Parser-Directed Fuzzing

Björn Mathis
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany
bjoern.mathis@cispa.saarland

Rahul Gopinath
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany
rahul.gopinath@cispa.saarland

Michaël Mera
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany
michael.mera@cispa.saarland

Alexander Kampmann
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany
alexander.kampmann@cispa.saarland

Matthias Höschele
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany
hoeschele@cs.uni-saarland.de

Andreas Zeller
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany
zeller@cispa.saarland

Abstract

To be effective, software test generation needs to well cover the space of possible inputs. Traditional *fuzzing* generates large numbers of random inputs, which however are unlikely to contain keywords and other specific inputs of non-trivial input languages. *Constraint-based test generation* solves conditions of paths leading to uncovered code, but fails on programs with complex input conditions because of path explosion. In this paper, we present a test generation technique specifically directed at *input parsers.* We systematically produce inputs for the parser and track comparisons made; after every rejection, we satisfy the comparisons leading to rejection. This approach effectively covers the input space: Evaluated on five subjects, from CSV files to JavaScript, our pFuzzer prototype covers more tokens than both random-based and constraint-based approaches, while requiring no symbolic analysis and far fewer tests than random fuzzers.

CCS Concepts • **Software and its engineering → General programming languages**.

Keywords fuzzing, test generation, parsers

ACM Reference Format:
Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. 2019. Parser-Directed Fuzzing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 13 pages.
https://doi.org/10.1145/3314221.3314651

1 Introduction

The field of software test generation aims at improving reliability and robustness of software by subjecting it to artificially generated inputs. Since every behavior of a program, including unwanted ones, can be triggered through its inputs, it is important to have *valid* inputs that reach program functionality without being rejected as invalid, and to have a large *variety* in these inputs in order to cover a maximum of functionality. So far, both goals have been addressed by two testing strategies.

- *Traditional fuzzing* [37] operates at the *lexical* level, quickly producing large numbers of inputs with random characters. Fuzzing is very easy to deploy, and quickly finds bugs in lexical and syntactical processing. For programs with nontrivial input languages, though, a pure random approach is unlikely to generate complex input structures such as keywords—already producing a string like `"while"` by pure chance from letters only is $1 : 26^5$ (11 million), not to speak of a condition or a statement that would have to follow the keyword.
- *Constraint-based test generation* [5] operates at the *semantic* level, considering the semantics of the program under test. It satisfies conditions on the path towards (yet) uncovered code, using constraint solving and symbolic analysis to easily solve short paths, especially at the function level. The problem of constraint-based test generators, however, is scalability: For nontrivial input languages, they quickly suffer from a combinatorial explosion of paths to be explored.

In the above context, a "nontrivial" input language need not be a programming language (although these probably rank among the most complex input languages). The Wikipedia page on file formats [33] lists 1,435 data input formats, from AAC to ZZT; while all of these formats have at least one parser, few of these formats have machine-readable grammars or other language specifications. Even if a program accepts a data exchange format with a fixed syntax such as

XML or JSON, finding the valid tags and labels it accepts will be hard. We thus pose the input language challenge: *Given a program with a nontrivial input language, the challenge is to find a test generator that covers all lexical and syntactical features of that language.*

In this paper, we propose a novel *syntax-driven approach* for the problem of generating plausible inputs. We call it *parser-directed fuzzing* as it specifically targets *syntactic* processing of inputs via input parsers. Our assumptions are that the program under test (1) processes input elements (characters) sequentially, that it (2) compares these elements against possible *valid* values, and that it (3) either accepts inputs as valid or rejects them as invalid—i.e., the typical features of a syntactic parser. We also assume that we can *track comparisons* of input characters; we do this through *dynamic tainting* of inputs, which allows us to relate each value processed to the input character(s) it is derived from.

Our approach relies on the observation that parsers rely on a *lookahead* of a limited number of characters, which is very often just a single character. Hence, rather than trying to solve the complete input, we look for any character that lets the parser advance further without erroring out.

Our approach, illustrated in Figure 1, specializes towards parsers that process one input character at a time, joining characters into tokens and these again into syntactic structures—that is, the "textbook" approach to parsing. It starts by feeding a small fixed string to the program (in general one random character [1]). This string is typically rejected by the input parser. However, on rejection of the input, our fuzzer derives the character comparisons made to each index of the input. That is, it identifies the *valid prefix* in the input, as well as the character comparisons made to the first invalid character. The fuzzer then corrects the invalid character to pass one of the character comparisons that was made at that index, and the new string is fed back to the parser. This new string will typically fail at the next character, at which point, we repeat the process. This process is continued until the parsing phase ends (that is, the string that was fed to the program is accepted by the parser). The complete valid input is saved as a possible input for the program.

After illustrating our approach on a detailed example (Section 2), we present our two key contributions:

1. **A test generator aimed at input parsers.** Our approach, discussed in Section 3, is the first test generation approach that systematically explores and covers the syntactical input space as accepted by input parsers. It only requires dynamic tainting, tracking of comparisons, and structural coverage information, which is far more lightweight than symbolic analysis. Our approach is guaranteed to produce valid inputs for input parsers that scan ahead a fixed number of characters. The prototype implementation (Section 4) works on C programs, which it instruments using LLVM.
2. **A comparison with lexical and semantical test generators.** In our evaluation on five subjects from CSV to JavaScript (Section 5), we approximate coverage of the input space by assessing the set of valid tokens produced; clearly, if a test generator fails to produce some language token, it also cannot cover the associated program features. We show that our syntax-driven approach covers the input space better than state-of-the-art "lexical" mutation-based fuzzers such as AFL [37], while requiring fewer tests by orders of magnitude; it also outperforms state-of-the-art "semantic" constraint-based fuzzers such as KLEE [5]. All of our inputs are syntactically valid by construction.

After discussing related work (Section 6), Section 7 points out current limitations and directions to future work. Section 8 is the conclusion and thus concludes this paper.

2 Parser-Directed Fuzzing in a Nutshell

Consider the following problem: Given a program P, how can we *exhaustively* cover the *syntactic features* of its input language?

To illustrate our approach, let us assume we want to exhaustively test the program P. We know nothing about P; in particular, we have no documentation or example inputs. What we know, though, is that

1. P accepts some input I sequentially as a string of characters; and that
2. P can tell us whether I is a valid or an invalid input.

We further assume that we can *observe* P processing I: Specifically, we need to be able to observe the dynamic *data flow of input characters* from I as P processes them.

Figure 1 illustrates how we explore the capabilities of P's input parser by means of directed test generation. The key idea is to observe all *comparisons an input character goes through*, and systematically satisfy and cover alternatives, notably on rejection.

We start with an empty string as input, which is rejected by P as invalid immediately as *EOF* is encountered. The *EOF* is detected as any operation that tries to access past the end of given argument. This error is fixed in the next round by testing P with a random string, say `"A"` (I = `"A"`). Indeed, this input is also rejected by P as invalid. However, before rejecting the input, P checks I for a number of properties. Only after these checks fail does P reject the input:

1. Does I start with a digit?
2. Does I start with a `'('` character?
3. Does I start with `'+'` or `'-'`?

All these conditions are easy to satisfy, though—and this is a *general* property of parsers, which typically only consider the single next character. Our test generator picks one

[1]Ignoring *EOF* detection for the time being. See Section 3 for details.

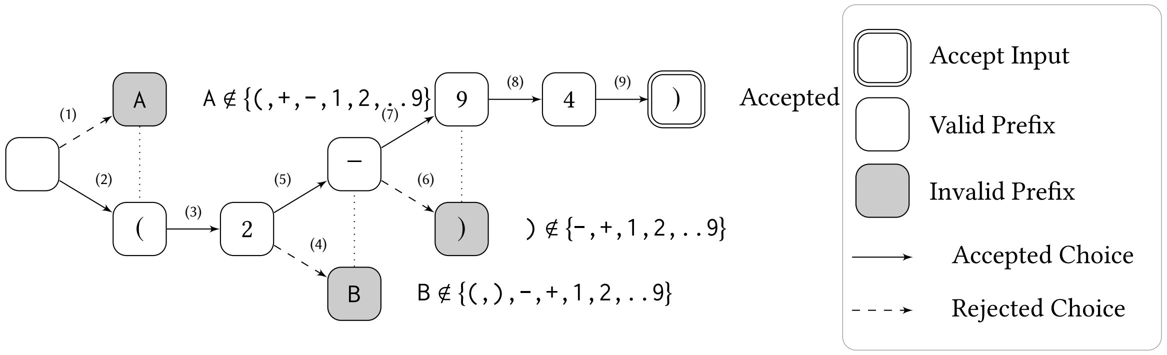

Figure 1. How parser-directed fuzzing works. We start with an empty string `""` as input to the instrumented program. This prefix is immediately rejected by the input parser, but we detect an attempt to access the index `0` of the input string. (An attempt to access a character beyond the length of the input string is interpreted as the program encountering *EOF* before processing is complete). Detecting this, we (1) choose a random character from the set of all ASCII characters, say `'A'`, and append it to the current input prefix `""`, resulting in `"A"`. This prefix is again rejected by the program, but not before we detect a number of failed comparisons at the index `0` where `'A'` was compared against `'('`, `'+'`, `'-'` and digits `'1'`, `'2'`, ... We replace our last appended character with a new character from this list of comparisons, say `'('`, and (2) the prefix `"("` is sent to the program again. We detect another *EOF* at this time, and we append (3) another random character from the ASCII set `'2'`. We get lucky at this time, however, and detect another *EOF*, and we (4) append another random character `'B'` to the prefix resulting in `"(2B"`, which is run again. Continuing this process, we reach a complete valid input `"(2-94)"`.

condition randomly. Satisfying Property 2, it would produce an open parenthesis input (that is, `"("`). This prefix would now be accepted by *P* as valid.

After the acceptance of `"("` as a partial input, *P* conducts a check to see if another character follows `"("` by accessing the next character in the input. Since *P* reached the end of the string we consider the prefix as valid and add another random character. This results in the new prefix `"(2"`. This prefix however, is accepted as valid immediately, and the program attempts to access the next character. After detecting this attempt, we choose a random character from the set of all characters: `'B'`. This new prefix `"(2B"` is rejected after `'B'` is compared to the set of characters `'('`, `')'`, `'-'`, `'+'`, `'1'`, `'2'`, ..., from which we choose a replacement `'-'`. This is again a valid partial input, and needs more characters to be appended. For the next character the generator randomly chooses `')'` from the available list of all characters. This however, is rejected as invalid, and in consequence, it gets replaced by a valid character `'9'`. Continuing this process, we reach `"(2-94)"` which is accepted as a complete valid input by the parser.

In a consecutive execution with another random seed, the first condition to be addressed might be Property 1. Satisfying it yields `'1'`, which is a valid input. At this point, we may decide to output the string and reset the prefix to empty string, or continue with the generated prefix. If we continued with the prefix, we may append `'+'` which will again cause the parser reaching the end of the input, so we append a random character and get `"1+B"` as input. This is rejected, but only after again checking for a number of expected characters that could follow. These would be the same checks already performed on the input `"A"`: digits, parentheses, `'+'`, and `'-'`. We randomly choose the condition Property 2, where again the prefix `"1+("` would be invalid on its own, so we again choose one prefix for further computations.

By continuing this process, we thus obtain more and more inputs that systematically cover the capabilities of the parser. In the end, we obtain a *set of legal inputs that covers all the conditions encountered during parsing*:

```
1  11   +1   -1   1+1   1-1   (1)
```

We see that our mystery program *P* in fact takes arithmetic expressions as inputs. Furthermore, instead of randomly guessing a correct combination the input is built character by character. Thus, building a valid input of size *n* takes in worst case *2n* guesses (assuming the parser only checks for valid substitutions for the rejected character).

One might ask: How likely is it that one can instrument the program under test but does not have enough knowledge about the input format to apply more sophisticated fuzzing approaches (like grammar based fuzzing)? Obviously, any test generation approach, including parser-directed fuzzing, can benefit from human knowledge, be it a grammar or just the available keywords. Despite this, there are some good reasons to still use parser-directed fuzzing:

- First, our approach is also applicable to binary code as it only relies on dynamic tainting and the comparisons made during execution, both are available for binary code as well.
- Second, even if we knew the grammar, it is often not available in a machine readable form and creating a

correct grammar is a time consuming task. And even if a formal grammar exists, it could contain errors or encode more or less features than implemented in the code. Concluding, parser-directed fuzzing makes it possible to test the code fully automatic and without any prejudice.

3 Testing Input Parsers

What we thus introduce in this work is a test generator specifically built to explore *input parsers*. Our fuzzer combines the best of both lexical and constraint-based approaches: it solves an easy constraint, namely that it replaces the character that was lastly compared with one of the values it was compared to. On the other hand, similar to random fuzzing, parser-directed fuzzing produces new inputs rapidly and verifies the correctness of each input using the program under test. With this method, the search space is reduced significantly. That is, the number of possible replacements at each position of the input is constrained by the comparisons that are made at this position (namely the ones the input parser expects at this index). Identifying a replacement for the last character is computationally cheap, especially compared to full fledged constraint solving. This combination of character replacement and semi-random fuzzing on a small search space makes parser-directed fuzzing efficient and effective compared to its competition.

While the idea is simple in principle, it hides some complexities in practice. For example, consider a simple parenthesis input language which require well-balanced open and close parentheses. For this input language, at each step, a parser would compare the last character of a valid prefix with both `'('` and `')'`. Hence, to extend the valid prefix say `"(()(("`, one could choose either `'('` or `')'`. The problem is that, when one appends an open parenthesis, a corresponding close parenthesis has to be inserted at some point in the future, and given that we are relying on random choice between `'('` and `')'`. The probability of closing after n steps [2] is given by $\frac{1}{n+1}$. After 100 characters, this probability is about 1%, and continues to decrease as we add more characters. Hence, relying on random choice to get us a valid complete string does not work in practice.

Naively, one could think of using depth- or breadth-first search to explore the possible character replacements. Depth-first search is fast in generating large prefixes of inputs but may not be able to close them properly as we have seen before, and may therefore get stuck in a generation loop. Breadth-first search on the other hand explores all combinations of possible inputs on a shallow level and is therefore helpful in closing prefixes (like appending closing parentheses). Generating a large prefix is, however, hard as we have to deal with the combinatorial explosion. A combination of both would be useful for simple input structures but fails for more complex ones as depth-first search might open too many elements that need to be closed that is beyond the capability of the breadth-first search to close within a given computational budget.

pFuzzer uses a *heuristic search* to guide the input generation through the large search space of possible inputs. It primarily takes structural coverage information into account for deciding which inputs should be executed next. Structural coverage alone though would lead to a kind of depth-first search which would generate large and complex prefixes that are very hard to close and will likely never end up in a valid input. Thus, the heuristic also tries to guide the search to a closing path which will lead to smaller but valid and diverse inputs.

The heuristic value used in pFuzzer is based on branch coverage. However, we do not apply coverage naively but rather combine different perspectives on the covered code to help guidance through the search space based on the context.

3.1 Achieving Coverage

Algorithm 1 sketches the general generation loop of our approach. We start with an empty set that will later contain the branches covered by valid inputs (Line 2). Furthermore we store all non-executed inputs in a priority queue throughout fuzzing (Line 3). The inputs in the queue are primarily sorted based on the heuristic defined in the procedure in Line 47: first the number of newly covered lines of the parent is taken (Line 48), then the length of the input is subtracted and two times the length of the replacement is added (Line 49). Using the length of the input avoids a depth-first search based on the coverage as very large inputs have less priority. By using the length of the replacement we can lead the algorithm to more interesting values, e.g. values that stem from string comparisons. Such replacements will likely lead to the complex input structures we want to cover. Furthermore, recursive-descent parsers increase their stack when new syntactical features are discovered (e.g. an opening brace) and decrease their stack on characters that close syntactical features (e.g. a closing brace). Therefore, at Line 50 we take the average stacksize between the second last and last comparison into account [3], larger stack sizes will give less priority to the input. Finally, we add the number of parents to the heuristic value (Line 50). This number defines how many substitutions were done on the search path from the initial input to the current input. Inputs with fewer parents but the same coverage should be ranked higher in the queue to keep the search depth and input complexity low.

[2]The parenthesis language is an instance of a Dyck path. The number of total paths with $2n$ steps that stay positive is $\binom{2n}{n}$. Out of these, those that end in 0 after $2n$ steps is $\frac{1}{n+1}\binom{2n}{n}$. This is the n^{th} Catalan number C_n. Hence, the probability of a closed Dyck path is $\frac{1}{n+1}$, which after 100 steps is 1% – We ignore those paths that reached zero and rebounded in both denominator and numerator for convenience.

[3]We omit a concrete implementation of avgStackSize here to keep the algorithm short.

Algorithm 1 Parser-Directed Fuzzing Algorithm.

```
 1: procedure FUZZ
 2:     $vBr \leftarrow \emptyset$
 3:     $queue \leftarrow$ empty queue
 4:     $input \leftarrow$ random character
 5:     $eInp \leftarrow input$
 6:     while True do
 7:         $valid, comps \leftarrow$ RUNCHECK($input, vBr, queue$)
 8:         if not $valid$ then
 9:             $valid, comps \leftarrow$ RUNCHECK($eInp$)
10:             if not $valid$ then
11:                 ADDINPUTS($eInp, branches, vBr, comps$)
12:             end if
13:         end if
14:         $input \leftarrow queue$.GET( )
15:         $eInp \leftarrow input$ + random character
16:     end while
17: end procedure
18:
19: procedure ADDINPUTS($inp, branches, vBr, comps$)
20:     for all $c$ in $comps$ do
21:         $cov \leftarrow$ HEUR($branches, vBr, inp, c$)
22:         $new\_input \leftarrow$ replace $c$ in $inp$
23:         add $new\_input$ to $queue$ based on $cov$
24:     end for
25: end procedure
26:
27: procedure RUNCHECK($inp, vBr, queue$)
28:     ($exit, branches, comps$) $\leftarrow$ RUN($inp$)
29:     if $exit = 0 \land (branches \setminus vBr) \neq \emptyset$ then
30:         VALIDINP($inp, branches, vBr, queue, comps$)
31:         return $True, comparisons$
32:     else
33:         return $False, comparisons$
34:     end if
35: end procedure
36:
37: procedure VALIDINP($input, branches, vBr, queue, comps$)
38:     PRINT($input$)
39:     $vBr \leftarrow branches \cup vBr$
40:     for all $inp$ in $queue$ do
41:         $cov \leftarrow$ HEUR($inp.branches, vBr, inp, inp.c$)
42:         reorder $inp$ in $queue$ based on $cov$
43:     end for
44:     ADDINPUTS($input, branches, vBr, comps$)
45: end procedure
46:
47: procedure HEUR($branches, vBr, inp, c$)
48:     $cov \leftarrow$ SIZE($branches \setminus vBr$)
49:     $cov \leftarrow cov -$ LEN($inp$) $+ 2 *$ LEN($c$)
50:     $cov \leftarrow cov -$ AVGSTACKSIZE( ) $+ inp.numParents$
51:     return $cov$
52: end procedure
```

Each loop iteration while fuzzing consists of two program executions. First the input without a random extension is executed (Line 7), then the input with random extension is executed (Line 9). We use this technique because the input without extension is generated from an input by replacing the last compared character with one of the characters it was compared to (e.g. Line 22), which means that we would never append a character but always replace the last character of the input [4]. Therefore, we need to append a new character to the end of the input, and check if it is used in any comparisons—which in turn means that with a high chance all previous characters are valid. On the other hand, if we always append a new character, pFUZZER may be very unlikely to produce a correct input because as soon as the correct replacement was done, a new appended character will make the input invalid again. By running both we can ensure to not get stuck but also find all valid inputs along the search path.

While looking for inputs that cover new code, we first concentrate on the number of branches that were not covered by any valid input beforehand. Those inputs that cover new branches have a higher priority in the queue. This metric gives the strongest evidence on which input prefixes are the most promising to explore further. As the already covered branches only contain branches covered by valid inputs it does not contain branches of error handling code. Hence, simply taking all covered branches would favor invalid inputs at some point and guide the search to use invalid prefixes. To avoid that the search gets stuck at paths with invalid inputs, we only consider the covered branches up to the last accepted character of the input. In particular, we consider all covered branches up to the first comparison of the last character of the input that is compared in the program.

3.2 Making the Input Valid

As soon as new code is covered the search algorithm needs to "close" the input, i.e. make it valid. This is important as trying to cover as much code as possible with each and every input will lead to very complex inputs that are hard to close, possibly taking hours of search to find the correct sequence of characters that make the input finally valid. Think about a simple bracket parser looking for the first valid input. Say the parser is able to parse different kinds of brackets (round, square, pointed, ...). As we never had any valid input any time a different opening bracket is added, more code would be covered and we might end up generating many different opening brackets, closing them though might be hard as one has to generate at each position the correct counter part for the respective opening bracket. To avoid such a generation loop we count each found new branch only once for each

[4] Not all parsers use an *EOF* check, hence we need the random extension to check if a new character is expected or the substitution was wrong.

input and also taking stack size and input length into account. Hence, we generate small prefixes that are simple to close.

After the first few valid inputs are found, the majority of the code is covered. At this point it gets significantly harder to find characters that help closing an input by just considering the newly covered branches themselves. For example, if we already generated an `if()`-statement in a valid input beforehand, the code handling the closing brace would have already been covered, we would not see any new coverage on generating another closing brace for an `if()`-statement. Therefore, we take input length, stack size and number of parents into account to favor those inputs that are less complex and keep the number of opened syntactical features low. This makes it possible to leave coverage plateaus in the search space since often a closing character would for example lead to a lower stack. Finally, to avoid generation of inputs that cover the same path as already generated inputs, pFuzzer keeps track of all paths that were already taken (based on the non-duplicate branches) and ranks inputs based on how often they executed on the same path, ranking those highest that cover new paths.

After an input was closed and covered new branches in the code, all remaining inputs in the queue have to be re-evaluated in terms of coverage. Due to the large search space, re-running all inputs again on the subject under test takes too much time. A faster way is storing all relevant information to compute the heuristic along with the already executed input, and simply re-calculating the heuristic value again (e.g. the loop at Line 40).

4 Implementation

We have implemented our approach in a fuzzing tool called pFuzzer. pFuzzer takes as input a C program, which it compiles and instruments using LLVM. The instrumentation serves the purpose of parser-directed fuzzing in that it (1) dynamically taints input characters and derived values throughout execution. When read, each character is associated with a unique identifier; this taint is later passed on to values derived from that character. If a value is derived from several characters, it accumulates their taints. Runtime conversion functions such as `atoi()` are wrapped such that the taints automatically propagate correctly. Any comparisons of tainted values (mostly character and string comparisons) are (2) tracked as well.

To drive the test generation heuristics, the instrumentation also tracks function and branch coverage, specifically (3) the sequence of function calls together with current stack contents, and (4) the sequence of basic blocks taken. pFuzzer is not optimized for speed, and thus its instrumentation adds considerable overhead; as a rule of thumb, executions are slowed down by a factor of about 100. All the collected information is saved after program execution, and then drives test generation for the next program run as detailed in Section 3.

5 Evaluation

We evaluated the performance of pFuzzer against KLEE and AFL. We evaluated the fuzzers on two aspects:

Code Coverage. The first measure we look at is the *code coverage* obtained, i.e. how many of the branches in the subject programs would actually be taken. Code coverage is a common metric in testing and test generation; generally speaking, covering a piece of code is necessary to uncover errors in this very code.

Input Coverage. The second measure we look at is *input coverage* obtained, i.e. how many aspects of the input language are actually covered. To this end, we measure how many different *tokens* are produced and what the characteristics of these tokens are. In general, coverage of input language features is necessary to trigger functionality associated with these features.

5.1 Evaluation Setup

Table 1. The subjects used for the evaluation.

Name	Accessed	Lines of Code
INIH	2018-10-25	293
CSVPARSER	2018-10-25	297
cJSON	2018-10-25	2,483
TINYC	2018-10-25	191
MJS	2018-06-21	10,920

For our evaluation, we use five input parsers with increasing input complexity, summarized in Table 1.[5] Starting with the simple input formats INI [3] and CSV [20], we also test the tools on JSON [10], TINYC [22] (a subset of C) and MJS [6] (a subset of JavaScript). We set up all programs to read from standard input (a requirement for AFL), and to abort parsing with a non-zero exit code on the first error (a requirement for pFuzzer). Furthermore, we disabled semantic checking in MJS as this is out of scope for this paper.

AFL is run with the standard configuration. As we cannot change the CPU scaling policy on our computing server, AFL_SKIP_CPUFREQ is enabled. Since AFL requires a valid input to start with but we want to evaluate the ability of all tools to generate valid inputs without program specific knowledge, we give AFL one space character as starting point. This character is accepted by all programs as valid while still being generic enough such that we think AFL is in no advantage compared to KLEE and pFuzzer not having any input to start with. KLEE is run with the `uclibc`, `posix-runtime`, and `optimization` options enabled. Furthermore, we run KLEE to only output values if they cover

[5]As pFuzzer is a research prototype and thus we want to keep the engineering effort reasonable, we restrict ourselves to randomly selected recursive descent parsers implemented in C as we need to special handle some library functions (like *strcmp()*).

new code (otherwise KLEE would produce millions of test inputs, for which calculating the coverage would take weeks and is therefore out of scope for this paper). For AFL and KLEE, we determine the valid inputs by running the programs under test and checking the exit code (non-zero means invalid input); pFuzzer by construction only outputs valid inputs that cover new code.

Test generation for complex subjects needs time. All tools were run for 48 hours on each subject in a single-core thread on Intel processors with 3.3GHz running Ubuntu 14.04.5 in the KLEE Docker container. All tests were run three times; we report the best run for each tool.

5.2 Code Coverage

A general measure to assess the quality of the test suite is code coverage. This metric is used to predict the general *chance* of a set of tests to find an error in the program. It is important to note that coverage of code is first and foremost a *necessity* to find bugs in this code. To use it as a universal measure for test quality, one must assume that bugs are distributed evenly across code, and that all bugs are equally likely to be triggered, which typically is not the case.

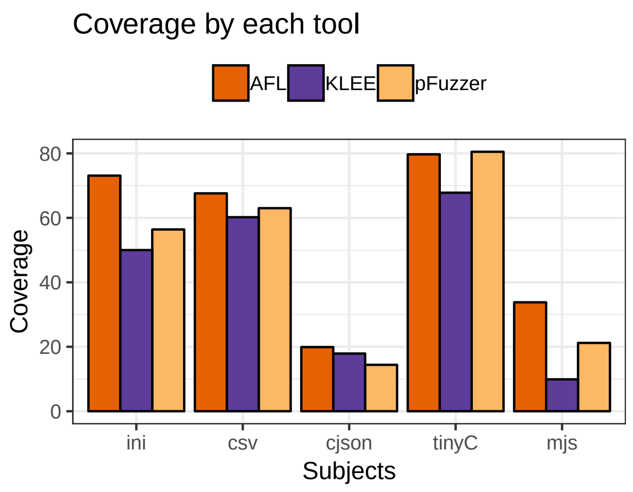

Figure 2. Obtained coverage per subject and tool.

Figure 2 shows the coverage of the valid inputs generated by each tool. Each input is parsed; tinyC and mjs also execute the program. We use gcov to measure branch coverage. All subjects may also contain to some extent code that is used for parsing from a different input source, printing or even generating (e.g. cJSON also functions as a library which contains code to generate a json object). Since it is not always clear which code can be covered and all tools run on the same code, we decided to leave those artifacts in even though they cannot be covered. All tools can still be compared on each individual subject. In one case, we manually fixed an input `while(9);` to `while(0);` to avoid an infinite loop during the execution of the generated test inputs.[6]

csv and ini. Starting with the least complex input structures, csv and ini, we can see that AFL performs better than pFuzzer in terms of coverage. For both subjects AFL has a high advantage: it is random, shallow, and fast. Therefore, it is able to generate many different characters in a short time, almost all of which are accepted by ini and csv. For both subjects, covering all combinations of two characters achieves perfect coverage. As an example, one of the most challenging structures in the inputs of those parsers is the section delimiter in ini which needs an opening bracket followed by a closing bracket. Between those, any characters are allowed. Hence, generating such valid inputs for those subjects is easy. AFL is in advantage here as all those properties are easy to reach and can be covered by exploring the input space randomly. pFuzzer performs worse here as in some circumstances semantic properties of the input are checked, or non-printable characters are used in comparisons which are not covered by our comparison extraction yet.

json. For json, both KLEE and AFL obtain a higher coverage than pFuzzer, which misses the one feature set of the conversion of UTF16 literals to UTF8 characters. The problem is that the developers of the parser rely on an implicit information flow and we currently do not handle them, because naively tainting all implicit information flows can lead to large overtainting [21].[7] Therefore, we never reach the parts of the code comparing the input with the UTF16 encoding. Nonetheless, in contrast to AFL, pFuzzer is able to generate keywords such as `true`, `false` and `null`, and cover the associated code.

tinyC. On tinyC, pFuzzer is actually able to generate inputs that cover more code than AFL. The reason for this lies in the complexity yet simplicity of the implementation of tinyC: the subject accepts rather simple inputs (like simple arithmetic expressions) but they cover only a small part of the code and this coverage is easily achieved. To go beyond these easy to reach code parts, one has to generate more complex structures like loops and branching statements, which parser-directed fuzzing can do.

mjs. For mjs, AFL achieves a much higher code coverage than pFuzzer. KLEE, suffering from the path explosion problem, finds almost no valid inputs for mjs. Examining code coverage, we found that AFL mostly covers code triggered by single characters or character pairs. Again, as with tinyC,

[6]Such infinite loops would normally be addressed by implementing a timeout; however, gcov loses all coverage information when the program under test is interrupted. AFL also generates an input that triggers a hang in tinyC. As this is an `if`-statement which should actually terminate, we are not able to fix the hang.

[7]For example: the program may read the input character by character in a loop header and processes it in the loop body. All values in the loop body would be tainted, leading to an over-approximation of taints.

those parts of the code that require very specific sequences of characters to make the input valid are not covered by AFL.

Summarizing, AFL achieves its coverage by brute force, trying out millions of different possible inputs, generating 1,000 times more inputs than pFuzzer. Thus, if one wants to cover shallow code, AFL is the tool to choose.

AFL quickly covers code that is easy to reach.

However, our manual coverage investigation also shows that as soon as it comes to more structured parts in the program, both AFL and KLEE fail to generate inputs that are able to cover them. Therefore, in the next section, we attempt to capture the input features that characterize these structured parts.

5.3 Input Coverage

In parser code, uncovered parts often are specific language constructs that are optional and need to be specifically tested. Think about tinyC: whatever input is given, it must contain at least one expression, therefore expressions are tested in depth anyway. Constructing a valid `while` loop on the other hand, requires two elements:

- The `while` keyword itself. Such a long keyword is hard to generate by pure chance—even if a fuzzer would generate letters only, the chance for producing it would be only 26^5, or 1 in 11 million. Guidance by code coverage, as implemented in AFL, does not help here, as all five characters take the exact same path through the program.
- The elements following the `while` keyword also must be produced—i.e., a parenthesized expression and at least an empty statement.

To assess input quality, we evaluate the *variety of tokens* generated by each approach—notably, what kind of tokens each of the approaches would be able to generate in its valid inputs. To this end, we first collected all possible tokens by checking the documentation and source code of all subjects and then checked how many different tokens appear in each subject. Tables 2, 3 and 4 contain the numbers of possible tokens per length and for each length a set of examples. Strings, numbers and identifiers are classified as one token as they can consist of many different characters but will all trigger the same behavior in the program. Any non-token characters (e.g. whitespaces) are ignored from the count.

Table 2. json tokens and their number for each length.

Length	#	Examples
1	8	`{ } [ ] - : ,` *number*
2	1	*string*
4	2	`null true`
5	1	`false`

Our results are summarized in Figure 3, presenting the different tokens generated for ini, csv, json, tinyC, and mjs.

ini and csv have few tokens; on ini, KLEE fails to detect an opening and a closing bracket. Those are used to define a section.

json has a structured input format, posing first challenges for test generators. As we can see from Table 2, AFL misses *all* json keywords, namely `true`, `false` and `null`. This supports the assumption that the random exploration strategy of AFL has trouble finding longer keywords. KLEE, however, is still able to cover most of the tokens; only a comma is missing. As KLEE works symbolically, it only needs to find a valid path with a keyword on it; solving the path constraints on that path is then easy. pFuzzer, by contrast, is able to cover all tokens, more than the other tools.

Table 3. tinyC tokens and their number for each length.

Length	#	Examples
1	11	`< + - ; = { } [ ]` *identifier number*
2	2	`if do`
4	1	`else`
5	1	`while`

tinyC comes with few tokens, but a number of keywords, listed in Table 3. As shown in Figure 3, simple constructs needing only one or two characters are easy to generate for AFL; the semi-random approach eventually guesses the correct characters and their respective ordering. KLEE does not find any keyword. pFuzzer is still able to cover 86% of all tokens, missing only the `do` and `else` token. AFL still covers 80% of all tokens but also misses a `while` token; and while KLEE covers 66% of all tokens, it only covers short ones, missing *all* keywords of tinyC.

Table 4. mjs tokens and their number for each length.

Len	#	Examples
1	27	`{ [ ( + & ?` *identifier number* ...
2	24	`+= == ++ /= &= |= != if in` *string* ...
3	13	`=== !== <<= >>> for try let` ...
4	10	`>>>= true null void with else` ...
5	9	`false throw while break catch` ...
6	7	`return delete typeof Object` ...
7	3	`default finally indexOf`
8	3	`continue function debugger`
9	2	`undefined stringify`
10	1	`instanceof`

mjs is our most challenging test subject, and it continues the trends already seen before. As shown in Figure 3, KLEE mostly fails, whereas AFL can even generate short keywords.

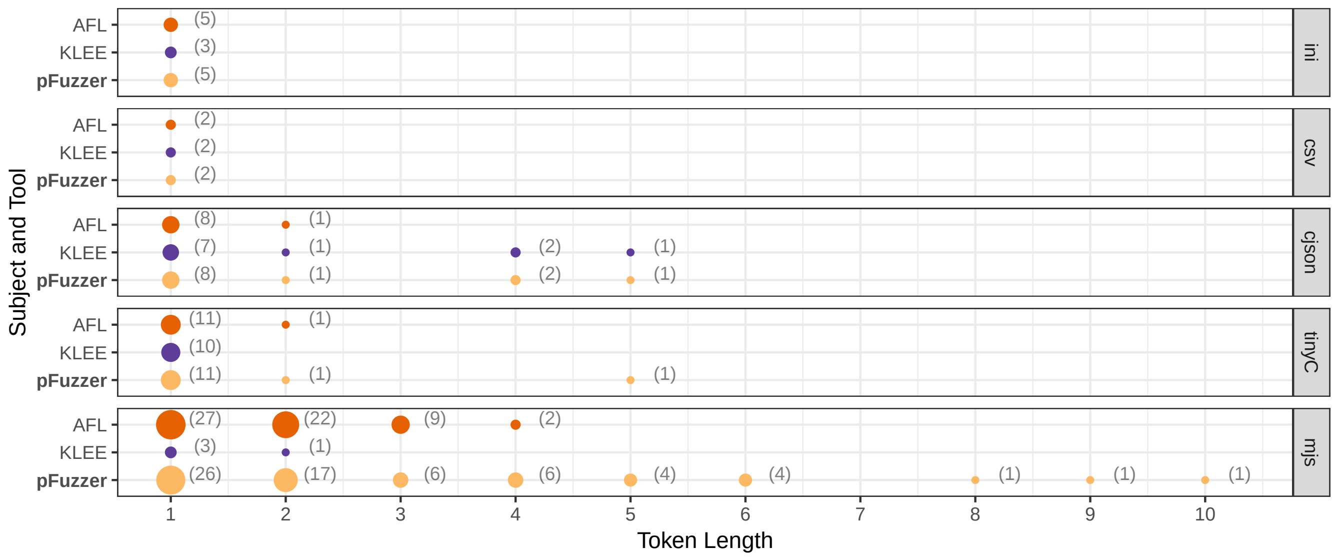

Figure 3. The number of tokens generated grouped by token length for INI, CSV, JSON, TINYC and MJS.

Being able to produce a `for` deserves a special recommendation here, as a valid `for` loop needs the keyword `for`, an opening parenthesis, three expressions ending with a semicolon and a closing brace. As it comes to longer tokens and keywords, AFL is quickly lost, though; for instance, it cannot synthesize a valid input with a `typeof` keyword. pFUZZER synthesizes a full `typeof` input and also covers several of the longer keywords, thus also covering the code that handles those keywords.

Summing up over all subjects, we see that for *short* tokens, all tools do a good job in generating or inferring them. The exception is KLEE on MJS, which results in a lower average number:

Across all subjects, for tokens of length ≤ 3, AFL finds 91.5%, KLEE 28.7%, and pFUZZER 81.9%.

The longer the token, though, the smaller the chance of either AFL or KLEE finding it. pFUZZER, on the other hand, is able to cover such keywords and therefore also the respective code handling those keywords.

Across all subjects, for tokens of length > 3, AFL finds 5%, KLEE 7.5%, and pFUZZER 52.5%.

This is the central result of our evaluation: *Only parser-directed fuzzing is able to detect longer tokens and keywords in the input languages of our test subjects.* By extension, this means that *only pFUZZER can construct inputs that involve these keywords,* and *that only pFUZZER can generate tests that cover the features associated with these keywords.*

At this point, one may ask: Why only 52.5% and not 100%? The abstract answer is that inferring an input language from a program is an instance of the halting problem, and thus undecidable in the first place. The more concrete answer is that our more complex subjects such as TINYC and MJS make use of tokenization, which breaks explicit data flow, and which is on our list of challenges to address (Section 7). The final and pragmatic answer is that a "better" technique that now exists is better than an "even better" technique that does not yet exist—in particular if it can pave the way towards this "even better" technique.

6 Related Work

6.1 Fuzzing Techniques

Fuzzing was introduced by Miller et al. [26] for evaluating the robustness of UNIX utilities against unexpected inputs. The main difference between fuzzing and other *blackbox* test generation methods is that fuzzing relies on very weak oracles—checking only for crashes or hangs, which lets it explore the input space automatically. Miller used a simple random character generator to generate inputs to the programs under test, varying both input length and content randomly. However, this kind of simple fuzzing is ineffective against programs that expect structured inputs such as interpreters and compilers. Hence practitioners rely on different techniques to generate syntactically valid (or near valid) inputs. These techniques are commonly classified as *blackbox* techniques, *whitebox* techniques, and their *hybrids* [24].

6.1.1 Blackbox Fuzzing

Blackbox fuzzing techniques generate input data given some *information about the data,* ignoring the program under test. These include *mutational* approaches and *generational* approaches [27]. *Mutational* fuzzing approaches [37] start with a few sample inputs, and rely on simple mutations such as byte swaps or character insertion to generate new inputs. These techniques hope to explore the input regions close to the mutation, by maintaining the global structure but varying the local structure sufficiently. These mutations may be evolved based on some fitness function to ensure better exploration of input space.

Generational approaches rely on some formal input specification such as the input grammar or a data model to generate valid inputs [34]. Test case generation using grammars has been used for compiler testing [4, 17] from very early on. One of the problems with simple generational approaches has been the unavailability of input models. Another is that simple input models may not adequately capture all constraints. A promising approach is to learn the model [1] and constraints [32] from a sample corpus.

6.1.2 Whitebox Fuzzing

Whitebox fuzzing techniques make use of *program code.* Whitebox fuzzing techniques are again classified into those that rely on *dynamic execution* of program, and those that rely on *symbolic execution* of the program.

In the *dynamic execution* approach, the program is typically instrumented such that the path taken by an execution for a sample input can be observed. The sample input is then mutated [37], and mutations that take previously unseen paths are given higher preference for further exploration. The input mutation may be guided [8] using dynamic taints [2, 11, 12], and constraint negation of executed paths.

In the *symbolic execution* approach, the program is symbolically executed to infer constraints on inputs; a constraint solver then generates inputs satisfying them. SAGE [13], was one of the first *whitebox* fuzzers, and uses symbolic execution and constraint solving to generate outputs for fuzzing (Sage also relies on seed samples and their executions [14] for the initial collection of constraints). Another is *Mayhem* [7] by Cha et al. that won the *DARPA Cyber Grand Challenge* in 2016. Mayhem prioritizes the paths where memory access to symbolic addresses that can be influenced by user input is identified, or symbolic instruction pointers are detected. A similar approach is *concolic execution* and *hybrid concolic execution* which relies on fuzzing for initial exploration but shifts to symbolic execution for resolution of checksums and magic bytes [25, 36].

6.1.3 Hybrid Fuzzing

With Hybrid fuzzing techniques, researchers try to infer a program model [31] or input grammar from either observing the behavior of the program on multiple inputs, using formal approaches [1], machine learning based on previously available corpus [9, 15], or observing and summarizing the program execution [19].

The main difference we have with these tools is that they rely on a previously existing corpus of valid inputs for the model generation. The problem with this approach is that, the available inputs often encode assumptions about the program behavior which need be neither correct nor complete. It is precisely these assumptions that we seek to avoid with parser-directed fuzzing.

We note that there has been a number of fuzzers that specifically target parsers [16], especially interpreters and compilers [18, 35], and special purpose parsers can often incorporate domain knowledge to obtain better results.

6.2 Specific Fuzzing Tools

We compare with specific state-of-the-art competitors.

Driller [30] attempts to combine the advantages of symbolic execution with those of fuzzing. It relies on fuzzing to explore the input space initially, but switches to symbolic execution when the fuzzer stops making progress—typically, because it needs to satisfy input predicates such as magic bytes.

Driller uses symbolic execution and is vulnerable to the combinatorial path explosion [8] when trying to reach deep program states. Since pFuzzer does not use symbolic execution, pFuzzer does not suffer from this problem and may be able to achieve deeper coverage.

VUzzer [28] relies on the key observation that fuzzing can be aided by a feedback loop from control and data flow application features. VUzzer uses taint analysis to infer type of data and offsets in input, which relates to branches. These specific offsets and values are prioritized for mutation. One of the problems with VUzzer is that the position of magic bytes is fixed – the offsets are inferred statically. That is, VUzzer can not deal with magic bytes whose location may be different in different input files [8].

VUzzer is similar to pFuzzer in that it tracks both taint information and character comparisons. However, unlike VUzzer, the taint information and character comparison information in pFuzzer is collected and used dynamically. Secondly, unlike VUzzer, pFuzzer does not require an initial set of seed inputs to operate.

Steelix [24] is another mutation based fuzzer. It improves upon the state of the art by adding a comparison progress feedback. The comparison progress can avoid problems with multi-byte string comparisons by providing progress information on the string being composed.

pFuzzer uses an approach similar to Steelix's comparison progress. The main difference from Steelix is that the mutations for Steelix is primarily random, with *local exhaustive mutations* for solving magic bytes applied only if magic bytes are found. pFuzzer on the other hand, uses comparisons as the main driver. The mutations always occur at the last index where the comparison failed.

AFL [37] is a coverage-guided mutational fuzzer that can achieve high throughput. In the absence of instrumentation, it is also able to gracefully degrade to naive fuzzing using blind exploration of input space. The effectiveness of AFL depends highly on initial *seed inputs,* which it uses to explore the input subspace near the given samples.

AFL requires less overhead than pFuzzer since the only instrumentation it requires is tracking coverage. Nonetheless, pFuzzer improves upon AFL in numerous ways. First, pFuzzer leverages taint tracking and character comparison tracking to bypass the requirement of initial seed samples. Second, the mutations produced by pFuzzer are driven by parser behavior, compared to the blind mutations by AFL.

AFL-CTP [23] is a transformation pass that converts calls of *strcmp()* and *memcmp()* to nested *if*-statements to help the coverage guidance of AFL. For *strcmp()* and *memcmp()* AFL gets no coverage feedback until they report a match. Splitting the comparison in *if*-statements makes it possible to achieve new coverage on each matching character or byte.

The approach only transforms calls with argument size known at compile time (i.e. one argument must be a string literal for *strcmp()* respectively the number of compared bytes must be a constant for *memcmp()*). In our subjects most of the string comparisons do not fulfill this requirement, but even if it was possible to drop this condition, in many parsers code is heavily reused, i.e. different keywords are parsed with the same code and the same comparison function is called for different keywords. Hence, prefixes of different keywords are indistinguishable regarding coverage: the prefix *wh* of the *while* keyword would produce the same coverage as the prefix *fo* of the *for* keyword. pFuzzer, on the other hand, monitors the calls to *strcmp()* dynamically and therefore recognizes the different comparisons made, hence it is able to find and use the different keywords. If indeed it is possible to transform *strcmp()* and *memcmp()* in such a way that for different keywords AFL recognizes new coverage, AFL might be able to achieve similar results in terms of token coverage as pFuzzer does at the moment.

Angora [8] is a mutation based fuzzer that improves upon AFL. It incorporates byte-level taint tracking, context-sensitive branch coverage (that can distinguish different program states reached), and type and shape inference and uses *search based gradient descent* for help with checksum resolutions. Of all the fuzzers, Angora is closest in approach to pFuzzer.

We believe that Angora's technique is relatively heavy-weight in comparison to pFuzzer. This can be seen in [8] Section 5.6, where the authors say that "Angora runs taint tracking once for each input, and then mutates the input and runs the program many times without taint tracking". Each time a new branch is detected with an input, Angora runs the taint tracking, along with possibly thousands of runs without taint tracking again until it hits on a new branch (Algorithm 1, Line 16 [8]).

pFuzzer improves upon Angora in multiple ways.[8] First, unlike Angora, which tries to solve the complete path constraints, pFuzzer is only interested in generating inputs that can pass a specified parser. Further, pFuzzer employs light weight analysis of character comparisons to determine the mutation, while Angora uses search based on gradient descent, which is more heavy weight. Further, pFuzzer uses a simple (mostly) monotonic increase in the input string length, which means that only the last few (often a single) characters need to be mutated for further exploration. This reduces the computational expenditure significantly.

At the end of this comparison, let us point out that while each approach has their specific strengths and weaknesses, they may well complement each other. A pragmatic approach could be to start fuzzing with a fast lexical fuzzer such as AFL, continue with syntactic fuzzing such as pFuzzer, and solve remaining semantic constraints with symbolic analysis.

7 Limitations and Future Work

While our approach surpasses the state of the art, it leaves lots of potential for further improvement, addressing the challenges of even more complex languages. Our future work will address the following topics:

7.1 Table-Driven Parsers

Our current implementation is limited to recursive-descent parsers. The coverage metric will not work on table-driven parsers out of the box as such a parser defines its state based on the table it reads rather the code it is currently executing. This is an obvious, yet not severe limitation. First, the coverage metric still works as a general guidance—instead of code coverage, one could implement coverage of table elements. Thus, the general search heuristic would still work especially as the implicit paths and character comparisons do also exist in a table driven parser. Second, recursive descent parsers, especially those produced by hand are one of the most common types of parsers. A cursory look at the top 17 programming languages in Github [29] shows that 80% of them have recursive descent parsers. Finally, tables for a table driven parser are almost always generated using parser generators such as *Yacc* and *Antlr*. Hence, a grammar is already available, and one can thus apply *grammar-based fuzzing* for superior coverage of input features. Table driven parsers thus make a formidable challenge, but more from an intellectual viewpoint than a practical one.

7.2 Tokenization

A second limitation is *loss of taint information during tokenization*. Programs that accept complex input languages often have a *lexical* phase where a light weight parser that accepts a *regular* language is used to identify logical boundaries in the incoming character stream, and split the character stream into tokens. This can happen either before the actual parsing happens or, as in tinyC and mjs, interleaved with the parsing, where the lexer is activated each time a new token is required by the parser.

[8] Angora is unavailable at this time, and a number of its technical details can only be guessed at.

The problem with tokenization is that tokens represent a break in data flow. For example, consider this fragment that generates a token LPAREN:

```
if (next_char() == '(')
  return LPAREN;
```

The token LPAREN is generated without a *direct* data flow from the character *"("* to the token. As our prototype relies on direct taint flow, it is unable to correctly taint LPAREN.

Our coverage metrics and other heuristics circumvent this problem to some extent. Each time a token is accepted that we have not seen before, a new branch is covered in the parser. Let us assume we already saw an opening parenthesis, e.g. in an input containing an `if()`-statement. If we now want to generate a `while()`-statement and already have the prefix `while`, putting an opening parenthesis after the `while` will not cover new branches in the tokenizer. Still, in the function parsing the `while` statement we would now see that the branch checking if the opening parenthesis is present is covered. Thus, based on the increased coverage, we would use the prefix `while(` for further exploration, and this is how pFuzzer can still generate complex inputs.

We are currently investigating means to identify *typical tokenization patterns* to propagate taint information even in the presence of implicit data flow to tokens, such that we can recover the concrete character comparisons we need.

7.3 Semantic Restrictions

Our approach focuses on *syntactic* properties. Still, many nontrivial input languages have *semantic* restrictions as well—in many programming languages, it is necessary to declare named resources before their use. These are context-sensitive features and are usually verified *after the parsing phase*. However, our technique has no notion of a *delayed constraint*. It assumes that if a character was accepted by the parser, the character is correct. Hence, the input generated, while it passes the parser, fails the semantic checks. This, to some extent, mirrors the difficulty with the lexing phase. Such semantic restrictions need to be learned throughout generation, and is one of the frontiers we want to tackle.

7.4 Recursion

Most complex input languages contain recursive structures. While parser-directed fuzzing is a reasonable technique to explore comparatively short sequences of inputs, it is inefficient to use it beyond a shallow exploration of the input features without too much recursion. For generating larger sequences, it is more efficient to rely on parser-directed fuzzing for initial exploration, use a tool to mine the grammar from the resulting sequences, and use the mined grammar for generating longer and more complex sequences that contain recursive structures. The ability to mine grammars already exist [19], and its incorporation will increase the effectiveness of our tool chain. Indeed, the stumbling block in using a tool such as AutoGram right now is the lack of valid and diverse inputs. Using a human to produce these inputs runs the risk that a human being will only produce inputs with features that they *think* the program implements. However, such mental models are based on assumptions about the program which can be incomplete or incorrect even for comparatively simple programs. Using inputs thus produced runs the risk of developing blind spots on the inputs composed using grammar based fuzzers. A similar problem occurs when one relies on pre-existing grammars (when available). These grammars often encode knowledge of a programs input at some point in time, and according to what an ideal program that implements the grammar should behave. Specifications can often change, and programs can incorporate new features not seen in the original grammar. Hence, parser-directed fuzzing, which does not rely on any such assumptions, fills an important place in automatic test generation.

8 Conclusion

In a program, only valid inputs survive the parsing stage and are able to test the actual program functionality. Yet, generating valid test inputs for parsers is a challenge. "Lexical" approaches such as traditional fuzzing fail because of the sheer improbability to generate valid inputs and keywords (while being good at testing the input rejection capability of a program's parsing stage), whereas the symbolic constraint solving of "semantic" approaches fails due to combinatorial explosion of paths. With parser-directed fuzzing, we present the first approach that, given nothing but the parser at hand, infers and covers substantial parts of input languages—up to the complexity of real programming languages.

Our approach relies on the key observation that most parsers process and compare a single character at a time. We use dynamic tainting to track the comparisons made with the last character; we fix the last character when the input is rejected; and we add new characters when the end of input is reached without error. These steps are sufficient to produce high quality inputs with tokens and keywords, outperforming state-of-the-art "lexical" fuzzers such as AFL or "semantic" fuzzers like KLEE.

Right now, nontrivial input languages are the main roadblock for effective test generation at the system level. "Syntactic" parser-directed fuzzing thus has a large potential for the future of test generation. As more and more of its limitations will be lifted (Section 7), users will be able to generate syntactically valid inputs for a large class of programs—and thus easily reach, exercise, and test program functionality in a fully automatic fashion. Once we can synthesize the simplest Java program `class C { public static void main(String args[]) {} }` from a Java compiler, syntax-directed testing will have reached its full potential.

A replication package is available at:

https://tinyurl.com/pfuzzer-replication

References

[1] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing Program Input Grammars. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 95–110. https://doi.org/10.1145/3062341.3062349

[2] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. 2012. A Taint Based Approach for Smart Fuzzing. In *International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, Washington, DC, USA, 818–825.

[3] Ben Hoyt and contributors. 2018. inih - Simple .INI file parser in C, good for embedded systems. https://github.com/benhoyt/inih. Accessed: 2018-10-25.

[4] D. L. Bird and C. U. Munoz. 1983. Automatic Generation of Random Self-checking Test Cases. *IBM Systems Journal* 22, 3 (Sept. 1983), 229–245. https://doi.org/10.1147/sj.223.0229

[5] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *USENIX conference on Operating systems design and implementation*, Vol. 8. 209–224.

[6] Cesanta Software. 2018. Embedded JavaScript engine for C/C++ https://mongoose-os.com. https://github.com/cesanta/mjs. Accessed: 2018-06-21.

[7] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy*. IEEE, 380–394.

[8] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *IEEE Symposium on Security and Privacy*. http://arxiv.org/abs/1803.01307

[9] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 95–105.

[10] Dave Gamble and contributors. 2018. cJSON - Ultralightweight JSON parser in ANSI C. https://github.com/DaveGamble/cJSON. Accessed: 2018-10-25.

[11] Will Drewry and Tavis Ormandy. 2007. Flayer: Exposing Application Internals. In *USENIX Workshop on Offensive Technologies (WOOT '07)*. USENIX Association, Berkeley, CA, USA, Article 1, 9 pages.

[12] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based Directed Whitebox Fuzzing. In *International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 474–484. https://doi.org/10.1109/ICSE.2009.5070546

[13] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Queue* 10, 1, Article 20 (Jan. 2012), 20:20–20:27 pages.

[14] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium*, Vol. 8. 151–166.

[15] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&fuzz: Machine learning for input fuzzing. In *IEEE/ACM Automated Software Engineering*. IEEE Press, 50–59.

[16] HyungSeok Han and Sang Kil Cha. 2017. IMF: Inferred Model-based Fuzzer. In *ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 2345–2358. https://doi.org/10.1145/3133956.3134103

[17] K. V. Hanford. 1970. Automatic Generation of Test Cases. *IBM Systems Journal* 9, 4 (Dec. 1970), 242–257. https://doi.org/10.1147/sj.94.0242

[18] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments.. In *USENIX Conference on Security Symposium*. 445–458.

[19] Matthias Höschele and Andreas Zeller. 2016. Mining Input Grammars from Dynamic Taints. In *IEEE/ACM Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 720–725. https://doi.org/10.1145/2970276.2970321

[20] JamesRamm and contributors. 2018. csv_parser - C library for parsing CSV files. https://github.com/JamesRamm/csv_parser. Accessed: 2018-10-25.

[21] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. 2011. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*.

[22] Kartik Talwar. 2018. Tiny-C Compiler. https://gist.github.com/KartikTalwar/3095780. Accessed: 2018-10-25.

[23] laf intel. 2016. Circumventing Fuzzing Roadblocks with Compiler Transformations. https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/. Accessed: 2019-03-12.

[24] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: program-state based binary fuzzing. In *ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ACM, 627–637.

[25] Rupak Majumdar and Koushik Sen. 2007. Hybrid Concolic Testing. In *International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, Washington, DC, USA, 416–426. https://doi.org/10.1109/ICSE.2007.41

[26] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. In *Workshop of Parallel and Distributed Debugging*. Academic Medicine, pages ix–xxi,.

[27] Charlie Miller, Zachary NJ Peterson, et al. 2007. *Analysis of mutation and generation-based fuzzing*. Technical Report. Independent Security Evaluators.

[28] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. Vuzzer: Application-aware evolutionary fuzzing. In *Network and Distributed System Security Symposium*.

[29] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 155–165.

[30] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *Network and Distributed System Security Symposium*, Vol. 16. 1–16.

[31] Joachim Viide, Aki Helin, Marko Laakso, Pekka Pietikäinen, Mika Seppänen, Kimmo Halunen, Rauli Puuperä, and Juha Röning. 2008. Experiences with Model Inference Assisted Fuzzing. In *USENIX Workshop on Offensive Technologies (WOOT'08)*. USENIX Association, Berkeley, CA, USA, Article 2, 6 pages.

[32] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *IEEE Symposium on Security and Privacy*. IEEE, 579–594.

[33] Wikipedia. 2018. List of File Formats. https://en.wikipedia.org/wiki/List_of_file_formats. Accessed: 2018-11-14.

[34] Jingbo Yan, Yuqing Zhang, and Dingning Yang. 2013. Structurized grammar-based fuzz testing for programs with highly structured inputs. *Security and Communication Networks* 6, 11 (2013), 1319–1330.

[35] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 283–294.

[36] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Conference on Security Symposium*. USENIX Association.

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