found in Sect. 3 where we present our method in detail, followed by an experimental evaluation of our approach in Sect. 4 as well as an in-depth discussion on related work in Sect. 5.

2 Preliminaries

In this section we introduce the necessary notation and theory for the rest of the paper. We assume the reader is familiar with basic concepts of language theory. An excellent reference for that is the classical book by Hopcroft and Ullman [9].

Let Σ be an *alphabet*, i.e., a finite set of symbols. A finite sequence of symbols taken from Σ is called a *word* or *string* over Σ. The free monoid of Σ, i.e., the set of all (finite) strings over Σ plus the empty string λ, is denoted as Σ^* and known as the *Kleene star* of Σ. If $v, w \in \Sigma^*$, then $vw \in \Sigma^*$ is the *concatenation* of v and w and $|vw| = |v| + |w|$ is its length. If $u = vw$, then v is a *prefix* of u and w is a *suffix*. A *language* is any subset of Σ^*.

A *grammar* is formally defined as a 4-tuple $G = (N, \Sigma, P, S)$, where N and Σ are finite disjoint sets of *nonterminal* and *terminal symbols* respectively, $S \in N$ is the *start symbol* and P is a finite set of *production rules*, each of the form:

$$(\Sigma \cup N)^* N (\Sigma \cup N)^* \to (\Sigma \cup N)^*.$$

We say G *derives* (or equivalently *produces*) a string y from a string x in one step, denoted $x \Rightarrow y$, iff there are $u, v, p, q \in (\Sigma \cup N)^*$ such that $x = upv$, $p \to q \in P$ and $y = uqv$. We write $x \Rightarrow^* y$ if y can be derived in zero or more steps from x, i.e., $\Rightarrow^*$ denotes the reflexive and transitive closure of the relation $\Rightarrow$.

The language of G, denoted as $\mathcal{L}(G)$, is the set of all strings in Σ^* that can be derived in a finite number of steps from the start symbol S. In symbols,

$$\mathcal{L}(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$$

In this work, Σ always denotes the "input" alphabet (e.g., the set of ASCII characters) of a given executable (binary) program p. The set of *valid inputs of* p is defined as the subset of Σ^* formed by all well formed inputs for p. In symbols:

$$\mathit{validInputs}(p) = \{w \in \Sigma^* \mid w \text{ is a well formed input for } p\}$$

The definition of a well formed input for a given program p depends on the application at hand. In our setting we only need to assume that it is possible to determine whether a given input string w is well formed or not for a program p by simply running p with input w.

As usual, we assume that $\mathit{validInputs}(p)$ is a context-free language. Consequently, there is a context-free grammar G_p such that $\mathcal{L}(G_p) = \mathit{validInputs}(p)$. Recall that a grammar is context-free if its production rules are of the form $A \to \alpha$ with A a single nonterminal symbol and α a possibly empty string of terminals and/or nonterminals.

Our main contribution in this paper is a novel algorithm that takes as input a program p and a finite (small) subset I of $\mathit{validInputs}(p)$, and infers a grammar G_p such that $\mathcal{L}(G_p)$ approximates $\mathit{validInputs}(p)$. We say "approximates" since it is *not* decidable in our setting (see [5]) whether $\mathcal{L}(G_p) = \mathit{validInputs}(p)$. To evaluate how well $\mathcal{L}(G_p)$ approximates $\mathit{validInputs}(p)$, we measure the precision and recall of $\mathcal{L}(G_p)$ w.r.t. $\mathit{validInputs}(p)$ as in [2], among others.

In our setting we first fix a procedure to calculate the probability distribution of a language, starting from its corresponding grammar. Following [2] we use random sampling of strings. Let $G = (N, \Sigma, P, S)$ be a context-free grammar.

As a first step, G is converted to a *probabilistic context-free grammar* by assigning a *discrete distribution* $\mathcal{D}_A$ to each nonterminal $A \in N$. As usual, $\mathcal{D}_A$ is of size $|P_A|$, where P_A is the subset of productions in P of the form $A \to \alpha$. Here, we assume that $\mathcal{D}_A$ is *uniform*. We can then *randomly sample a string x from the language* $\mathcal{L}(G, A) = \{w_i \in \Sigma \mid A \Rightarrow^* w_i\}$, denoted $x \sim \mathcal{P}_{\mathcal{L}(G,A)}$, as follows:

- Using $\mathcal{D}_A$ select randomly a production $A \to A_1 \cdots A_k \in P_A$.
- For $i = 1, \ldots, k$, recursively sample $x_i \sim \mathcal{P}_{\mathcal{L}(G,A_i)}$ if $A_i \in N$; otherwise let $x_i = A_i$.
- Return $x = x_1 \cdots x_k$.

The *probability distribution $\mathcal{P}_{\mathcal{L}(G)}$ of the language* $\mathcal{L}(G)$ is simply defined as the probability $\mathcal{P}_{\mathcal{L}(G,S)}$ induced by sampling strings in the probabilistic version of G defined above.

We can now measure the quality of a learned (or inferred) language $\mathcal{L}'$ with respect to the target language $\mathcal{L}$ in terms of precision and recall.

- The *precision* of $\mathcal{L}'$ w.r.t. $\mathcal{L}$, denoted $\mathit{precision}(\mathcal{L}', \mathcal{L})$, is defined as the probability that a randomly sampled string $w \sim \mathcal{P}_{\mathcal{L}'}$ belongs to $\mathcal{L}$. In symbols, $\Pr_{w \sim \mathcal{P}_{\mathcal{L}'}}[w \in \mathcal{L}]$.
- Conversely, the *recall* of $\mathcal{L}'$ w.r.t. $\mathcal{L}$, denoted $\mathit{recall}(\mathcal{L}', \mathcal{L})$, is defined as $\Pr_{w \sim \mathcal{P}_{\mathcal{L}}}[w \in \mathcal{L}']$.

We say that $\mathcal{L}'$ is a good approximation to $\mathcal{L}$ if it has both, high precision and high recall. Note that, a language $\mathcal{L}' = \{w\}$, where $w \in \mathcal{L}$, has perfect precision, but most likely has also very low recall. On the other hand, $\mathcal{L}' = \Sigma^*$ has perfect recall, but probably low precision.

It is well known that there are effective algorithms that may infer a finite automaton (and hence also a regular grammar or regular expression) to recognize $\mathcal{L}$ given a regular language $\mathcal{L}$ and a "teacher" that can answer *membership* and *equivalence queries* w.r.t. $\mathcal{L}$. The first and most well known algorithm of this kind was introduced by Dana Angluin [1] and is known as L*.

The *membership query* inquires as to whether a provided string is a part of the target language. Since it pertains to determining if a particular input to a program p belongs to $\mathit{validInputs}(p)$, this can obviously be addressed in our context. The *equivalence query* tests if the target language exactly matches the language that a given automaton (or grammar in our example) recognizes. Otherwise, the "teacher" ought to be able to offer a counterexample.

We leverage the approach NL* presented in [4] as part of our strategy (i.e. as a subroutine) in our algorithm to infer context free grammars from program input samples. This approach is based on L* and helps learning regular languages efficiently, although it learns residual-finite state machines as opposed to deterministic finite automata. The assumption is the same as for L*: a "teacher" who is able to respond to membership and equivalence questions. However, because we are unable to answer the equivalency question in our environment, we must instead rely on statistical sampling to look for counterexamples.

Algorithm 1: Fuzzing-based Grammar Inference Algorithm

Input : Seed inputs $I \subseteq \text{validInputs}(p)$, set of terminals Σ and program p
Output: Inferred grammar G'_p

$G_s := \mathit{findSeedGrammar}(I, p)$;
$N_s := G_s.\mathit{getNonTerminals}()$;
$G'_p := G_s.\mathit{clone}()$;
for $A \in N_s$ **do**
  $c := \mathit{null}$;
  repeat
    $M := \mathit{runNL}^*(p, A, \Sigma, N_s, c)$;
    $c := \mathit{searchForCounterexample}(M, p, A, G_s, 1000, 10)$ ▷ See Alg. 2;
  until $c = \mathit{null}$;
  $\alpha := M.\mathit{toRegularExpression}()$;
  $G'_p.\mathit{add}(\text{“}A \to \alpha\text{”})$
return G'_p

3 Algorithm

The goal of our approach is to infer a context-free grammar G'_p given a program p, a set of terminal symbols Σ as well as some valid inputs I. Ideally the language $L(G'_p)$ produced by our inferred grammar G'_p should be able to produce the input language $\mathit{validInputs}(p)$ of p such that $L(G'_p) = \mathit{validInputs}(p)$. To achieve our goal, we apply the following steps:

1. **Seed Grammar Extraction:** First we extract a seed grammar G_s from p using the valid inputs I.
2. **Seed Grammar Expansion:** Next we continuously expand the rules of G_s to achieve a better coverage of $\mathit{validInputs}(p)$. We do this by utilizing the NL* algorithm. While learning the rules of G'_p, we apply grammar-based fuzzing to find counterexamples needed during the learning process.
3. **Grammar-based fuzzing:** At some point we have inferred a grammar G'_p where finding a counterexample is hard because we have, or nearly have, identified $\mathit{validInputs}(p)$. We can run our grammar-based fuzzer indefinitely at this point until we uncover another counterexample, if one exists.

In this section we first give a formal description of our algorithm, followed by an example to better illustrate the learning process. Finally, we briefly discuss how our algorithm may be applied in a grammar-based fuzzing setting.

3.1 Learning context-free grammars

Assume we have a program p. We want to learn a grammar G_p such that $L(G_p)$ approximates $\mathit{validInputs}(p)$ as well as possible in terms of precision and recall (see preliminaries). As usual, the set of terminal symbols Σ of the target grammar G_p, or equivalently the set of characters accepted by p, is assumed to be known. We further assume an initial (finite) subset I of $\mathit{validInputs}(p)$. Our strategy is based on extracting a seed grammar from p using I, and then

Algorithm 2: Adapted Equivalence Query

Function: searchForCounterexample(M, p, A, G_s, n, m)
Input : Automaton M, program p, Seed Grammar G_s, NonTerminal A, Set of parse Trees T and maximum number n and m of trials and mutations per trial, respectively.
Output : Counterexample string if found. Otherwise, *null*.

```
1 G := M.toGrammar();
2 for (i := 0; i < n; i++) do
3     w := G.generateString();
4     if ¬membershipQuery(G_s, w, A, 𝒯, p) return w          ▷ See Alg. 3;
5     for (j := 0; j < m; j++) do
6         w' := w.applyMutation();
7         if w' ∉ L(G) ∧ membershipQuery(G_s, w', A, 𝒯, p) return w';
8 return null
```

Algorithm 3: Adapted Membership Query

Function: membershipQuery($G, w, A, \mathcal{T}, p$)
Input : Grammar $G = (N, \Sigma, P, S)$, $w \in (N \cup \Sigma)^*$, $A \in N \setminus \{S\}$, set $\mathcal{T}$ of parse trees, program p.
Output : *true* if $A \to w$ is deemed to be a good candidate for extending the productions of G. Otherwise *false*.

```
1  if ¬∃T x(T ∈ 𝒯 ∧ x ∈ nodes(T) ∧ label(x) = A) return false;
2  s := deriveString(w, G)              ▷ Derives s ∈ Σ* from w using G ;
3  T_A := parseTree(A, w, s, G)         ▷ Using left-most derivation and A → w ;
4  for each T ∈ 𝒯 do
5      for each x ∈ nodes(T) with label(x) = A do
6          T.replaceSubTree(x, T_A)     ▷ Subtree rooted at node x;
7      input := T.toString();
8      if input ∉ validInputs(p) return false;
9      else
10         if p.parseTree(input) = T return true else return false;
```

expanding it using grammar-based fuzzing of p until we obtain a good approximation of $validInputs(p)$. Grammar-based fuzzing of p means that we execute p with randomly sampled inputs produced from a given grammar. The concrete strategy is described in Alg. 1.

The seed grammar extraction is done by the function findSeedGrammar(I, p) (line 1 in Alg. 1). The set of non-terminals N_s of the seed grammar G_s is formed by the names of all functions called by executing p with inputs from I (line 2 in Alg. 1). The set of productions P_s of G_s is obtained as follows. For each $u \in I$,

1. Extract from p and u a parse tree T_u for u, where the internal nodes of T_u are labelled by non-terminals in N_s and the leaves are labelled by terminals in Σ.

2. For each internal node e of T_u with children $c_1, \ldots, c_n$ ordered from left to right, add the rule $A \to L_1 \ldots L_n$ to P_s, where $A, L_1, \ldots, L_n$ are the labels of the nodes $e, c_1, \ldots, c_n$, respectively.

The extraction of the parse trees T_u simply requires to track the function calls during an execution of p with input u. This can be done in multiple ways. In our case, we use the dynamic symbolic execution framework in our eknows platform [13]. A perfectly good alternative is to use dynamic tainting as in [8]. We omit here the technical details of this process as they are well known.

By construction, $I \subseteq L(G_s)$. Furthermore, if we assume that p uses a recursive descent parsing technique where each procedure/function implements one of the nonterminals of a grammar G_p such that $L(G_p) = \mathit{validInputs}(p)$, then $L(G_s) \subseteq \mathit{validInputs}(p)$. Moreover, for every nonterminal symbol A of G_s, we have that $\mathcal{L}(G_s, A) = \{w \in \Sigma \mid A \Rightarrow^* w\}$ is non empty, i.e., for every non-terminal A of the seed grammar there is at least one finite derivation that starts with A and produces a word in Σ^*.

Next the algorithm tries to augment the set of productions in G_s, i.e., in the seed grammar, to better approximate $\mathit{validInputs}(p)$ (line 4-11 in Alg. 1). It proceeds by considering, for every non-terminal symbol $A \in N_s$, a new rule of the form $A \to \alpha$, where α is a regular expression such that $\mathcal{L}(\alpha) \subseteq (N_s \cup \Sigma)^*$. We search for a suitable α using the NL* algorithm [4] couple with the procedure described in Alg. 3 and 2 to answer the necessary membership and equivalence queries, respectively. At the end of this process, we obtain α by translating (following the well known standard procedure) the automaton returned by NL* into an equivalent regular expression (line 10).

Each time NL* needs to answer a membership query for a string $w \in N_S \cup \Sigma$, it calls the procedure described in Alg. 3 with the following parameters:

- Seed grammar G_s.
- String w.
- Non-terminal A.
- Set $\mathcal{T}$ of parse trees such that $T_u \in \mathcal{T}$ iff $u \in I$ and T_u is the tree extracted in the previous stage (i.e., during the inference process of the seed grammar G_s) by tracking the function calls in the run of p with input u.
- Program p.

The function *deriveString* in line 2 of Alg. 3 produces a string $s \in \Sigma^*$ starting from w by applying the rules of G with a grammar-based fuzzing technique. Note that the function $\mathit{parseTree}(A, w, s, G)$ in line 3 returns the parse tree corresponding to the left-most derivation $A \Rightarrow w \Rightarrow^* s$ of G. On the other hand, $p.\mathit{parseTree}(\mathit{input})$ in line 10 returns the parse tree obtained by tracking the function calls in the run of p with *input*, using the procedure explained earlier. The remaining parts of this algorithm are self-explanatory.

Whenever NL* needs answer to the equivalence query for an automaton M, we apply the heuristic described in Alg. 2. Notice, that we do not have in this

```
 parse → expr
  expr → term | term+term | term−term
  term → factor | factor∗factor | factor/factor
factor → 1 | 2 | 3 | (expr)
```

Listing 1. Target Grammar

context a properly defined regular language that NL* needs to learn. Instead, we search for a counterexample string in $\mathcal{L}(M)$ that does not satisfy the (adapted) membership query expressed by Alg. 3, or vice versa. The search for a counterexample is performed until one is found or a maximum number of trials n has been reached. In each trial, the algorithm first derives a string $w \in (N_s \cup \Sigma)$ by applying grammar-based fuzzing with a grammar G equivalent to the automaton M (line 3). If w does not satisfy the (adapted) membership query, the algorithm returns w as a counterexample (line 4). Otherwise, it generates a mutation w' of w (line 6). If w' is not in the language recognized by G (or equivalently M) but satisfies the (adapted) membership query, then the algorithm returns w' as a counterexample (line 7). Otherwise, it tries different mutations of w up to a maximum number m. For the cases considered in the experiments reported in this paper, $n = 1000$ and $m = 10$ gives us optimal results and good performance.

3.2 Example

We will provide an example run of our algorithm to give a better understanding of the formal definition above. We will use the example grammar provided in Lst. 1 as G_p. Assume we have a parser p where $validInput(p) = L(G_p)$. We know the set of terminals $\Sigma = \{1, 2, 3, (,), +, -, *, /\}$ as well as the initial set of inputs $I = \{\text{“1”}\}$. We start extracting a seed grammar G_s by tracking p while executing $u \in I$. This returns a parse tree T_u. Figure 1 displays T_u as well as the result of transforming T_u to G_s.

Next we want to expand the rules of G_s. At this point we will use the expansion process of the rule `factor` $\rightarrow 1$ as a showcase example. To expand the rule, we want to learn `factor` $\rightarrow \alpha_{\texttt{factor}}$. First we have to identify which symbols are used by $\alpha_{\texttt{factor}}$. We do this by using static analysis to identify which functions are called by `factor` and add the according non-terminals as well as Σ. In our case the set of symbols is $\{\Sigma \cup \texttt{expr}\}$. Now we can apply NL* to learn $\alpha_{\texttt{factor}}$. To do so, we have to answer both membership-queries as well as equivalence-queries. First we will give some examples on how membership-queries are answered while learning the rules `expr` $\rightarrow \alpha_{\texttt{expr}}$ as well as `factor` $\rightarrow \alpha_{\texttt{factor}}$:

1. $w \in \mathcal{L}(\alpha_{\texttt{expr}})$ for $w =$ `term term`: We start by replacing the children of `expr` in T_u with w. Next we replace left non-terminals in the tree by applying the rules of G_s. This process is illustrated in Fig. 2. Finally, we transform the newly built parse tree to $w' = 11$ and execute p with w'. As $w' \notin validInputs(p)$, we return $w \notin \mathcal{L}(\alpha_{\texttt{expr}})$.
2. $w \in \mathcal{L}(\alpha_{\texttt{expr}})$ for $w =$ “`term + term`”: Again, we start by replacing the children of `expr` in T_u with w. Next we replace left non-terminals in the tree

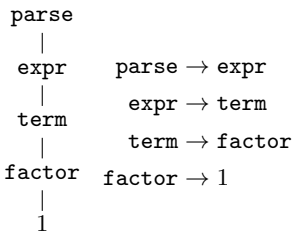

Fig. 1. From parse tree to grammar

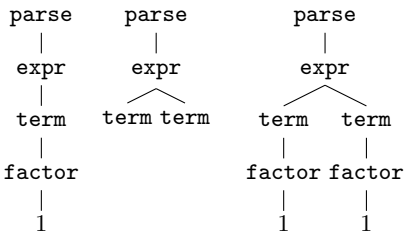

Fig. 2. Parse tree evolution of query 1

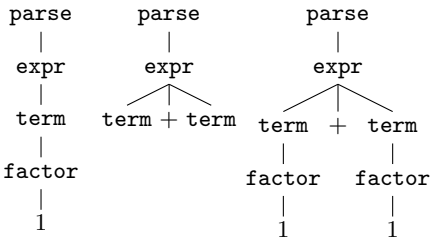

Fig. 3. Parse tree evolution of query 2

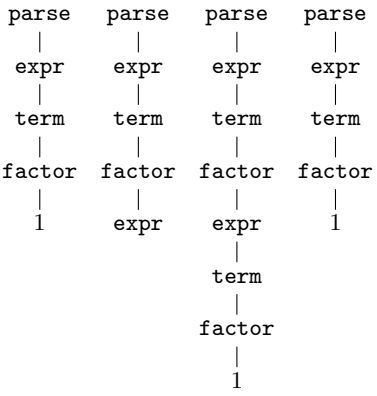

Fig. 4. Parse tree evolution of query 3

by applying the rules of G_s as illustrated in Fig. 3. When we execute p with input $w' = 1 + 1$ we see that $w' \in \mathit{validInputs}(p)$. Now we check the call tree of the execution and see that it is equivalent to the parse tree of w', so we return $w \in \mathcal{L}(\alpha_{\texttt{expr}})$.

3. $w \in \mathcal{L}(\alpha_{\texttt{factor}})$ for $w =$ "`expr`": Again, we replace the children of `expr` in T_u with w and resolve left non-terminals with G_s. This leads to the trees shown in Fig. 4. Executing p with input $w' = 1$ shows that $w' \in \mathit{validInputs}(p)$. Note that the call tree of executing w' (see the most right tree in Fig. 4) is not equivalent to the parse tree of w' so we return $w \notin \mathcal{L}(\alpha_{\texttt{factor}})$.

Next we provide an example on how we will answer an equivalence-query while learning the rule $\texttt{expr} \rightarrow \alpha$. We convert the given automaton M to a regular expression $\alpha = \texttt{term} \mid \texttt{term} + \texttt{term}$. Next we build a new grammar G_α containing a single rule of the form $S \rightarrow \texttt{term} \mid \texttt{term} + \texttt{term}$ with S being the start symbol. We continue by searching for a counterexample by generating new words using a grammar-based fuzzer using G_α. Assume we find a counterexample $c = \texttt{term} - \texttt{term}$ by applying a mutation to a generated word $\texttt{term} + \texttt{term}$. We return c and continue to answer membership-queries until the next equivalence-query is performed. In our case, the next automaton would translate to the regular expression $\alpha = \texttt{term} \mid \texttt{term}(-\texttt{term} \mid +\texttt{term} \mid \epsilon)$. As this already is the correct α, we will not be able to find another counterexample and stop

```
 parse → expr
  expr → term | term ( −term | +term | ϵ)
  term → factor | factor ( /factor | ∗factor | ϵ)
factor → 1 | 2 | 3 | (expr)
```

Listing 2. Learned Grammar

Table 1. Experiment Results

Target	Precision	Recall	MQ	EQ	Time
ExprParser	1.0	1.0	6 898	7	14s
MailParser	1.0	1.0	8 482	6	1m 42s
HelloParser	1.0	1.0	7 168	2	1m 09s
AdvExprParser	1.0	1.0	22 984	8	1m 47s
JsonParser	1.0	1.0	35 058	35	3m 23s

fuzzing when the maximum specified amount of tries is reached. Finally, we add `expr` → `term` | `term`(−`term` | +`term` | ϵ) to G'_p. We repeat the process described above for every non-terminal in G_s. The resulting grammar G'_p is shown in Lst 2. This grammar is equivalent to G_p given in Fig. 1.

3.3 Application in Grammar-based Fuzzing

While learning a rule $A \to \alpha$ we systematically explore a small part of p, more specifically we explore the function which is contributed to the non-terminal A. When an arbitrary membership-query is executed, we guide the parser to the exact part of the code where the query is parsed by executing the query in a known context. This has the effect that, while performing an equivalence query, we can effectively fuzz exactly that part of the code until we find a counterexample. Additionally, a positive side effect is that the mutations we insert will most likely explore border-line cases within the context of A which reduces the search space effectively. As the learning of a rule $A \to \alpha$ is completely independent from learning another, we may learn all the rules simultaneously. This enables us to run one grammar-based fuzzer targeting each function of p separately for an infinite amount of time, pausing fuzzing only when we are able to fine-tune the used grammar by a deterministic search with NL*.

4 Experiments & Evaluation

In this section we evaluate the performance of our fuzzing-based grammar inference method. Given a single input word, we apply our technique to different parsers and calculate the precision and recall of the inferred grammar. For each test run we start with a given target grammar G_p and automatically generate a parser p that accepts our target grammar G_p using the compiler-compiler Coco/R[1]. We apply our fuzzing-based grammar inference algorithm

[1] `https://ssw.jku.at/Research/Projects/Coco/`

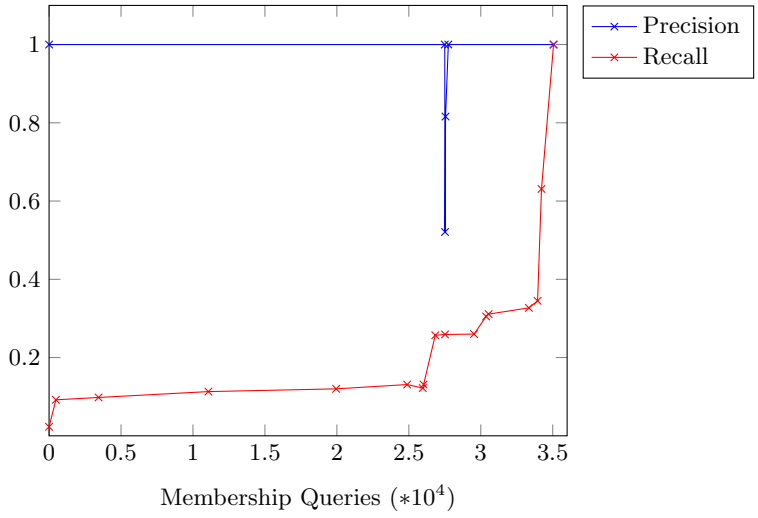

Fig. 5. Json Learning process

on this setup. If a specified maximum amount of allowed membership queries has been reached without finding a counterexample, execution is stopped and the current state of the inferred grammar G'_p is returned. Finally, we calculate $\mathit{precision}(L(G'_p), L(G_p))$ and $\mathit{recall}(L(G'_p), L(G_p))$ by randomly sampling 1 000 words each.

Table 1 displays the results, with the first column indicating the targeted parser. The second and third columns show the resulting precision and recall of our extracted grammar, followed by the total amount of unique membership-queries (MQ) performed, the total number of equivalence-queries (EQ) performed, and the time elapsed. The target grammar G_p implemented by each parser p as well as the inital input I for each test-run is given in App. A. Our results in Tbl. 1 show that we are able to recover a grammar that perfectly matches the target grammar for the 1 000 inputs examined. This demonstrates that our technique can recover a context-free grammar from a recursive top-down parser.

In the following we provide more details for the experiment "JsonParser". The target grammar is given in Lst. 3. Listing 4 shows our learned grammar. The learned grammars of the other parsers are given in the appendix. Additionally Fig. 5 shows a more detailed analysis of the learning process for "JsonParser". We calculate $\mathit{precision}(L(G'_p), L(G_p))$ and $\mathit{recall}(L(G'_p), L(G_p))$ every time an equivalence query is performed, where G'_p is the current state of the inferred grammar. Again, we use 1 000 randomly sampled words each. As can be seen in Fig. 5, precision stays at 1 most of the time during the learning process, due to the fact that $L(G_s) \subseteq L(G_p)$ (see Sect. 3.1).

If only a portion of the desired language is accepted by the rule at the measurement point, precision remains at 1. Precision may gradually drop as you learn more rules over time. This could occur when attempting to identify the correct body of a rule, in particular when the rule accepts a superset of the

Initial Input: "{'a'. $1e-0$,'' : [$true$, $true$]}"

```
      Json → Element.
   Element → Ws Value Ws.
        Ws → " ".
     Value → Object|Array|
             String|Number|
             "true"|"false"|"null".
    Object → "{" Ws [String Ws ":"
             Element["," Members]] "}".

   Members → Member ["," Members].

    Member → Ws String Ws ":" Element.
     Array → "[" Ws [Value Ws
             [ "," Elements ]] "]".

  Elements → Element [ "," Elements ].

    String → "'" Characters "'".
Characters → ϵ|Character Characters.
 Character → "a"|"b"|"c".
    Number → Integer Fraction
             Exponent.
   Integer → ["−"] ("0"|
             Onenine [Digits]).

    Digits → Digit [Digits].
     Digit → "0"|Onenine.
   Onenine → "1"|"2"|"3".
  Fraction → ϵ|"." Digits.
  Exponent → ϵ|"E" Sign Digits|
             "e" Sign Digits.
      Sign → ϵ|"+"|"−".
```

Listing 3. Json Target Grammar

```
      Json → Element.
   Element → Ws Value Ws.
        Ws → " ".
     Value → Object|Array|
             String|Number|
             "true"|"false"|"null".
    Object → "{" (Ws "}"|Ws ("}"|
             String Ws ":" (Element
             "}"| Element ("}"|","
             Members "}"))))).

   Members → Member|
             Member ("," Members|ϵ).

    Member → Ws String Ws ":" Element.
     Array → "[" (Ws "]"|Ws ("]"|
             Value (Ws "]"|Ws ("]"|
             "," Elements "]")))).

  Elements → Element|
             Element ("," Elements|ϵ).

    String → "'" Characters "'" .
Characters → ϵ|Character Characters.
 Character → "a"|"b"|"c".
    Number → Integer Fraction
             Exponent.
   Integer → "0"|Onenine|
             Onenine (Digits|ϵ)|
             "−" (Onenine|Onenine
             (Digits|ϵ)|"0").

    Digits → Digit|Digit (Digits|ϵ).
     Digit → "0"|Onenine.
   Onenine → "1"|"2"|"3".
  Fraction → ϵ|"." Digits.
  Exponent → ϵ|"E" Sign Digits|
             "e" Sign Digits.
      Sign → ϵ|"+"|"−".
```

Listing 4. Json Learned Grammar

wanted language. These inaccuracies are automatically fixed when a counterexample is found. For example the drop in precision in Fig. 5 occurs while learning a rule which consumes integers. The learned automaton accepts words containing preceding zeros as well as words containing more than one "-" at the beginning. Both are not accepted by the parser. As such, the precision of the learned grammar was lowered to 0.5. After the drop in precision, first the issue with multiple "-" symbols is fixed by providing a counterexample. This raises precision to 0.8. Finally, after providing another counterexample and consequently disallowing preceding zeros, the rule is learned correctly and precision increases back to 1.

In terms of recall, we see a consistent increase over time as the learnt grammar is expanded, and as a result, the learned language grows significantly. When a rule that consumes terminals is learned, the boost in recall is often greater. For example, the final spike in recall occurs while learning the rules for parsing digits and mathematical symbols.

We must remark that we have rarely used optimizations in our implementation, which leaves a lot of room for improvement. Possible performance improvements include (i) using hash-tables to store previously seen membership-queries

instead of a plain-text list, (ii) replacing the early-parser used to determine whether a grammar produces a given word with something more efficient, (iii) using paralellization to speed up fuzzing, and (iv) to simultaneously learn the different rules of the seed grammar.

5 Related Work

Extracting context-free grammars for grammar-based fuzzing is not a new idea. Several methods exist for grammar learning which try to recover a context-free grammar by means of membership-queries from a black-box, such as by beginning with a modestly sized input language and then generalizing it to better fit a target language [2,15]. Another approach synthesizes a grammar-like structure during fuzzing [3]. However, this grammar-like structure has a few shortcomings, e.g., multiple nestings that are typical in real-world systems are not represented accurately [8]. Other methods use advanced learning techniques to derive the input language like neural networks [7] or Markov models [6]. Although black-box learning is generally promising, it suffers from inaccuracies and incompleteness of learned grammars. It is shown in [1] that learning a context-free language from a black box cannot be done in polynomial time. As a result, all pure black-box methods must give up part of the accuracy and precision of the learnt grammars.

Due to limitations with black-box approaches there exist several white-box methods to recover a grammar. If full access to the source code of a program is given, described methods fall under the category of grammar inference. Known methods for inferring a context-free language using program analysis include AUTOGRAM [10] and MIMID [8]. Unlike its predecessor AUTOGRAM, which relies on data flow, MIMID uses dynamic control flow to extract a human readable grammar. Finally, [11] describes how a grammar can be recovered using parse-trees of inputs, which is then improved with metrics-guided grammar refactoring. All of the aforementioned grammar inference methods share the same flaw: They all primarily rely on a predetermined set of inputs from which a grammar is derived that corresponds to this precise set of inputs. If some parts of a program are not covered by the initial set of inputs, the resulting grammar will also not cover these parts. However there exist some methods that attempt to automatically generate valid input for a given program, such as symbolic execution [14].

6 Conclusion

Our main contribution is a novel approach for grammar inference that combines machine learning, grammar-based fuzzing and program analysis. Our approach, in contrast to other efforts, reduces reliance on a good set of seed inputs while keeping other advantageous features of grammar inference techniques. This reduction in the original input set causes us to perform more membership queries since we need to uncover paths that we lose by randomly sampling the input set. We exchanged some of the benefits of complex program analysis, such as dynamic symbolic execution, for less complex program analysis, such as call tree

extraction, to speed up the execution because our approach was designed with grammar-based fuzzing in mind. This was done in order to process such a vast number of inputs effectively. We can cease grammar inference and resume fuzzing the target program using the inferred grammar whenever we are certain that we have a good enough approximation of our input language. Despite the trade-offs outlined above, our preliminary findings show that we can still learn the target input language accurately in a reasonable amount of time, especially for more complicated input languages like JSON.

In the future, we aim to improve our learning technique by looking into ways to learn context-free grammars from any program without being restricted by recursive top-down parsers. Furthermore, we want to enhance the current implementation with a range of performance optimizations so that we may utilize it to uncover security problems in a real-world scenario.

References

1. Angluin, D., Kharitonov, M.: When won't membership queries help? J. Comput. Syst. Sci. **50**(2), 336–355 (1995)
2. Bastani, O., Sharma, R., Aiken, A., Liang, P.: Synthesizing program input grammars. In: PLDI. pp. 95–110. ACM (2017)
3. Blazytko, T., Aschermann, C., Schlögel, M., Abbasi, A., Schumilo, S., Wörner, S., Holz, T.: GRIMOIRE: synthesizing structure while fuzzing. In: USENIX Security Symposium. pp. 1985–2002. USENIX Association (2019)
4. Bollig, B., Habermehl, P., Kern, C., Leucker, M.: Angluin-style learning of NFA. In: IJCAI. pp. 1004–1009 (2009)
5. E.Gold: Language identification in the limit. Inf. Control. **10**(5), 447–474 (1967)
6. Gascon, H., Wressnegger, C., Yamaguchi, F., Arp, D., Rieck, K.: Pulsar: Stateful black-box fuzzing of proprietary network protocols. In: SecureComm. LNICST, vol. 164, pp. 330–347. Springer (2015)
7. Godefroid, P., Peleg, H., Singh, R.: Learn&fuzz: machine learning for input fuzzing. In: ASE. pp. 50–59. IEEE Computer Society (2017)
8. Gopinath, R., Mathis, B., Zeller, A.: Inferring input grammars from dynamic control flow. CoRR **abs/1912.05937** (2019)
9. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley (1979)
10. Höschele, M., Zeller, A.: Mining input grammars with AUTOGRAM. In: ICSE (Companion Volume). pp. 31–34. IEEE Computer Society (2017)
11. Kraft, N., Duffy, E., Malloy, B.: Grammar recovery from parse trees and metrics-guided grammar refactoring. IEEE Trans. Software Eng. **35**(6), 780–794 (2009)
12. Mathis, B., Gopinath, R., Mera, M., Kampmann, A., Höschele, M., Zeller, A.: Parser-directed fuzzing. In: PLDI. pp. 548–560. ACM (2019)
13. Moser, M., Pichler, J.: eknows: Platform for multi-language reverse engineering and documentation generation. In: 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 559–568 (2021)
14. Moser, M., Pichler, J., Pointner, A.: Towards attribute grammar mining by symbolic execution. In: SANER. pp. 811–815. IEEE (2022)
15. Wu, Z., Johnson, E., Yang, W., Bastani, O., Song, D., Peng, J., Xie, T.: REINAM: reinforcement learning for input-grammar inference. In: ESEC/SIGSOFT FSE. pp. 488–498. ACM (2019)

A Experiment Details

A.1 ExprParser

Initial Input: “1”

```
COMPILER Expr
PRODUCTIONS
  Expr = Term [("+"|"-") Term].
  Term = Factor [("*"|"/") Factor].
  Factor = "1" | "2" | "3" | "(" Expr ")".
END Expr.
```

Listing 5. Expr Target Grammar

```
COMPILER Parse
PROSUCTIONS
  Parse = expr.
  expr = term | term ( "-" term | "+" term | ϵ).
  term = factor | factor ( "/" factor | "*" factor | ϵ).
 factor = "1" | "2" | "3" | "("expr")".
END Parse.
```

Listing 6. Expr Learned Grammar

A.2 MailParser

Initial Input: “a@a.aa”

```
COMPILER Mail
TOKENS
  Char = "a" .. "z".
PRODUCTIONS
  Mail = String "@" String "." Tag.
  String = Char { Char }.
  Tag = Char Char [Char].
END Mail.
```

Listing 7. Mail Target Grammar

```
COMPILER Parse
PRODUCTIONS
Parse = string "@" string "." tag.
string = char string_q1 .
string_q1 = (char string_q1  | ϵ).
tag = char (char | char (char | ϵ) ) .
char = ("z" | "y" | "x" | "w" | "v" | "u" | "t" | "s" | "r" |
    "q" | "p" | "o" | "n" | "m" | "l" | "k" | "j" | "i" | "h"
    | "g" | "f" | "e" | "d" | "c" | "b" | "a" ).
END Parse.
```

Listing 8. Mail Leanred Grammar

A.3 HelloParser

Initial Input: “hello”

```
COMPILER Hello
PRODUCTIONS
  Hello = ("H" | "h") ("E" | "e") ("L" | "l")
          ("L" | "l") ("O" | "o").
END Hello.
```

Listing 9. Hello Target Grammar

```
COMPILER Parse
PRODUCTIONS
Parse = ("h" ("e" ("l" ("l" ("o" | "O" ) | "L" ("o" | "O" ) )
      | "L" ("l" ("o" | "O" ) | "L" ("o" | "O" ) ) ) | "E" (
    "l" ("l" ("o" | "O" ) | "L" ("o" | "O" ) ) | "L" ("l" ("
    o" | "O" ) | "L" ("o" | "O" ) ) ) ) | "H" ("e" ("l" ("l"
     ("o" | "O" ) | "L" ("o" | "O" ) ) | "L" ("l" ("o" | "O"
     ) | "L" ("o" | "O" ) ) ) | "E" ("l" ("l" ("o" | "O" )
    | "L" ("o" | "O" ) ) | "L" ("l" ("o" | "O" ) | "L" ("o"
    | "O" ) ) ) ) ).
END Parse.
```

Listing 10. Hello Learned Grammar

A.4 AdvExprParser

Initial Input: “1”

```
COMPILER AdvExpr
PRODUCTIONS
  AdvExpr = Expr.
  Expr = Term { ("+"|"-") Term }.
  Term = Factor { ("*"|"/") Factor}.
  Factor = Num | "(" Expr ")".
  Num = Digit { Digit }.
  Digit = "1" | "2" | "3".
END AdvExpr.
```

Listing 11. AdvExprParser Target Grammar

```
COMPILER Parse
PRODUCTIONS
AdvExpr = Expr .
Expr = Term Expr_q2 .
Expr_q2 = ("-" Term Expr_q2   | "+" Term Expr_q2   | ϵ).
Term = Factor Term_q2 .
Term_q2 = ("/" Factor Term_q2   | "*" Factor Term_q2   | ϵ).
Factor = (Num | "(" Expr ")" ).
Num = Digit Num_q1 .
```

```
Num_q1 = (Digit Num_q1 | ϵ).
Digit = ("3" | "2" | "1").
Parse = AdvExpr.
END Parse.
```

Listing 12. AdvExprParser Learned Grammar

A.5 JsonParser

Initial Input: "{ '*a*' . 1*e* − 0 , '' : [*true* , *true*] }"

```
COMPILER Json
PRODUCTIONS
  Json = Element.
  Value = Object | Array | String | Number |
          "true" | "false" | "null".
  Object = "{" Ws [String Ws ":" Element
           ["," Members]] "}".
  Members = Member ["," Members].
  Member = Ws String Ws ":" Element.
  Array = "[" Ws [Value Ws [ "," Elements ]] "]".
  Elements = Element [ "," Elements ].
  Element = Ws Value Ws.
  String = "'" Characters "'".
  Characters = "" | Character Characters.
  Character = "a" | "b" | "c".
  Number = Integer Fraction Exponent.
  Integer = ["-"] ("0" | Onenine [Digits]).
  Digits = Digit [Digits].
  Digit = "0" | Onenine.
  Onenine = "1" | "2" | "3".
  Fraction = "" | "." Digits.
  Exponent = "" | "E" Sign Digits | "e"
             Sign Digits.
  Sign = "" | "+" | "-".
  Ws = " ".
END Json.
```

Listing 13. Json Target Grammar

```
COMPILER Parse
PRODUCTIONS
Json = Element.
Element = Ws Value Ws.
Value = ("true" | "null" | "false" | String | Object | Number
    | Array ).
Object = "{" (Ws "}" | Ws ("}" | String Ws ":" (Element "}" |
    Element ("}" | "," Members "}" ) ) ) ) .
Array = "[" (Ws "]" | Ws ("]" | Value (Ws "]" | Ws ("]" | ","
    Elements "]" ) ) ) ) .
```

```
String = """ Characters """ .
Number = Integer Fraction Exponent .
Ws = " " .
Members = (Member | Member ("," Members | ϵ) ).
Member = Ws String Ws ":" Element.
Elements = (Element | Element ("," Elements | ϵ) ).
Characters = (ϵ | Character Characters ).
Character = ("c" | "b" | "a" ).
Integer = (Onenine | Onenine (Digits | ϵ) | "0" | "-" (Onenine
    | Onenine (Digits | ϵ) | "0" ) ).
Fraction = (ϵ | "." Digits ).
Exponent = ("e" Sign Digits | ϵ | "E" Sign Digits ).
Onenine = ("3" | "2" | "1" ).
Digits = (Digit | Digit (Digits | ϵ) ).
Digit = (Onenine | "0" ).
Sign = (ϵ | "-" | "+" ).
Parse = Json.
END Parse.
```

Listing 14. Json Learned Grammar