Learning Input Tokens for Effective Fuzzing

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

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

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

ABSTRACT

Modern fuzzing tools like AFL operate at a *lexical* level: They explore the input space of tested programs one byte after another. For inputs with complex *syntactical* properties, this is very inefficient, as keywords and other tokens have to be composed one character at a time. Fuzzers thus allow to specify *dictionaries* listing possible tokens the input can be composed from; such dictionaries speed up fuzzers dramatically. Also, fuzzers make use of dynamic tainting to track input tokens and infer values that are expected in the input validation phase. Unfortunately, such tokens are usually implicitly converted to program specific values which causes a loss of the taints attached to the input data in the lexical phase.

In this paper, we present a technique to extend dynamic tainting to not only track explicit data flows but also taint implicitly converted data without suffering from taint explosion. This extension makes it possible to augment existing techniques and automatically infer a set of tokens and seed inputs for the input language of a program given nothing but the source code. Specifically targeting the lexical analysis of an input processor, our lFuzzer test generator systematically explores branches of the lexical analysis, producing a set of tokens that fully cover all decisions seen. The resulting set of tokens can be directly used as a dictionary for fuzzing. Along with the token extraction seed inputs are generated which give further fuzzing processes a *head start*. In our experiments, the lFuzzer-AFL combination achieves up to 17% more coverage on complex input formats like JSON, LISP, TINYC, and JAVASCRIPT compared to AFL.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; *Parsers*; • **Theory of computation** → *Regular languages*.

KEYWORDS

fuzzing, test input generation, parser

ACM Reference Format:
Björn Mathis, Rahul Gopinath, and Andreas Zeller. 2020. Learning Input Tokens for Effective Fuzzing. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Los Angeles/Virtual, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3395363.3397348

ISSTA '20, July 18–22, 2020, Los Angeles/Virtual, CA, USA

ACM ISBN 978-1-4503-8008-9/20/07.
https://doi.org/10.1145/3395363.3397348

1 INTRODUCTION

Fuzzing has emerged as one of the key test-generation technologies in recent years. By automatically generating millions of tests in a short time a fuzzer is able to reveal bugs in software that may have stayed undetected by manually written tests. Fully automatic state-of-the-art fuzzers like AFL [19] explore the input space bytewise and mostly randomly (with some coverage guidance). Thus, when fuzzing a program with a complex input structure, most of the generated inputs are invalid, and many features of the program cannot be covered as a random fuzzer is in general not able to produce keywords and complex structures: Generating a keyword like `while` randomly from letters has a chance of 1 in 26^5. A pure random fuzzer will need a long time to generate this keyword, let alone the structures that follow it.

Several solutions have been introduced to circumvent this problem, the most important being *coverage guidance*. Maximizing code coverage, a random fuzzer like AFL over time is able to generate inputs consisting of different tokens, generating valid prefixes that cover more and more code until a valid input is finally composed. For constrained input languages, however, this is still not sufficient: As fuzzers compose their inputs character by character, they have to determine each and every keyword in the input again and again.

The problem of finding keywords and other lexical structures can be dramatically alleviated by providing a *dictionary* of common input fragments. This allows fuzzers to compose inputs from dictionary entries—in other words, they can build inputs from *tokens* rather than individual characters, which can be highly beneficial for fuzzing, e.g. with AFL. Even though AFL puts tokens from its dictionary randomly together, its coverage guidance can approximate the value of a generated input. This finally leads the random generation to inputs that survive the input validation stage and reach actual functionality—*if* a dictionary was supplied by the user.

In this paper, we propose a new approach that takes a program and *automatically* and *without seeds* extracts the tokens of its input language using dynamic tainting of implicit data transformations, to produce a dictionary and seed inputs to speed up fuzzing. The **key idea** of our approach, sketched in Fig. 1, is to systematically create inputs that cover all branches of a *tokenizer*—that is, a program part that composes characters into tokens. Our approach extends our earlier work [12], especially our tool pFuzzer. To this end, we dynamically track the *comparisons* made on input characters and tokens. The comparisons on input characters are easy to satisfy, as a tokenizer usually compares one or more input characters against a predefined set of keywords or special characters. It returns or stores a constant value (the token) based on those comparisons, which is again used in comparisons against other tokens in the parser. Hence, the dynamic tainting needs to be able to follow taints from

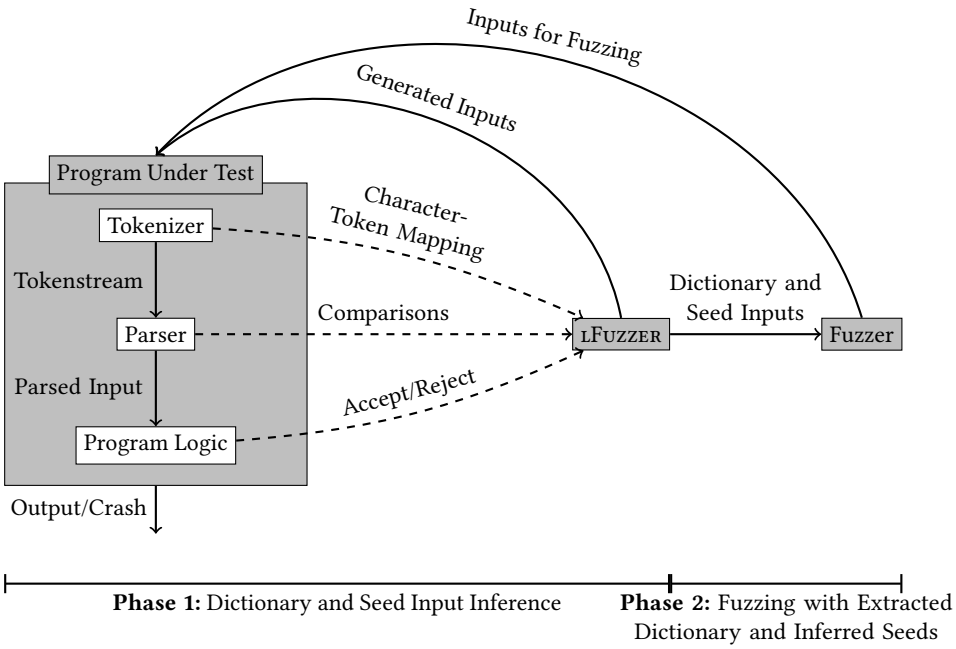

Figure 1: How lFuzzer works. In Phase 1, the learning phase, a dictionary and seed inputs are extracted. This is done as follows: lFuzzer generates an input and runs the program on this input. The tokenizer creates a token stream from the input (or the parser requests token after token from the tokenizer) and lFuzzer learns the mapping of each input character to the token it is converted to. The parser uses the tokens to parse the input, lFuzzer extracts the token comparisons made to generate the next input. If neither tokenizer nor parser reject the input, the program logic either outputs a result or crashes. In Phase 2, the *extracted dictionary* and *learned seed inputs* are used by a fuzzer to fuzz the program under test.

the input characters to the returned tokens to make use of the token comparisons in the parsing phase. Also, comparisons in the tokenizer that make up tokens can be used in fuzzing dictionaries.

In addition, we can use the token comparisons from the parser to create a *valid input.* We rely on the fact that a parser processes tokens one after another, comparing each input token against all valid tokens at this position before rejecting an input. We start with a random token which is likely rejected, extract the comparisons made on the token and replace it with one of the compared tokens, passing the first token comparison. We append new tokens until all token comparisons are passed. This input can be given as a seed to a fuzzer. Now we can create another seed input or start a fuzzer like AFL with the *extracted dictionary* and the *generated seed inputs.*

To the best of our knowledge, this is the first approach to *systematically taint, track and extract input tokens from a program under test* for the purpose of making test generation more efficient and more effective. Our token extraction approach solely relies on the comparisons made on input characters and tokens, making it easy to understand, implement, and extend for future research.

Our approach is effective. Our **evaluation** on six subjects ranging from csv to JavaScript shows that the dictionaries and seed inputs generated by our approach are more effective *and* more efficient for fuzzing. Compared to AFL and pFuzzer without any information, AFL with a dictionary of the string constants from the program code and AFL with seed inputs generated by pFuzzer, AFL given our seeds and dictionaries achieves at least comparable, but in general higher coverage. As the benefits increase with growing complexity of the input language, our work opens the door towards highly efficient fuzzing of programs with complex input languages.

The remainder of the paper is organized as follows. Section 2 describes how we enable dynamic tainting of implicit data transformations and systematically explore the lexical input space of tokenizers, producing dictionaries and seeds for further fuzzing. Section 3 details the evaluation, comparing our approach against AFL and pFuzzer as described above. After discussing the related work (Section 4) on token extraction and dictionary usage, we discuss limitations and future work in Section 5, before concluding the paper in Section 6.

2 EXTRACTING INPUT TOKENS

Our goal is twofold: improving dynamic tainting by allowing implicit data conversions and using this extension for improved magic byte fuzz-blocking elimination.

First, we want to improve the precision of dynamic tainting by tracking taints on implicit data transformations. Such transformations are extensively used in input validators, more specifically in the tokenization phase of such a validator in which one or more input characters are converted to a constant value, a so called token.

Second, with this dynamic tainting extension we want to automatically generate both a dictionary and a set of seed inputs which can be used as a guidance for fuzzing using no more than the program as initial information. The key idea is to generate inputs that cover all branches of the *tokenizer*. Hence, we use a *test generator for tokenizers* to extract knowledge for another, general test generator.

Generating tests that cover tokenizers is a difficult task. On the one hand, random fuzzers like AFL fail in the presence of tokens, even if guided by coverage. On the other hand, testing approaches that solve path conditions to cover all branches are challenged by the complex path conditions in input processors [5]. Using a grammar or some other input model would dramatically increase the efficiency of fuzzing. Still, for many input formats no such model is available.[1] The technique of *parser-directed fuzzing* [12] aims to strike a compromise between the two, specifically generating inputs for *parsers* that process one element at a time. Thus, the key idea of our work is to

(1) *Extend dynamic tainting to implicit data transformations*, using them to
(2) *Apply parser-directed fuzzing specifically to tokenizers,* covering all branches and, consequently, all lexical elements (tokens) of the input language; and subsequently
(3) *Extract these tokens* as dictionaries for effective fuzzing.

We implemented this approach in a tool called lFuzzer, being able to extract dictionaries and seed inputs from C programs.

2.1 Parser-Directed Fuzzing

As already mentioned, parsing makes an intensive use of implicit data conversions while also being hard to test with state-of-the-art fuzzers. Thus, we demonstrate the effectiveness of our approach by improving parser-directed fuzzing [12]. We extend our open-source implementation by Mathis et. al to also handle implicit data conversions. Hence, we shortly sketch the general idea of pFuzzer.

Parser-directed fuzzing [12] assumes that a parser processes inputs character by character, comparing each character against all expected characters at this position. Our earlier tool pFuzzer executes valid prefixes with random extensions and uses dynamic tainting to collect the comparisons done on the input characters. The collected comparisons are then used to find a valid substitution for the random extension.

We detail the approach on the example of an arithmetic expression parser: pFuzzer starts with a random character, e.g. '*&*', given to the program. This input is rejected but not before being checked if it is a *digit* or an *opening parenthesis*—the values a valid input for the expression parser can start with. In the next step pFuzzer replaces the random character with one of the values it was compared to, e.g. the character '*1*'. The input "*1*" is accepted, but pFuzzer can now decide to append another random character, looking for larger inputs. It could append '#' to the prefix "*1*", leading to "*1#*". '#' is compared against the characters that can follow a digit: '+' and '−' and replaced by one of them, leading to "*1+*" which is rejected. So pFuzzer appends another character, runs the program, analyzes the comparisons made on this character (being again comparisons against *digits* and an *opening parenthesis*), replaces the appended character with one of them, resulting in the valid input "*1+3*".

[1] Of course, if one writes a generic parser for a well known format (like json), a grammar would be available. Nonetheless, most of the time such formats are used to transport more specific data, e.g. for json a specific set of key-value pairs might be defined to exchange data between programs. With our approach it is possible to automatically extract such specific data without the need to define it manually.

Algorithm 1 Token Taint Propagation Algorithm

```
1:  lastTaint ← None
2:  procedure PROPAGATE(Ins)
3:      if TYPE(Ins) = Comp ∧ ISTAINTED(Ins) then
4:          lastTaint ← GETTAINT(Inst)
5:      else
6:          if TYPE(Ins) ∈ {Assign, Return, Expr} ∧ HASCONSTANT(Ins) then
7:              ASSIGNTOKTAINT(Ins.getConstant, lastTaint)
8:          end if
9:      end if
10:     if TYPE(Ins) = Return then
11:         lastTaint ← None
12:     end if
13: end procedure
```

In [12], we already mentioned tokenization as a strong limitation to fuzzing parsers and hence our approach. In the presence of a tokenization the character comparisons are shifted from the parser into the tokenizer. The tokenizer though compares every input character against *all characters and keywords* known to the program while a parser only compares against *valid characters* for the respective input position. In our example every input character would be compared against *digits*, '*(*', '+', '−', and '*)*'. Thus, pFuzzer may replace a randomly guessed character with an invalid value, e.g. the '*&*' pFuzzer started with in our example might be replaced with a '*)*'. Hence, the chances for pFuzzer replacing a guessed character with an incorrect value are higher, leading to more guesses until a valid input is composed. We believe that an explicit knowledge about a program's input tokens makes fuzzing much more efficient. Thus, we detail in the following how such tokens can be tainted, extracted and used for fuzzing (e.g. in parser-directed fuzzing).

We need to solve the following problems before we can use tokens in dictionaries and for seed input generation:

(1) Linking of input characters to tokens (i.e. tainting the tokens with the taints of the originating characters).
(2) Linking tokens to their actual meanings (e.g. a token `T_WHILE` to the keyword `while`)[2].

We will do this in three steps, sketching simple algorithms to explain our core ideas:

(1) Detecting tokenization patterns and enable taint propagation to generated tokens;
(2) Separating tokenizing and parsing code; and
(3) Correcting misclassified taints on tokens.

Finally, we will also detail how tokens can be used to efficiently generate seed inputs for further fuzzing.

2.2 Propagating Token Taints

Adapting the dynamic tainting engine of pFuzzer means detecting and tainting tokens with the taints of the characters they are

[2] A necessary feature to use the token comparisons in the parsing phase for seed input generation equal to pFuzzer which relies on comparisons in the parsing phase.

derived from. Because one can imagine an infinite amount of possibilities to create a token from input characters, this detection can only be an approximation. Still, there are some general patterns that are used when tokenizing the input, which can be detected and handled by the method presented in Algorithm 1. Every tokenizer must compare one or more characters from the input explicitly against predefined values to be able to decide if a character belongs to a predefined token. After this comparison, the tokenizer should return or store a constant number – the token created from the characters. To avoid large overapproximations we restrict the "distance" between the comparison made and the generation of the token, i.e. the generation should happen in the function the character comparison is done or immediately after the function returns. Algorithm 1 implements this token detection approach.

The function PROPAGATE in Line 2 is called on every instruction that was executed during a run of the program under test. If a comparison with a tainted value is done (Line 3), we store the taint attached to the value (Line 4). The next time a constant is stored, returned, or used in an arithmetic expression (Line 6), the stored taint, flagged as a token taint, is attached to this constant (Line 7), as we assume this to be a token assignment. Such token taints are handled by the dynamic tainting engine as any other taint and are thus propagated like normal taints. If they appear in comparisons with other constants, a token comparison is reported and can then be used for generating seed inputs (cf. Section 2.5). Line 10 and Line 11 ensure that the taint stored in *lastTaint* is deleted on a function return, as tokenization rarely happens over returns.

In the following we present two different tokenization patterns and explain how they are handled by Algorithm 1:

Basic. The most basic conversion is a direct conversion, the input is compared against some expected character or a keyword and the respective token is assigned to some variable or returned:

```
void tokenize(char c) {
   if (c == '{')
      return L_BRACE;
}
```

In this case Algorithm 1 detects the comparison of *c* against '*{*' in Line 3, and will taint the constant *L_BRACE* in Line 7 as it is a returned constant value which is detected by Line 6.

Comparison Function. Similarly to the basic conversion the comparison may be implemented in a custom function. Here we first taint the return value of the function and then taint the newly created token:

```
bool isLBrace(char c) {
   return c == '{';
}
int tokenize(char c) {
   if (isLBrace(c))
      return L_BRACE;
}
```

The comparison of *c* against '*{*' is done in the method *isLBrace*, the return value of the method is tainted by the standard dynamic tainting engine (*c* is tainted, thus the result of the comparison is tainted). In the function *tokenize*, this returned value is implicitly compared against the value *false* in the *if-condition*, triggering Line 3 and Line 4, storing the taint of the returned value for later usage in *lastTaint*. On the return of *L_BRACE* the condition in Line 6 evaluates to true, hence the stored taint in *lastTaint* is used to taint the constant as in the basic case (Line 7).

Algorithm 2 Tokenization Phase Detection Algorithm

```
1:  procedure DETECTTOKENIZATION(CallGraph)
2:      for node ∈ CallGraph do
3:          if HASCHARACTERCOMPARISON(node) then
4:              MARKTOKENIZE(node)
5:          end if
6:      end for
7:      while NEWNODEMARKED(CallGraph) do
8:          for node ∈ CallGraph do
9:              if ISPARENTTOKENIZE(node) then
10:                 MARKTOKENIZE(node)
11:             end if
12:         end for
13:     end while
14: end procedure
```

2.3 Detecting the Tokenization Phase

The tainting engine may over-approximate and report wrong token comparisons (e.g. due to tainting a constant that is no token which is later used in comparisons). Algorithm 2 divides the program code in tokenizing and non-tokenizing based on the comparisons on input characters that are detected by the dynamic tainting engine. We designed this algorithm under the assumptions that parsing functions are never called by tokenizing functions and tokenization and parsing is strictly divided (so no function can be both at the same time). Thus, we can filter out token comparisons that are reported in the tokenizer, reducing the set of invalid token comparisons.

Algorithm 2 starts with the dynamic *CallGraph* of the application (Line 1) which is constructed during the program execution. The nodes (= functions) are iterated (Line 2) and functions containing input character comparisons (Line 3) are marked as tokenizing (Line 4). After that, the nodes are iterated (Line 8), and for each node it is checked if any of the parents (a caller of the function) is marked as tokenizing (Line 9). If so the respective node is marked as tokenizing as well (Line 10), finally approximating tokenizing and parsing code. The result is an underapproximation of the tokenizing code to avoid marking parser functions as tokenizing. This would hinder the input generation that uses the comparisons in the parser.

2.4 Correcting Misclassified Taints

It may happen that even after filtering token comparisons from tokenizing code, some reported token comparisons are still wrong token comparisons. This happens because of the over-approximation in the tainting engine marking any constant after a comparison as a token. Most of those constant values are only used within the tokenizing code and are therefore filtered by Algorithm 2, but some of them may also appear in the parsing code, leading to noise.

With Algorithm 3 we filter this noise by calculating which token values are used in majority on a specific input index[3]. The algorithm is based on the assumption that the noise, the wrongly reported

[3]Each character from the input has a fixed index that identifies the character. A taint also has the index information to map the taint back to the characters it stems from.

Algorithm 3 Misclassified Taints Correction Algorithm

```
1:  procedure correctTokens(tokComps)
2:      majorityDict ← majorityVote(tokComps)
3:      filterByMajority(tokComps, majorityDict)
4:  end procedure
5:
6:  procedure majorityVote(tokComps)
7:      tokCounter ← dictionary()
8:      for cmp ∈ tokComps do
9:          indexValue ← tokCounter.get(cmp.idx)
10:         indexValue.increase(cmp.val, 1)
11:     end for
12:     for el ∈ tokCounter do
13:         el.Value() ← max(el.Value())
14:     end for
15:     return tokCounter
16: end procedure
```

token comparisons, only appear in a small amount while the actual token comparisons form the majority. Thus, in the procedure majorityVote (Line 6) we create a dictionary delivering for each index the most used token value. Concretely, we first iterate over all token comparisons (Line 8) and extract for each comparison the index information as well as the actual token value used (Line 9). Then, in Line 10 we count the number of token comparisons in which the token was tainted with the given index-value combination.

After all comparisons are analyzed, the algorithm evaluates the found numbers by iterating over all elements of the *tokCounter* dictionary (Line 12). Each element now contains a mapping from an index to the token values and their number of appearance during the execution. For example, we have the following comparisons:

```
(index: 5, tokenvalue: 10)
(index: 5, tokenvalue: 10)
(index: 5, tokenvalue: 1234)
(index: 5, tokenvalue: 10)
```

There are four comparisons on index *5*, *three* with the token value *10* and *one* with the token value *1234*—we assume the token value *10* to be correct. Hence, in Line 13 the index *5* is mapped to the token value *10* and then all token value mappings are returned in Line 15.

Finally, in Line 3 the majority dictionary from the majorityVote function is used to filter all token comparisons on every index that have another token value than the one in the dictionary. In our example, the comparison with the value *1234* would be filtered.

2.5 Using Tokens for Seed Input Generation

Knowing the token definitions as early as possible is a crucial feature for successfully generating syntactically valid inputs. Hence, as soon as lFuzzer recognizes a new character or string in a comparison on an input character, it runs the program with the new value. For example, the first time lFuzzer finds a string comparison against *while*, it would run the program under test with the input *while* to extract the information to which token it is converted to. As the tokenizer is usually stateless, the keyword will be converted to a token and compared against all possible tokens a valid input can start with, giving us the wanted character-token mapping. With the knowledge about token comparisons and definitions we can track the comparisons in the parsing phase and generate valid inputs as efficient as pFuzzer, *even if a tokenization phase is present.*

Table 1: The subjects used for the evaluation.

Name	Accessed	Lines of Code
csvparser	2018-10-25	297
inih	2018-10-25	293
cJson	2018-10-25	2,483
lisp	2019-03-19	2741
tinyC	2018-10-25	191
mjs	2018-06-21	10,920

3 EVALUATION

3.1 Setup

We stick to the test subjects used in [12][4]; details are listed in Table 1. The input languages range from simple formats as csv [9], ini [2], and json [4] up to complex formats like lisp [10], C (tinyC) [11], and JavaScript (mjs) [3]. As only two of the original subjects (mjs and tinyC) use tokenization[5] we decided to incorporate another complex subject with tokenization: a lisp interpreter.

For the evaluation we run lFuzzer in combination with AFL, i.e. we create a dictionary and a set of seed inputs with lFuzzer and then let AFL fuzz the subjects with this information. As baseline we compare against AFL[6], run in two different modes. First, we run AFL with no dictionary and one test containing one whitespace as a seed input (since AFL requires a correct test to start fuzzing). Second, we run AFL given the set of strings extracted from the program under test, using the same seed as before. For getting those strings we first compiled the subject into a human readable bitcode format and then extracted the string literals by iterating over the global values of the bitcode file and writing the global strings to the AFL dictionary.[7] This delivers all string literals from the source code. Furthermore, to show that our changes made to pFuzzer in lFuzzer actually improves its fuzzing capabilities, we compare against pFuzzer in two different modes: first, pFuzzer is run until timeout, second, pFuzzer is used to extract seed inputs, then AFL is run until timeout with the extracted inputs. As pFuzzer does not extract any tokens, AFL is run without a dictionary.

The evaluation was run for 24 hours per subject (excluding the static instrumentation and compilation time); the token learning and seed generation of lFuzzer is included in the 24 hours. The experiments were repeated *four* times to adhere to the non-determinism of all tools and were run on an Ubuntu 14.04.5 docker container with 3.3 GHz Intel processors, no tool was set up to use parallelization. Due to technical restrictions on our machine, AFL was run with AFL_SKIP_CPUFREQ enabled. We do **not provide**

[4] As in [12], the subjects are set up to read from the standard input (such that AFL can fuzz them), and to abort parsing on the first error with a non-zero exit code. mjs and lisp are changed such that semantic failures do not lead to a non-zero exit code.
[5] We still use the other subjects in the evaluation to show that lFuzzer does not harm the fuzzing process if no tokenization phase is available.
[6] As we assume nothing but the program as input to the fuzzer (no manual information like seed inputs or a dictionary), we only use fuzzers that meet this requirement.
[7] lFuzzer only requires LLVM bitcode to perform its analysis, so we decided to give AFL_Dict the same level of abstraction.

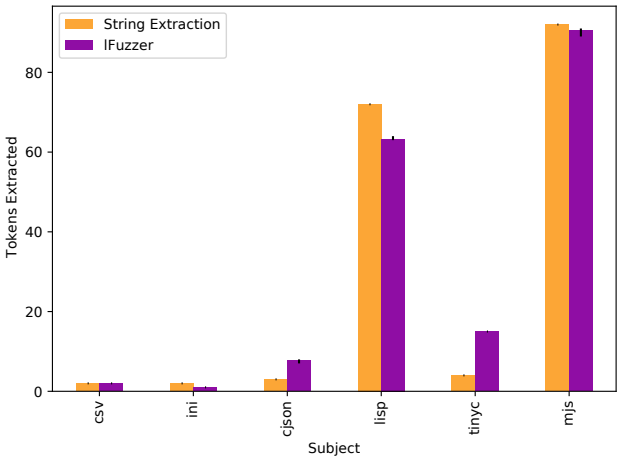

Figure 2: The number of valid tokens extracted per method and subject with error lines showing the minimal and maximal number of valid tokens extracted over all runs.

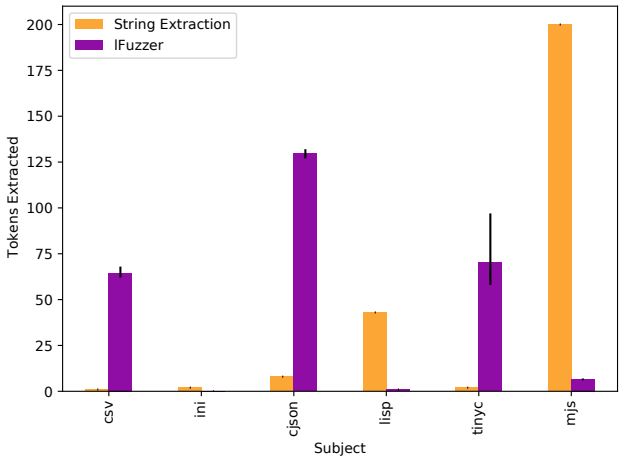

Figure 3: The number of non-tokens (strings that are no tokens) extracted per method and subject with error lines showing the minimal and maximal number of non-tokens extracted over all runs.

any seed inputs to any fuzzer as we want to evaluate the input generation capabilities without additional knowledge.

To show the effectiveness of lFuzzer we evaluate the tools on three aspects:

Token Extraction. First, we show that on programs with complex input formats lFuzzer extracts tokens with higher precision and recall compared to naive string extraction.

Code Coverage. Second, we prove that our dictionary and seed generation improves coverage when fuzzing with AFL.

Tokens Used. Third, we look at how many tokens are actually used in the test inputs produced by any of the tools.

During our experiments we found out that the instrumented version of lisp produced by AFL has a bug resulting in segmentation faults. Thus, we had to exclude seed tests produced by lFuzzer and pFuzzer that start with " (# " as otherwise AFL would not start.

3.2 Tokens Extracted

Fig. 2 shows how many valid tokens were found with the static string extraction and the active token learning of lFuzzer. As pFuzzer does not extract any tokens it will be omitted in this section. At first, one might think all tokens of a subject can be found with string extraction, but the results for cJson and tinyC show another picture: lFuzzer finds more tokens than the string extraction. The missing keywords are single character tokens, e.g. a semicolon in tinyC as those are not present anymore in the bitcode. They are character constants in the original source code and as such compiled to integer constants in the bitcode.

For mjs and lisp, the picture changes: most of the existing tokens are present as strings in the code. lFuzzer misses some of them due to its dynamic token extraction—a token can only be found if it is seen during the learning phase.

When looking at the wrongly extracted tokens in Fig. 3, we first and foremost see that those subjects that do not have a tokenization phase tend to cause lFuzzer to report wrong tokens. We assume that some part of the code looks like a tokenization but accepts any character combination from the input, causing false positives. For tinyC, a lookahead causes a correctly detected token to be appended by other characters that followed the token while lFuzzer was running, resulting in some non-tokens being reported. Our token recognizer is designed to combine all characters to a token that were accessed between two token usages. In the case of tinyC more characters where accessed between two token usages than actually belonged to the token, hence this lookahead caused random characters to be appended to the extracted token values. This mostly happens for values with undefined length like variable identifiers or numbers as they need to be parsed character by character until an invalid character is detected (hence it is used in a comparison but not part of the token).

For mjs and lisp, the number of wrong tokens is very low compared to the string extraction method. The reason for this is twofold. First, programs with a more complex input format usually also contain a better error handling, trying to give the user a profound hint on why an erroneous input is actually invalid. Thus, different error messages have to be embedded in the code resulting in many different strings that are no valid tokens of the input language. Second, those subjects have a tokenization phase, hence lFuzzer is actually able to find and extract tokens and differentiate between actual tokenization code and the code that looks like such but is not.[8]

Table 2 shows the precision and recall of lFuzzer compared to naive string extraction on subjects with a tokenization phase (tinyC, lisp, mjs). The precision of lFuzzer is 27.7% higher compared to the precision of plain string extraction as our approach does not suffer from extracting strings that are not used as tokens but for example for error handling and user communication. Surprisingly though, the recall is 0.3% higher as well, coming from

[8] lFuzzer tries to find the tokenization part of the code which works well if there is a tokenization phase but may lead to false positives if no tokenizing code is present.

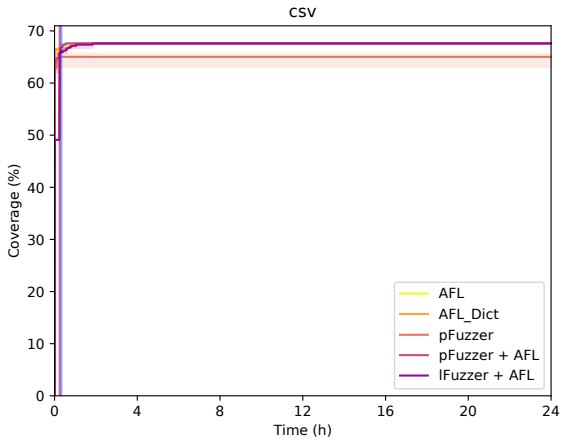

Figure 4: Average, min- and maximum coverage for CSV. The red/blue vertical area indicates when LFUZZER/PFUZZER handed over to AFL including a solid line for the average.

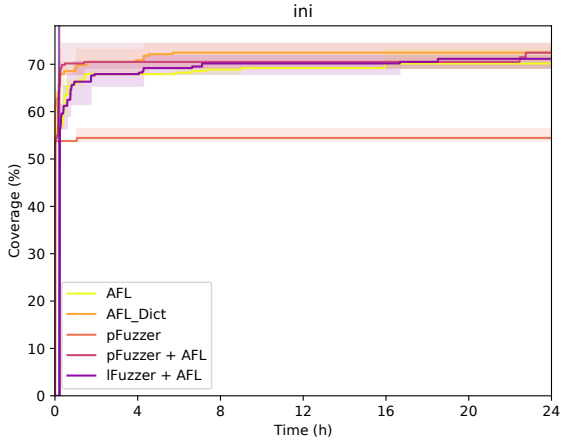

Figure 5: Average, min- and maximum coverage for INI. The red/blue vertical area indicates when LFUZZER/PFUZZER handed over to AFL including a solid line for the average.

single character tokens that are compiled to integer constants in the LLVM bitcode and are thus not extracted.

LFUZZER has a 27.7% higher precision and 0.3% higher recall regarding token extraction on subjects with a tokenization phase.

3.3 Coverage

In the following we use branch coverage achieved by the syntactically valid inputs each tool generated.

CSV and INI. In Fig. 4 and Fig. 5 we can see that programs with a simple input format can easily be covered by any tool. As all tools are based on random mutations and CSV as well as INI do not have complex interdependent syntactic features, the whole feature space can easily be covered. For CSV, a comma and a line break is sufficient to cover most of the code, INI's most complex input feature is a comment: arbitrary text surrounded by an opening and a closing bracket. PFUZZER misses some features and feature combinations leading to a lower coverage than AFL and LFUZZER can achieve. In combination with AFL the same coverage can be reached mainly because AFL on its own is able to achieve the coverage.

JSON. More interesting is the JSON subject which parses a much more complex input format, hence we see a slower and diverse increase in coverage over time for the different tools. The results in Fig. 6 show that almost all tools perform similarly good, with AFL_DICT having the most coverage (20.2%, compared to PFUZZER + AFL having 20.1%, LFUZZER having 19.9%, AFL having 18.4%, and PFUZZER having 14.6%). The similar results can be explained as follows: AFL_DICT has knowledge about the few existing keywords in CJSON, thus it is able to cover the code handling those, while also covering all the code AFL covers anyway. As CJSON does not have a tokenization phase, LFUZZER's token learning cannot come into play, resulting in falling back to using character comparisons. PFUZZER on the other hand is designed to work well on subjects with a parsing but no tokenization phase and thus covers some code very fast, in the long run though AFL is needed to cover code that PFUZZER cannot cover on its own. The low coverage all tools achieve is explained by the fact that CJSON contains generator code (code that creates a JSON string from a JSON object). In our experiments

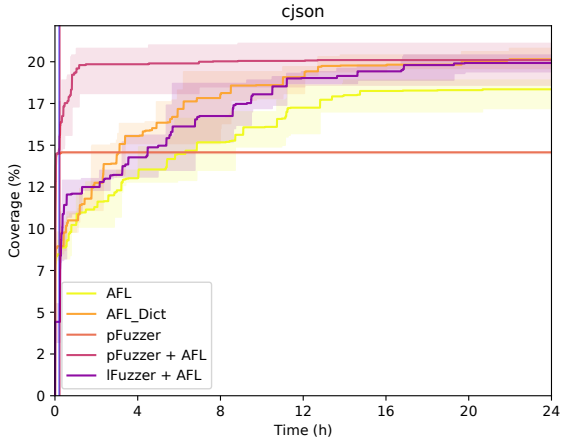

Figure 6: Average, min- and maximum coverage for CJSON. The red/blue vertical area indicates when LFUZZER/PFUZZER handed over to AFL including a solid line for the average.

Table 2: Precision and Recall on extracted strings regarding their token validity on subjects with a tokenization phase (TINYC, MJS, LISP).

Tool	Precision	Recall
String Extraction	40.7%	88.5%
LFUZZER	68.4%	88.8%

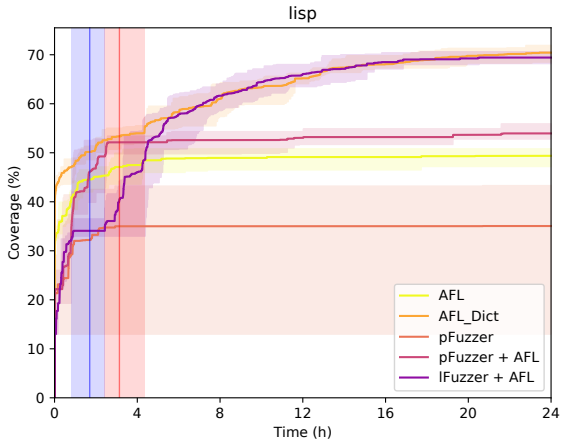

Figure 7: Average, min- and maximum coverage for LISP. The red/blue vertical area indicates when LFUZZER/PFUZZER handed over to AFL including a solid line for the average.

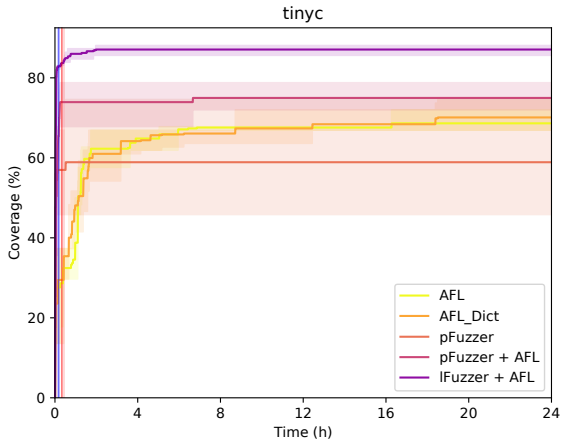

Figure 8: Average, min- and maximum coverage for TINYC. The red/blue vertical area indicates when LFUZZER/PFUZZER handed over to AFL including a solid line for the average.

we focus on the parsing part of the program, hence the generator code is not triggered by the tools. As all tools cannot cover this part of the code, the comparison is still fair.

The missing tokenization phase also causes LFUZZER to incorrectly detect arbitrary character combinations as tokens which seemingly confuses AFL, resulting in a slower increase in coverage but ultimately leading to similar coverage as AFL_DICT achieves.

LISP. LISP, still having a simple syntax but a tokenization phase, shows how the generated dictionary and seed inputs of LFUZZER increase the performance of AFL. In Fig. 7 we can see that the LFUZZER-AFL combination achieves similar coverage as AFL_DICT, having around 20% more coverage than AFL alone. For LISP, the seed inputs generated by LFUZZER are small and only cover a small part of the input space, still they are an efficient guidance for AFL to generate more complex inputs. LISP has a semantic phase which handles most of the keywords, "hiding" the actual token comparisons, making it impossible for LFUZZER to generate complex inputs. Many keywords are mapped to the same token and the token is then semantically analyzed, making it possible to extract the keywords but impossible to generate seed inputs (as the token comparisons used to generate those inputs are missing). This shows the great opportunities of our symbiotic approach—even if one of the tools alone fails to achieve coverage, the other tool is still able to address this flaw. Hence, PFUZZER alone is not able to produce a diverse set of inputs and thus achieves the worst coverage. AFL on the other hand profits from the seed inputs PFUZZER generates and covers more code compared to running alone.

TINYC. On TINYC on the other hand we can see the power of LFUZZER on its own. As shown in Fig. 8, the coverage our approach achieves is almost the maximum coverage reached throughout the fuzzing run, finally resulting in 17% more coverage than AFL_DICT. LFUZZER successfully generates inputs that cover almost all the language features of TINYC, leaving out only a few keywords or feature combinations that are then filled by AFL shortly after it

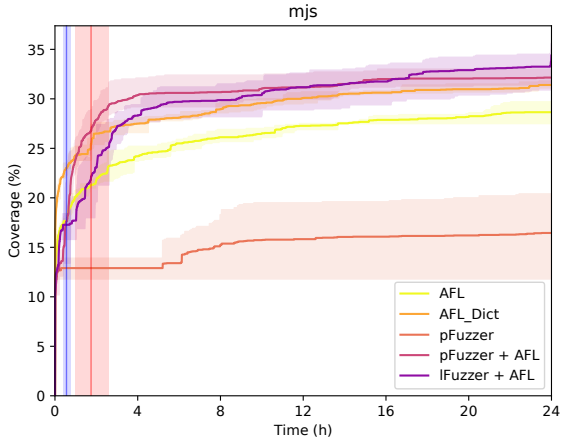

Figure 9: Average, min- and maximum coverage for MJS. The red/blue vertical area indicates when LFUZZER/PFUZZER handed over to AFL including a solid line for the average.

started fuzzing. PFUZZER on the other hand struggles with the tokenization phase of TINYC (as mentioned in Section 2.1) resulting in a lower coverage than the one LFUZZER achieves. In combination with AFL the coverage is better than for AFL alone, but still worse compared to LFUZZER as the tokens are missing to support the AFL mutations. Those results can be explained by the structure of TINYC: every keyword has only a small feature space, i.e. each keyword is handled by a few lines of code. In contrast to that, MJS for instance has many internal functions, meaning that even if a successful call to the function is generated, the code coverage is highly dependent on the function arguments.

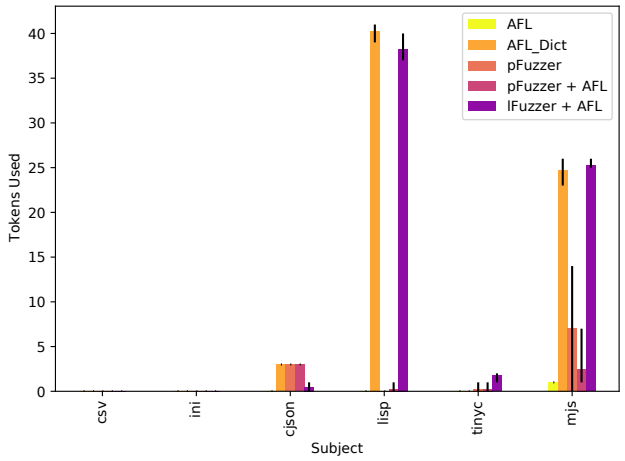

Figure 10: The number of valid tokens with size greater than 3 used per tool and subject with minimum and maximum number of tokens used over several runs.

mjs. On our most complex subject implementing an interpreter for a subset of JavaScript we can see the interplay of all components of lFuzzer in its full form. First, a precise set of tokens is learned and later given to AFL. Second, a diverse set of seed inputs is generated giving AFL a good starting point for further fuzzing. Hence, the lFuzzer-AFL combination outperforms AFL_Dict, being faster in generating coverage and achieving 3% more coverage. pFuzzer again is blocked by the tokenization phase of mjs resulting in a low overall coverage. Together with AFL the coverage gets significantly better, still the tokens provided by lFuzzer improve the AFL fuzzing process even more. In contrast to cJson, the low coverage of mjs results from the complex input format of this subject, making it hard for any approach to fully cover the subject.

On subjects with complex input structures, lFuzzer achieves on weighted average 2.3% and up to 17% more coverage than AFL_Dict.

3.4 Tokens Used

Finally, in Fig. 10 we look at the tokens with more than three characters used in the syntactically valid inputs generated by each tool. Except for tinyC, AFL_Dict uses a similar number of tokens. tinyC has only a small set of valid tokens, but those are used in complex structures, thus making it hard for a random approach like AFL to use them properly. On mjs and lisp AFL_Dict and lFuzzer use almost the same number of tokens correctly in valid inputs, a result that is also reflected in the coverage graphs: on both subjects they achieve similar results. pFuzzer, also in combination with AFL, is not able to use a large set of diverse tokens in its generated inputs, being worse on almost all subjects, only for cJson it is able to perform better than lFuzzer and equally well as AFL_Dict. On mjs the pFuzzer-AFL combination uses fewer tokens than pFuzzer alone because AFL does not generate a large diverse set of tokens on its own and pFuzzer runs shorter compared to the solo run.

As we have already seen in Section 3.2, the input dictionary for both approaches has similarly many valid tokens. Even though the number of invalid tokens in the dictionary might deviate, AFL generates such a huge number of inputs in 24 hours that eventually the right tokens are used at the correct positions, achieving new coverage and thus guiding AFL towards a valid input.

lFuzzer and AFL_Dict are both able to use a large number of tokens (weighted average of 78% and 81% of all tokens) in valid inputs.

In summary, the evaluation shows that (1) dictionaries are of great benefit for fuzzing, no matter if they are consisting of statically extracted strings or dynamically extracted tokens and (2) the combination of a precise dictionary and a set of diverse and valid seed inputs improves fuzzing on *languages with complex input formats* significantly, something only lFuzzer can achieve.

In general, the more complex the input language, the greater the benefits of automatic dictionary extraction and seed input generation as done by lFuzzer.

4 RELATED WORK

To the best of our knowledge, we are the first to dynamically taint, track and extract tokens from a program under test for more efficient fuzzing. Therefore, we will look into the research done on dictionaries and their optimization and usage for fuzzing.

4.1 pFuzzer

First and foremost, we have to mention our own work pFuzzer [12], as this work builds on the approach and research results. In pFuzzer, we were targeting parsers and generate inputs that survive the parsing stage and are able to test the program logic. This is done by tracking the comparisons done on the input characters and using them to systematically build a valid input. Each time a character is rejected it gets replaced by one of the values it was compared to, iteratively creating an input that survives more and more comparisons in the parser until it is finally accepted by the parser. We extended the pFuzzer work by improving the dynamic tainting technique to not only track character comparisons but also taint tokens that are implicitly composed from input characters; enabling the usage of token comparisons[9]. Using the extended dynamic tainting technique, lFuzzer automatically identifies and extracts tokens from the source code, using them (1) for generating a dictionary that can be used for further fuzzing and (2) for generating seed inputs by adapting their iterative input creation technique but lifting it to the token comparison level.

4.2 Learning Input Structure

Maybe the closest approach to ours is the one by Shastry et al. [16], statically inferring a dictionary from the source code. They apply backward slicing and control-flow graph analysis to find tokens and token conjunctions that can be used in a dictionary to improve fuzzing. In contrast to their approach we are using dynamic program analysis which is more robust to unusual code patterns. While

[9] In [12], we explicitly mentioned token conversions as a limitation, saying that this prevents pFuzzer from testing complex input processors efficiently.

Shastry et al. need heuristics to find the comparisons and values for their analysis, we can simply observe all comparisons in the program with dynamic tainting and report the comparison values. Static code analysis may miss dynamic code features (like an array which is filled with keywords in a loop), which may lead to omitted tokens. A keyword detection algorithm might be implemented as follows:

```
// tokenorder: T_WHILE, T_IF, T_UNDEF
char* kwds[] =  {"while", "if", null};
bool isKeyword(char* c) {
   int sym = 0;
   while (kwds[sym] != 0
      && strcmp(c, kw[sym]) != 0) {
      sym++;
   }
   return sym;
}
```

The keywords array could also be filled dynamically. For a static approach it is very hard to extract the keywords from the array. For a dynamic approach, no matter how the array is initialized, the *strcmp()* function will be called with all values in the array (assuming no comparison matches), thus making it easy to extract the keywords for a dictionary construction. Furthermore, *our approach also provides a set of seed inputs that is used to give the fuzzer a head start for fuzzing* by providing a set of valid syntactic structures of the input format that can be used for recombination and mutation.

Different approaches have been used to learn inputs or their structure and improve fuzzing with the gained knowledge. Höschele et al. [8] presented an approach to learn the input format in form of a context-free grammar from the program under test. They use a set of valid seed inputs to explore the program execution and infer a grammar based on the program structure and the consumption of input characters during parsing. Similar to our approach they are using the structural information of a program to gain input format knowledge; but first, *we are not relying on seed inputs, we generate them* and second, we do not extract the full input format but only the tokens used by the program. A combination of both approaches might be beneficial though, as detailed in Section 5.3.

Godefroid et al. [6] apply recurrent neural networks to learn the statistical distribution of input elements from a large corpus of valid inputs and then generate new inputs with the neural network. In contrast to our approach, they do not extract the input elements explicitly but encode the knowledge in a neural network, making it not accessible out of the box for existing fuzzing techniques. Furthermore, the corpus of inputs to learn from needs to be large to train the neural network while *our approach extracts tokens and seed inputs with having nothing more than the program under test.*

4.3 Selecting and Using Seeds

With Redqueen, Aschermann et al. [1] presented an approach which made it possible to circumvent different fuzzing roadblocks, among others magic bytes. In general they rely on a similar approach as the authors of Vuzzer [14], observing the control and data flow of an application and finding parts of the input that belong to branching conditions. Hence, these tools which rely on dynamic tainting can make use of our improved tainting framework to also observe token comparisons. We lift the token comparisons back to the character comparisons they represent enabling the magic byte solving to also work beyond the tokenizer. With a feedback loop portions of the input are gradually replaced with different characters until the branching condition switches and a new branch is taken. This is similar to our seed generation technique, but we are explicitly tracking the data flow, even beyond tokenization, and are replacing rejected input values with the values they were compared to, even if the comparison was done on the token level. Our approach systematically constructs diverse inputs that survive the parsing stage and test the program logic.

Several works focus on the problem of *seed input selection.* Wang et al. [18] generate seeds via analyzing the corpus to learn a probabilistic context-sensitive grammar and using this grammar together with mutations to create a set of seeds that cover the least used features from the original seed set. Rebert et al. [15] looked at different seed selection strategies and evaluated them on different subjects to find out how seed selection influences the result of the fuzzing session. They found out that seed selection can actually help improving the fuzzing performance. In any case, seed selection assumes seeds to be present to select from. In contrast, our approach creates a set of seed files, not needing any starting input. Still, the seeds we currently generate are not optimized in any way, hence seed selection might further increase the fuzzing performance.

5 LIMITATIONS AND FUTURE WORK

As shown in the evaluation, our approach is able to taint and track tokens during program execution which improves fuzzing significantly on subjects with complex input structures. Still, some limitations remain to be solved in future work; we list them in the following and provide ideas for solutions:

5.1 Parsing Style

Similar to [12], our technique is limited to recursive-descent parsers, relying on some assumptions: First, the program under test needs to have a tokenization phase, meaning that there is a part in the code which consecutively compares slices of the input against predefined characters and keywords and forwards the resulting tokens one after another to the parser. As this is the textbook approach to writing input processors for complex input formats, we believe that most of the handwritten parsers are designed like this. Second, we rely on dynamic tainting to track the input characters and the comparisons made on them throughout the program execution.

Other parsing styles like table-driven parsing, the common alternative to recursive-descent parsing, have a fundamentally different structure which we are not able to analyze at the moment. It might be possible to adapt the techniques presented in this paper to other parsing styles. Still, it is questionable if this is beneficial for the following reasons: First, 80% of the top 17 programming languages on Github are recursive-descent parsers [12] (with Clang [17] and GCC [13] being the most famous ones), hence only 20% are implemented differently. Second, table driven parsers are usually not manually written but generated from a machine readable grammar. Hence, one can apply *grammar-based fuzzing* which will be superior to character-based fuzzing.

5.2 Detecting Patterns

C is a language that is known for its freedom. A programmer can solve a problem in many different ways, some of them might be considered as straight-forward while others might be more specialized. The creation of a tokenizer is not excluded from this. While we are confident that we covered the typical tokenization patterns it might happen that we miss the generation of specific tokens or even the generation of all tokens. Our evaluation has shown that most input parsers are implemented close to the textbook approach, making it possible for us to extract the tokens we want.

5.3 Extracting Input Models

Tokens are not only valuable to our previous work pFuzzer [12], they can also be beneficial when combined with *grammar learning techniques*. AutoGram [8] and Mimid [7] implement approaches to infer context-free grammars from a program under test, tracking how individual input characters are processed within the program; AutoGram focuses on data flows, whereas Mimid uses control flow instead. The extracted tokens might serve as a great base for both approaches to construct terminals in the grammar. We would lift the minimal building blocks for the grammar from single characters to full tokens, making it easier to construct a grammar.

6 CONCLUSION

Fuzzing is one of the key technologies for software testing, currently experiencing a renaissance in research and industry. Improved methods made it possible to automatically test a wide variety of programs. Testing the actual functionality of programs with complex input formats though is a challenge that remains to be solved until today; state-of-the-art fuzzers mostly test the input rejection capabilities of the software under test rather than the actual functionality. With our implicit-data-transformation tainting, dictionary extraction, and seed input generation methods we make it possible to help fuzzers go beyond the input validation stage and test the actual program functionality.

Our approach is based on the observation that tokenizers are in general implemented by the book: one or more input characters are compared against predefined values and if one value matches the respective token is forwarded to the parser. We can find those patterns in the program execution using them for an *enhanced taint tracking*, extract the values to *generate a dictionary*, and also *build valid and diverse seed inputs* token by token. Thus, we are able to produce a foundation for fuzzing, outperforming AFL without any information and AFL given the strings from the program as dictionary while still being fully automatic.

Even though our results are very promising, this approach just serves as a foundation to show the potential of token extraction for fuzzing. Future combinations with more sophisticated techniques like grammar learning and following grammar-based fuzzing may result in even more efficient and effective testing. With this work we want to set one more milestone on the road towards efficient and fully automatic fuzzing of programs with complex input structures.

We are determined to making our research public and reproducible. lFuzzer and all evaluation data is available as open source at the project page:

https://github.com/uds-se/lFuzzer

ACKNOWLEDGMENTS

This work was partially supported with funds from the Bosch Research Foundation in the Stifterverband (Reference: T113/33825/19). We thank Andreas Zeller's team at CISPA and the anonymous reviewers for their great feedback.

REFERENCES

[1] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/

[2] 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.

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

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

[5] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(ACM SIGPLAN Conference on Programming Language Design and Implementation)*. ACM, New York, NY, USA, 206–215.

[6] 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.

[7] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining Input Grammars from Dynamic Control Flow. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) 2020*.

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

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

[10] Justin Meiners. 2019. Embeddable lisp interpreter written in C. https://github.com/justinmeiners/lisp-interpreter. Accessed: 2019-03-19.

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

[12] 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* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 548–560. https://doi.org/10.1145/3314221.3314651

[13] Joseph Myers. 2008. New_C_Parser. http://gcc.gnu.org/wiki/New_C_Parser. Accessed: 2019-05-09.

[14] 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*.

[15] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *USENIX Conference on Security Symposium* (San Diego, CA) *(SEC'14)*. USENIX Association, Berkeley, CA, USA, 861–875. http://dl.acm.org/citation.cfm?id=2671225.2671280

[16] Bhargava Shastry, Markus Leutner, Tobias Fiebig, Kashyap Thimmaraju, Fabian Yamaguchi, Konrad Rieck, Stefan Schmid, Jean-Pierre Seifert, and Anja Feldmann. 2017. Static program analysis as a fuzzing aid. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 26–47.

[17] The Clang Team. 2019. Clang - Features and Goals. http://clang.llvm.org/features.html#unifiedparser. Accessed: 2019-05-09.

[18] 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.

[19] Michal Zalewski. 2015. afl-fuzz: making up grammar with a dictionary in hand. https://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html. Accessed: 2019-05-07.