Bohemia - A Validator for Parser Frameworks

Anish Paranjpe
School of Electrical Engineering
and Computer Science
Pennsylvania State University
State College, PA 16801
Email: anish.paranjpe@gmail.com

Gang Tan
School of Electrical Engineering
and Computer Science
Pennsylvania State University
State College, PA 16801
Email: gtan@psu.edu

***Abstract*—Parsing is ubiquitous in software projects, ranging from small command-line utilities, highly secure network clients, to large compilers. Programmers are provided with a plethora of parsing libraries to choose from. However, implementation bugs in parsing libraries allow the generation of incorrect parsers, which in turn may allow malicious inputs to crash systems or launch security exploits. In this paper we describe a lightweight validation framework called Bohemia that a parsing library developer can use as a tool in a toolkit for integration testing. The framework makes use of the concept of Equivalence Modulo Inputs (EMI) in order to generate mutated input grammars to stress test the parsing library. We also describe the result of evaluating Bohemia with a set of parsing libraries that utilize distinct parsing algorithms. During the evaluation, we found a number of bugs in those libraries. Some of those have been reported to and fixed by developers.**

***Index Terms*—Parsing, validation, equivalence modulo inputs**

I. Introduction

Parsing is ubiquitous in a variety of tools: compilers, interpreters, natural language processors, network protocols, serialization, and many others. Any sufficiently large software project involves the implementation of a parser.

However, typical parsers are not trustworthy. For instance, the Windows operating system has hundreds of parsers for parsing different file formats such as JPEG and MP3; many security-critical bugs were found in these parsers [1]. As another example, over 1,000 parser bugs have been reported for the popular suite of Mozilla products, related to the ubiquitous file formats PDF, ZIP, PNG, and JPG [2]. These bugs can lead to security exploits. An example is a recent exploit dubbed "psychic paper" [3], which allows an attacker to escape an iOS sandbox by exploiting a bug in parsers for parsing property lists of iOS apps.

Many parsers are developed by using parsing libraries that implement some parsing algorithm (SLR, LALR, Earley parsing, etc.). A user of such a library specifies an input grammar and the library generates a parser according to the grammar. This approach allows ease of implementation.

On the other hand, parsing libraries are typically validated only through a set of test cases. The tests provided as part of a parsing library's codebase often contain only grammars that were discovered as bugs by library users. There is no formal assurance that these libraries generate the correct parsers for input grammars. As a result, bugs do exist in these libraries. These bugs cause incorrect parsers being generated by the libraries. Some of these bugs are security sensitive; for example, security-sensitive bugs were previously found in Yacc [4] and Flex [5]. Since a parser is typically the first component in a complex system that takes in user input, an incorrect parser can allow malicious inputs to crash the parser in the best case; in the worst case, a malicious input may trigger the generation of wrong semantic values that harm the system.

Previous work [6]–[14] has explored formal verification to verify the correctness of parsing libraries. Formal verification provides a high degree of assurance that is often desirable in practice; at the same time, it is labor intensive and takes a long cycle to develop. Further, these verification efforts have focused on specific parsing algorithms or specific kinds of grammars (e.g., network formats). In this paper, we explore a lightweight parsing validation framework called Bohemia. It performs mutation-based integration tests to detect bugs in a parsing library. By relaxing the proof requirement, Bohemia can target a wide range of parsing libraries across different programming languages.

Bohemia employs the concept of Equivalence Modulo Inputs (EMI [15]) to generate mutations of context-free grammars to test a parsing library. A mutation is a change in the structure of the context-free grammar, e.g., in the form of addition/deletion of production rules. A mutation is performed in a way so that the original grammar and the mutated one are equivalent with respect to a set of input strings. As such, the two grammars should both successfully parse the input strings. Consequently, a parsing library that fails to produce equivalent (modulo the input strings) parsers for those two grammars must have a bug.

In a broad sense, Bohemia is special form of fuzzing. However, in contrast to past research on fuzzing that generates random inputs or perform random mutations on well-formed inputs, Bohemia performs a form of *semantic fuzzing*; it mutates inputs via equivalence-preserving mutations and finds bugs by testing if a parsing library respects the equivalence. In the related work section, we will discuss more on related work in fuzzing and its connection to Bohemia.

Overall, Bohemia makes the following contributions:

- We propose a set of grammar EMI mutation strategies for stress testing parsing libraries. They manipulate a context-

free grammar to perform systematic parser validation.
- We introduce Bohemia, a practical implementation of EMI for testing parsing libraries.
- We report our evaluation of Bohemia, which found multiple bugs in existing parser libraries.

II. Background

Common parsing algorithms. There have been many parsing algorithms proposed in the literature, including LL(k), LR(k) [16], PEG and Packrat [17], [18], Earley [19], CYK [20]–[22], parsing with derivatives [23], [24]. In general, LALR (LR), Earley, CYK, and derivative parsing libraries can handle any context-free grammars. Therefore, Bohemia targets parsing libraries that employ those algorithms. Bohemia currently does not test any LL or PEG parsing libraries since they cannot handle arbitrary context-free grammars.

Equivalence Modulo Inputs Le et al. [15] introduced the concept of Equivalence Modulo Inputs (EMI) for validating optimizing C compilers. The general idea is to mutate existing real-world code in a systematic way to produce different but equivalent variants of the original code modulo a set of test inputs. The compiler should compile the equivalent variants to equivalent executables (modulo the test inputs). Resulting executables that are not equivalent with respect to those tests imply bugs in the compiler. When applying EMI to validating compilers, equivalence between executables refers to equality in executable outputs. In the simple case when programs are side-effect-free programs, equivalence can be tested by performing a simple comparison on outputs.

Orion [15] is a realization of the EMI concept applied to optimizing C compilers. Fig. 1 shows the basic workflow of Orion. In order to perform mutation, Orion utilizes a "profile and mutate" strategy. At first Orion profiles a program P on input set I. In the profile stage it records which statements in the program P are executed. It marks all the non-executed statements as dead code. In the mutation stage, it stochastically prunes dead code. The intuition behind pruning is a simple sanity check: if a certain piece of code is not executed by I, removing that code should not change the behavior of the program with respect to I.

Le et al. [25] and Sun et al. [26] extended Orion by adding two new techniques to perform mutation. The first technique inserted new code in a dead-code region. The intuition is that, if a dead-code region never gets executed, the new code put inside the region should not change the program behavior. The second technique inserted new synthesized code in live regions of a program. Such a synthesized code snippet is side-effect-free; that is, the state of the program before execution of the code snippet and the state after the execution of the code snippet are identical.

Since a compiler has a parser as its component, Orion in fact also tests the compiler's parser. However, its mutations were geared toward identifying bugs present in the compiler's optimizer. In contrast, Bohemia's mutations target the parser.

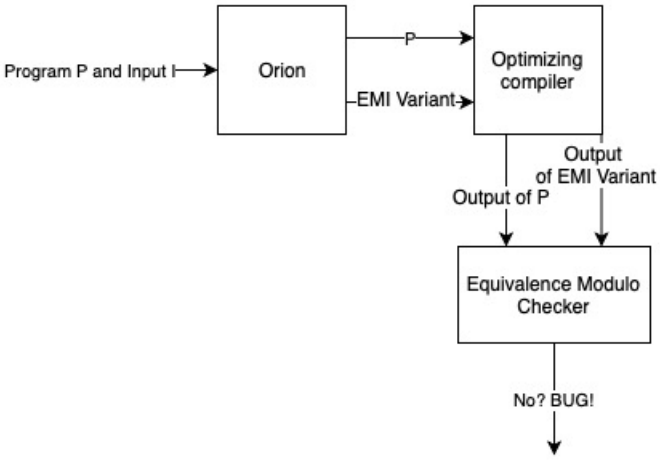

Fig. 1. Orion architecture.

III. Related Work

There has been a number of efforts of performing formal verification on parsing algorithms and libraries to obtain provable guarantees. Barthwal and Norrish [6] formalized and verified the correctness of SLR parsing. Ridge [7] verified the correctness of parser-combinator based parsing that can accommodate left recursion. Koprowski et al. [8] implemented a formally verified parser generator for Parsing Expression Grammars (PEG). The parser generator was compared with the JavaCC parser generator and was shown to be 2-3 times slower. RockSalt [9] included a verified parser for regular expression based DSL. Jourdan et al. [10] implemented a translation-validation framework to validate the correspondence between a context-free grammar and its LR(1) parser. The validator is further proven correct in an interactive theorem prover (Coq). Bernardy and Jansson [11] proved the correctness of Valiant's algorithm, an extension of CYK parsing. EverParse [14] explored how to compile a format DSL for describing network-centric formats to a library of verified monadic parser combinators. These verification efforts yielded highly trustworthy parser generators, but they were focused on specific algorithms. Further, some of these algorithms cannot take arbitrary context-free grammars [6], [8]–[10], [14] and result in slower parsers [8], [10]. In contrast, Bohemia performs lightweight mutation-based validation of existing, unverified parsing libraries that take arbitrary context-free grammars.

Researchers have also studied how to derive parsers and pretty printers from grammar specifications together with consistency proofs showing that the generated parsers and pretty printers satisfy some round-trip theorem [12], [13]. However, these systems provide a best-effort attempt in automating the proofs and may require user interaction in deriving parsers and proofs; further, Tan and Morrisett's work [12] processes regular expressions, not context-free grammars.

Another approach to finding parser bugs is fuzzing. However, randomly generating inputs, as in black-box fuzzing,

or randomly mutating well-formed inputs, as in mutation-based fuzzing, is unlikely to trigger deep semantic bugs in parsing libraries. For finding deep bugs via fuzzing, one can perform coverage-based fuzzing (sometimes referred to as *gray-box fuzzing*) or perform symbolic-execution based, *white-box fuzzing*. Coverage-based fuzzing (e.g., AFL [27]) instruments the input program to track coverage and use the coverage information as feedback to guide what input mutation should be kept for further fuzzing. In the space of white-box fuzzing, a typical example is Sage [1], which was used by Microsoft to identify a variety of bugs in parsers such as media decoders of the Windows operating system. By employing symbolic execution to inspect the internals of parsers, white-box fuzzing can identify deep bugs in parsers. Another way of finding deep bugs is through *differential testing* (e.g., [28]); in the context of parsing libraries, different testing takes two parsing libraries and feeds them the same grammar; a bug is identified if the libraries produce parsers with different behaviors. In a broad sense, Bohemia can be viewed as a form of fuzzing as it mutates grammars in an equivalence-preserving way. It offers another choice in the design space: it is capable of finding deep bugs and follows a black-box approach (and thus requires no access to the internals of parsing libraries); further, unlike differential testing, Bohemia tests one parsing library at a time.

IV. Methodology

We now describe how to apply EMI to validating parsing libraries. Inspired by Orion, we use the "profile and mutate" strategy when generating EMI variants. The mutations that we will describe are designed specifically for grammars and take different parsing algorithms into consideration.

Before proceeding, we introduce some notation:

- G: a context-free grammar. We use lowercase ascii letters for terminals of G, and uppercase names for nonterminals. Bohemia generates EMI variants of G.
- I: a set of input strings that belong to the language described by G. The size of I can be equal to or more than one.
- H: an EMI variant of G generated by the "profile and mutate" strategy.
- Mnemonic: a shorthand for representing production rules. In this shorthand, we order all production rules for a nonterminal and each rule is named using the nonterminal together with the order of the rule. Fig. 2 shows an example grammar with the mnemonics for the production rules. Since there are three production rules for `START`, we use `START_0` to stand for the first production rule, `START_1` for the second, and so on.
- Traversal (T): a list of production rules representing how an input string i is generated by a grammar G. We can generate the input string by starting from the start symbol and sequentially applying the rules on the left-most nonterminal of the current state. An example is shown in Fig. 3. We begin from rule `START_0` and get the string '`a START b`'. Then we apply `START_0` to that result. When applying the rule we expand the first nonterminal in the previously generated string of terminals and nonterminals. Hence. we get '`a a START b b`'. Then we apply `START_1` and get the final string.
- Parse Tree: a tree data structure internal to a parsing library that models the input string for a particular context-free grammar. Nodes in the parse tree have one-to-one correspondence to the symbols in the input grammar.
- Abstract Syntax Tree (AST): a condensed version of the parse tree. It is the final product of the parsing library.

```
START -> ATOM | ATOM + START
      |  ATOM - START

Mnemonics for the above grammar:
 START_0 = ATOM; START_1 = ATOM + START;
 START_2 = ATOM - START
```

Fig. 2. Examples of Mnemonics.

```
G: START -> a START b | c

i: aacbb

T: [ START_0, START_0, START_1 ]
1. START_0 => a START b
2. START_0 => a a START b b
3. START_1 => a a c b b
```

Fig. 3. A traversal example.

A. Equivalence

A key aspect of EMI is the definition of equivalence. How do we define the equivalence of two parsers for two different grammars on an input string? A parser for a grammar usually performs two tasks: (1) Recognize the input string; that is, decide if the input string is part of the language described by the grammar, and (2) Generate a parse tree and finally an AST that models the input string. Therefore, we define equivalence in two stages. In the first stage, the parsers for two grammars G and H are equivalent on I, if both can recognize all strings in I. In the second stage, we compare generated ASTs for strings in I. We say they are equal if, for each string in I, they produce the same AST. A description on how the equality check is performed in the framework is described in detail in the implementation section.

In all parsing libraries, generation of ASTs is achieved by introducing semantic actions for production rules. A semantic action for a production rule creates a node associated with the rule. Since Bohemia is designed to be applicable to all parsing libraries, the grammars it uses to validate parsing libraries do not have semantic actions. For these grammars, the parse tree for an input string is the same as the AST. This design makes it easier to generate library specific mutation files. However, as we will discuss, certain grammar mutations we introduce

```
G:  START -> FACTOR | FACTOR + START
          | FACTOR - START
    FACTOR -> ATOM | ATOM * FACTOR
            | ATOM / FACTOR
    ATOM -> 1 | - START | ( START )

i: (1-1*1)

Traversal of i:
  [ START_0, FACTOR_0, ATOM_2, START_2,
    FACTOR_0, ATOM_0, START_0, FACTOR_1,
    ATOM_0, FACTOR_0, ATOM_0 ]

l: { START_0, START_2, FACTOR_0,
     FACTOR_1, ATOM_0, ATOM_2 }

d: { START_1, FACTOR_2, ATOM_1 }

H:  START -> FACTOR | FACTOR - START
    FACTOR -> ATOM | ATOM * FACTOR
            | ATOM / FACTOR
    ATOM -> 1 | ( START )
```

Fig. 4. An example of pruning dead productions.

change the parse tree, implying Bohemia has to implement its own equivalence checker at the parser-tree level.

We are now ready to introduce the mutations that Bohemia performs to generate EMI variants of context-free grammars. For each kind of mutation, we will describe its intuition and also introduce an example.

B. Prune Dead Productions

The intuition behind this mutation is that, if a rule is not utilized in the production of an input string, the absence of the rule should not disturb the production of the string.

An input string `i` from `I` can be generated by performing a top-down traversal of `G` from its start symbol. The traversal is a list of productions that were used in expanding `G` to produce `i`. Fig. 4 shows an example grammar G and an input string `i` and its associated list of productions. The figure uses the notation of mnemonics we discussed earlier. We call a set of productions used in a traversal to produce `i` *live productions*, denoted by `l`. The difference of the set of all productions in G and `l` is the set of *dead productions*, denoted by `d`. These are the productions that were not used in the generation of `i`. The presence or absence of dead productions makes no difference to the generation of `i`. So, we can randomly eliminate a subset of dead productions to generate an EMI variant `H`. Fig. 4 shows one such `H` generated after removing `START_1` and `ATOM_1`. After removal, `H` should still be able to generate `i` using the same traversal. Generalizing this idea for a set of inputs `I`, we can generate `D`, the set of dead productions for all `i` in `I`. We then use this `D` to randomly prune the dead productions from `G` to generate mutations.

```
G:  START -> ATOM | ATOM + ATOM
    ATOM -> 1 | ( START )

H:  START -> ATOM | ATOM + ATOM
    ATOM -> 1 | ( START )
          | ( START ] | ( START }
```

Fig. 5. An example of adding structurally similar productions.

```
G:  START -> ATOM | ATOM + START
    ATOM -> 1 | ( START )

H:  START -> ATOM | ATOM + START
    ATOM -> 1 | ( START ) | START
```

Fig. 6. An example of introducing mutual recursion.

C. Add Productions

This mutation adds new productions to the existing grammar to generate EMI variants. There are two types: (1) add structurally similar productions, and (2) introduce mutual recursion.

The intuition behind the first type of mutation lies in the implementation of LL(k) and LR(k) parsing algorithms. During their execution, these algorithms look at a certain sequence of tokens and decide which productions match that sequence. For a small look-ahead value of k, the algorithm may not have enough information to make the right decision amongst structurally similar production rules. Hence in the future if a decision turns out to be wrong, the algorithm has to backtrack and explore other alternatives. The mutation of the first type essentially tests if this backtracking is properly implemented. Fig. 5 shows an example of this type of mutation. We pick the "`( START )`" production and create new productions by replacing '`)`' by '`]`' and '}'. When a parsing algorithm like LR(k) encounters a '`(`' it has to choose between three productions, and later backtracks if the choice was incorrect.

The intuition behind the second type of addition is to introduce mutual recursion. Adding this type of mutation tests if a parser library has correctly handled recursive references in a grammar definition and does not get stuck in an infinite loop. Fig. 6 shows an example of this type of mutation. We add a new production "`ATOM -> START`" to introduce mutual recursion. The expected behavior from a parsing library would be to detect a possible infinite loop and abort, or return the parse tree the same as the one produced by the original grammar. One complication is that, if the parser chooses to unwind the recursion a few steps deeper, the parse tree would be different. This has the potential of complicating equivalence checking. However, in the tests that we have performed on a set of parsing libraries, none of the parsing libraries have chosen to take this route. Therefore, this issue is ignored in the rest of the paper.

```
G:  START -> a START b | c

H:  START -> a START1 | c
    START1 -> START START2
    START2 -> b
```

Fig. 7. An example of rolling productions.

D. Roll Productions

In the previous two kinds of mutation the parse trees for the original grammar and its EMI variants are expected to be identical for the same input string. In this mutation, we introduce new nonterminals into the EMI variants and thus the mutation may change the parse tree. The intuition behind this mutation is to test the parse tree generation mechanism in a parsing library.

We introduce new nonterminals by selectively partitioning (rolling) existing production rules. A production rule in a grammar G is a sequence of terminals and nonterminals. Let us represent a production rule R as "S -> $A_1A_2A_3A_4...A_n$", where S is a nonterminal and each A_i is a terminal or a nonterminal. We can generate a new rule R1 by partitioning (rolling) the sequence of A_i's into substrings. Let S1 be a new nonterminal and let $A_iA_{i+1}...A_{i+j}$ be the substring. We now modify R to be "S -> A_1... S1 ...A_n" and add rule "S1 -> $A_iA_{i+1}...A_{i+j}$". In this way we generate the mutated grammar H. Fig. 7 shows an example of this mutation. We first partition "`a START b`" into `a` and "`START b`", and then partition "`START b`" into `START` and `b`.

For an input string, the parse tree generated by this kind of EMI variants differs from that of the original grammar, since new intermediate nonterminals are introduced in the variants. To address this, Bohemia marks the newly generated nonterminals as *inlinable*. The equivalence checker for the parse trees described later in the implementation section uses this information while comparing the generated parse trees.

E. Unroll Productions

We have previously introduced the concept of a traversal of an input string i on grammar G. Each stage in the expansion of a rule in the traversal can be represented by an intermediate parse tree. Given such an intermediate parse tree, we can take an internal node n of the tree and collect all the leaf nodes from the subtree rooted at n. We call that sequence of leaf nodes a partial intermediate string (PIS). What if we add a production rule going from n to that intermediate string to generate a mutation? Will the generated variant be equivalent to the original grammar modulo the input string? It is, as the partial intermediate string eventually leads to the generation of the input string.

By introducing such a production rule, we are basically cutting down the number of expansions required to generate the final string. Fig. 8 makes this idea clear with an example. For each rule in the traversal we generate a partial intermediate string in the order shown under expansion. For each PIS we

```
G:  START -> ATOM | ATOM + START
    ATOM -> 1 | ( START )

i : (1+1)

Traversal (T):
  [START_0, ATOM_1, START_1,
   ATOM_0,  START_0, ATOM_0]

Expansion:
1. START_0 => PIS = ATOM
No mutation as production
START -> ATOM already present.

2. ATOM_1 => PIS = ( START )
No mutation as production
ATOM -> ( START ) already present.

3. START_1 => PIS = ( ATOM + START )
New Rule
H1:  START -> ATOM | ATOM + START
           | ( ATOM + START )
     ATOM -> 1 | ( START )

4. ATOM_0 => PIS = ( 1 + START )
NEW RULE
H2:  START -> ATOM | ATOM + START
     ATOM -> 1 | ( START )
           | ( 1 + START )

5. START_0 => PIS = ( 1 + ATOM )
NEW RULE
H3:  START -> ATOM | ATOM + START
           | ( 1 + ATOM )
     ATOM -> 1 | ( START )
```

Fig. 8. An example of unrolling productions.

check if the production represented by it is already present for the nonterminal associated with the PIS. If the production is absent, then we generate an EMI variant by adding the current PIS as a production rule to G.

This mutation introduces ambiguity in the EMI variants. Some parser libraries are capable of handling ambiguous grammars and return all possible parse trees. Libraries that do not handle ambiguity should return one of the possible parse trees. The parse trees associated with the EMI variants can be compared for equivalence with the original grammar's parse tree by making respective nodes associated with the nonterminal present in the traversal as inlinable.

V. Implementation

In this section we describe how the mutations described in the previous section are put into action. The system is comprised of four components: (1) a mutation generator, (2)

an equivalence modulo checker, (3) input adapters, and (4) output adapters. We now describe each component in detail.

A. Mutation Generator

The mutation generator is responsible for generating EMI variants according to the mutation strategies described in the previous section. In order for the testing to be comprehensive, the mutation generator composes the four kinds of mutations into one large mutation in the following order: Unroller −> Roller −> Adder −> Pruner. Our experimentation showed that this particular order generates a good random set of mutations.

The mutation generator needs a grammar G as input. It internally traverses the grammar to generate a set of input strings. The size of the input-string set is configurable. In the previous section we described the unroller mutation by providing an example with only one input string. To be exhaustive, it is better to run the tests on a large set of strings. However, the unroller mutation is tightly coupled to the input string through its traversal. For the mutation to be effective on a large input set, the inputs must have a common set of mnemonics. The common mnemonic sequence is the common suffix of each of the traversals of the input strings. This sequence is used to generate the mutations. The generated variants are each capable of parsing the entire input set.

Each mutation generated by the unroller is then passed down the compose chain. The output of the mutation generator is a list of pairs of grammars and input-string sets. The input-string sets in all grammars are identical.

B. Equivalence Modulo Checker (EMC)

As described before, grammars used in Bohemia do not have semantic actions. This makes equivalence checking more complex as the parse trees can be different between the original grammar and an EMI variant. The difference in parse trees can arise because of three factors: (1) The roller mutation introduced new nonterminals, and the parsing library correctly added the necessary nodes in the parse tree; (2) The unroller mutation made the mutated grammar ambiguous, and the parsing library correctly returned a parse tree; (3) There is a bug in the parsing library. As described before, the roller mutation tags each newly generated nonterminal as inlinable. EMC starts by comparing the two trees enforcing strict equality from their root nodes. If it encounters a node n tagged as inlinable, it assigns the children of n to the parent of n and then deletes n. We provide an example of this inlining process in Appendix A.

C. Input and Output Adapters

Each parsing library has its own way of taking in context-free grammars. Some libraries have a custom file syntax, while others define DSLs in the languages in which the libraries are implemented. To cater to this wide variety of syntax, Bohemia comes bundled with an input adapter for each parsing library. The job of the input adapter is to convert a grammar and inputs generated by the mutation generator into a format that the parsing library can accept.

As is the case with inputs, the output generated by the parsing library is usually a data structure in the implementation language. The output adapter is responsible for converting this data structure into a JSON format, which is provided as input to the Equivalence Modulo Checker.

VI. Evaluation

Bohemia is implemented in Python 3.7.2. Excluding input and output adapters, Bohemia's implementation has around 400 SLOC. Together with input and output adapters, it has around 600 SLOC.

Bohemia was evaluated on five parsing libraries:

1) Lark [29]: A modern parsing library for Python, which can parse any context-free grammar. It allows the user to choose from a variety of parsing algorithms, including LALR, Earley, and CYK. Lark has been used to generate parsers in a variety of projects, listed on its webpage [29].
2) Nearley [30]: A simple, fast, and powerful parsing toolkit for NodeJS (JavaScript). It is used by a variety of projects: artificial intelligence, computational linguistics, file format parsers, and many more, which are listed on its website [30]. It employs the Earley parsing algorithm.
3) Derpy [31]: An implementation of the derivative parsing algorithm in Python.
4) Happy [32]: An implementation of LALR parsing for Haskell. The GHC implementation of the Haskell language uses Happy to generate its parser.
5) Earley.rb [33]: An implementation of the Earley algorithm in Ruby.

Table I shows the results of evaluation of Bohemia on five parsing libraries. We present the detail of each discovered bug in Section VII and summarize the results next. First, bugs are classified into two types:

1) Output Mismatch (OM) bugs: For a grammar G and a set of inputs I, if mutations of G do not produce equivalent output for inputs in I, we say that there exists an output mismatch bug.
2) Infinite Loop (IL) / Memory Overflow bugs (MO) in a parsing library. When taking a mutated grammar as input, the parsing library may get stuck in an infinite loop or get into a situation of running out of memory. These are also caused by programming bugs in the library.

As shown in Section VII, two discovered bugs are of the output mismatch kind. The presence of this kind of bugs in a parsing library means that it constructs a wrong parse tree for either the original grammar or a mutant. Wrong parse trees may trigger the generation of wrong semantic values that allow attackers to harm a system that uses the parsing library. For the infinite loop or memory overflow bugs, our evaluation discovered seven bugs. Such bugs may allow attackers to crash the parsing library or exhaust resources in systems that use the library—denial of service attacks.

TABLE I
EVALUATION RESULTS

Library	Lark	Nearley	Derpy	Happy	Earley.rb
Bug Count	4	1	2	1	1
Bug Type	OM and IL	MO	OM and IL	IL	IL
Bug Status	1 - Fixed (Existing) 2 - Confirmed 1 - Not Confirmed	1 - Confirmed	1 - Confirmed 1 - Not Confirmed	1 - Not Confirmed	1- Not Confirmed

It is worth pointing out that the infinite loop and memory overflow bugs can also be found by a fuzzing tool that randomly mutates input grammars and does not perform the equivalence testing step. Even for such a tool, we note that the mutations we proposed in Sec IV are necessary components. In addition, the output mismatch bugs can be identified only through equivalence testing. Our manual inspection of the output mismatch bugs Bohemia identified suggests that those bugs would be hard to find automatically through general fuzzing. The problem is that a general fuzzer does not have a notion of equivalence, which is application specific. Bohemia specializes in the parsing library domain and its mutations are designed to respect a notion of equivalence in this custom domain. When we mutate an input grammar to produce an equivalent grammar modulo input, a parser library is supposed to produce two equivalent parsers (modulo inpput). For example, Bug 6 described in Section VII caused the Earley parser in Lark to generate wrong data in the resulting parse tree; without asking users to inspect the parse tree for correctness, a mutation-based parser cannot identify that bug. As future work, we would like to perform further experiments to compare Bohemia with general fuzzing techniques and characterize the strengths and weaknesses of Bohemia.

All discovered bugs listed in Table I have been reported to developers. Of the nine discovered bugs, one is an existing bug, four bugs are new and confirmed by developers, and four are new and unconfirmed by developers yet. Those unconfirmed bugs were reported to developers but they did not respond to our reporting. Based on our manual inspection, we still believe they are true bugs. We categorize a bug as existing, if there was a previous bug report that involved a grammar similar to the one that we discovered. Of the nine bugs, four are present in an Earley parser, one in a CYK parser, two in a derivative parser, and two in an LALR parser. So far, one of the reported bugs in the Earley parser of the Lark library has been fixed thanks to our reporting.

VII. DESCRIPTION OF BUGS FOUND

We describe each bug with the grammar that caused it.

A. Bugs 1, 2, 3 and 4

Fig. 9 shows the grammars and the input set that trigger the bug. The variant grammar causes an infinite loop in the Earley and CYK parsers of Lark and this is an existing and confirmed bug. It also affects the Earley parser in Nearley, causing a memory overflow; this is a new and confirmed bug. It also affects the Earley parser in Earley.rb, causing an infinite loop; this is a new bug but unconfirmed by developers. Mutations at play are roller and adder.

```
G:  START -> a START b | c

I:  [acb, aacbb, aaacbbb, aaaacbbbb]

H:  START -> a START1 | c
    START1 -> START2
    START2 -> START3 | START1
    START2 -> START b
```

Fig. 9. Bugs 1, 2, 3, and 4.

B. Bug 5

```
G:  START -> 1 | 1 + START
           | START { START | ( START )

i:  (1-(((1-1)))-1)-1-1-1
```

Fig. 10. Bug 5.

Fig. 10 shows the grammar and the input that trigger the bug. It is quite peculiar, as the failure happens on the original grammar. The grammar fails to recognize the string and returns an empty parse tree. Some of the mutations also result in failure. The bug affects the derivative parser in Derpy and is a new and confirmed bug.

C. Bug 6

```
G:  START -> ATOM | ATOM + START
    ATOM -> 1 | ( START )

i:  (1+1+(((1+1)+1)+(1))+1)

H:  START -> ATOM | ATOM + START
           | ( ATOM + START )
    ATOM -> 1 | ( START )
```

Fig. 11. Bug 6.

Fig. 11 shows the grammars and the input string that trigger the bug. The bug affects the Earley parser in Lark. The parse

trees generated for the variant grammar do not match with that of the original grammar. The variant's parse tree contains some internal library specific data structures that should not be visible. The bug is a new and confirmed bug. Mutation at play is unroller.

D. Bug 7

```
G:  START -> FACTOR | FACTOR + START
           | FACTOR - START
    FACTOR -> ATOM | ATOM * FACTOR
            | ATOM / FACTOR
    ATOM -> 1 | ~ START | ( START )

i:  ~~1*(1+1*(~1/~~~(1-1)-1/1*1*1/1/1
    /1-1*1-1*1*1/1/1+1*1/1*1/1/1+1)*1
    )/1*1*1*1/1/1/1/1*1/1/1/1/1/1/1+1
    *1*1
```

Fig. 12. Bug 7.

Fig. 12 shows the grammar and the input that trigger the bug. It affects the derivative parser in Derpy. Same as bug 5, this bug is triggered even for the original grammar. The parser gets stuck in an infinite loop for the grammar and the input. Some mutations of the grammar also make the parser get stuck in an infinite loop. The bug is a new and unconfirmed bug.

E. Bug 8

```
G:  START -> 1 | 1 + START
           | START '-' START
           | ( START )

i:  1+1+1

H:  START -> 1 | 1 + START
           | START - START
           | ( START )
           | 1 + 1 + START
```

Fig. 13. Bug 8.

Fig. 13 shows the grammars and the input string that trigger the bug. It affects the LALR parser in Lark. The parser fails to recognize the input string. However, the LALR parser in Happy successfully recognizes and parses the input string. Mutation at play is unroller.

F. Bug 9

Fig. 14 shows the grammars and the input string that trigger the bug. It affects the LALR parser in Happy. The parser gets stuck in an infinite loop. However, the LALR parser in Lark successfully parses the input string. Mutations at play are roller and adder.

```
G:  START -> 1 | 1 + START
           | START - START
           | ( START )

i:  1+(1+1+1+1+1+1+1+1-1-1)

H:  START -> 1 | 1 + START
           | START - START
           | ( START1
    START1 -> START2
    START2 -> START3
    START3 -> START ) | START1
```

Fig. 14. Bug 9.

VIII. Conclusions and Future Work

We have presented four novel EMI mutation strategies for context-free grammars to expose bugs in parsing libraries. These mutation strategies stress test parsing libraries thoroughly. Bohemia, the realization of our mutation strategies, has successfully found bugs across five parsing libraries. As future work, we plan to explore new mutation strategies and more complex equivalence checking resulting from those strategies. Furthermore, how to extend Bohemia to support semantic actions in grammars is an interesting research direction. Finally, to understand the strengths and weaknesses of Bohemia in its effectiveness and efficiency of finding bugs, it would be interesting to systematically compare Bohemia with general fuzzing techniques experimentally, including both black-box and white-box fuzzing.

Acknowledgments

The authors would like to thank anonymous reviewers for their insightful comments and Rodrigo Branco for shepherding the paper. This work was supported by DARPA research grant HR0011-19-C-0073.

References

[1] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Sage: whitebox fuzzing for security testing," *Commun. ACM*, vol. 55, no. 3, pp. 40–44, 2012.
[2] Mozilla, "Home :: Bugzilla :: bugzilla.org," https://www.bugzilla.org/, 2020.
[3] Siguza, "Psychic paper," https://siguza.github.io/psychicpaper/, 2020.
[4] "CVE-2008-3196," https://nvd.nist.gov/vuln/detail/CVE-2008-3196.
[5] "CVE-2016-6354," https://nvd.nist.gov/vuln/detail/CVE-2016-6354, 2016.
[6] A. Barthwal and M. Norrish, "Verified, executable parsing," in *European Symposium on Programming (ESOP)*, 2009, pp. 160–174.
[7] T. Ridge, "Simple, functional, sound and complete parsing for all context-free grammars," in *Certified Programs and Proofs (CPP)*, vol. 7086, 2011, pp. 103–118.
[8] A. Koprowski and H. Binsztok, "TRX: A formally verified parser interpreter," *Logical Methods in Computer Science*, vol. 7, 2010.
[9] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan, "Rocksalt: Better, faster, stronger SFI for the x86," in *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2012, pp. 395–404.
[10] J.-H. Jourdan, F. Pottier, and X. Leroy, "Validating LR(1) parsers," in *European Symposium on Programming (ESOP)*, 2012, pp. 397–416.

[11] J. Bernardy and P. Jansson, "Certified context-free parsing: A formalisation of valiant's algorithm in Agda," *Logical Methods in Computer Science*, vol. 12, no. 2, 2016.
[12] G. Tan and G. Morrisett, "Bidirectional grammars for machine-code decoding and encoding," *Journal of Automated Reasoning*, vol. 60, no. 3, pp. 257–277, Mar 2018.
[13] B. Delaware, S. Suriyakarn, C. Pit-Claudel, Q. Ye, and A. Chlipala, "Narcissus: correct-by-construction derivation of decoders and encoders from binary formats," *Proceedings of the ACM on Programming Languages*, vol. 3, no. ICFP, pp. 82:1–82:29, 2019.
[14] T. Ramananandro, A. Delignat-Lavaud, C. Fournet, N. Swamy, T. Chajed, N. Kobeissi, and J. Protzenko, "Everparse: Verified secure zero-copy parsers for authenticated message formats," in *28th Usenix Security Symposium*, 2019, pp. 1465–1482.
[15] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2014, pp. 216–226.
[16] F. L. Deremer, "Practical translators for LR(K) languages," Cambridge, MA, USA, Tech. Rep., 1969.
[17] B. Ford, "Parsing expression grammars: A recognition-based syntactic foundation," in *31st ACM Symposium on Principles of Programming Languages (POPL)*, 2004, pp. 111–122.
[18] ——, "Packrat parsing:: Simple, powerful, lazy, linear time, functional pearl," in *ACM International Conference on Functional programming (ICFP)*, 2002, pp. 36–47.
[19] J. Earley, "An efficient context-free parsing algorithm," *Communications of the ACM*, vol. 13, no. 2, pp. 94–102, Feb. 1970.
[20] J. Cocke, *Programming Languages and Their Compilers: Preliminary Notes*. USA: New York University, 1969.
[21] D. H. Younger, "Recognition and parsing of context-free languages in time n^3," *Information and Control*, vol. 10, no. 2, pp. 189–208, 1967.
[22] T. Kasami, "An efficient recognition and syntax-analysis algorithm for context-free languages," Air Force Cambridge Research Laboratory, Tech. Rep., 1965.
[23] M. Might, D. Darais, and D. Spiewak, "Parsing with derivatives: a functional pearl," in *ACM International Conference on Functional programming (ICFP)*, 2011, pp. 189–195.
[24] M. D. Adams, C. Hollenbeck, and M. Might, "On the complexity and performance of parsing with derivatives," in *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2016, pp. 224–236.
[25] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," in *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015, pp. 386–399.
[26] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," in *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2016, pp. 849–863.
[27] M. Zalewski, "American fuzzy lop," 2017, [Online; accessed 1-August-2017]. [Online]. Available: http://lcamtuf.coredump.cx/afl/
[28] T. Petsios, A. Tang, S. J. Stolfo, A. D. Keromytis, and S. Jana, "NEZHA: efficient domain-independent differential testing," in *IEEE Symposium on Security and Privacy (S&P)*, 2017, pp. 615–632.
[29] "Lark," https://github.com/lark-parser/lark.
[30] "Nearley," https://nearley.js.org.
[31] "Derpy," https://github.com/agoose77/derpy.
[32] "Happy," https://www.haskell.org/happy/.
[33] "Earley.rb," https://github.com/joshingly/earley.

APPENDIX A
AN EXAMPLE OF INLINING

Fig. 16 describes an example of inlining during equivalence checking. The two parse trees are for the grammars in Fig. 15.

Fig. 16 (a) shows the parse tree for the input string aacbb on G. Fig. 16 (b) shows the parse tree for the same string on

```
G:  S -> a S b | c

H:  S -> a S1 | c
    S1 -> S S2
    S2 -> b
```

Fig. 15. EMC example grammar.

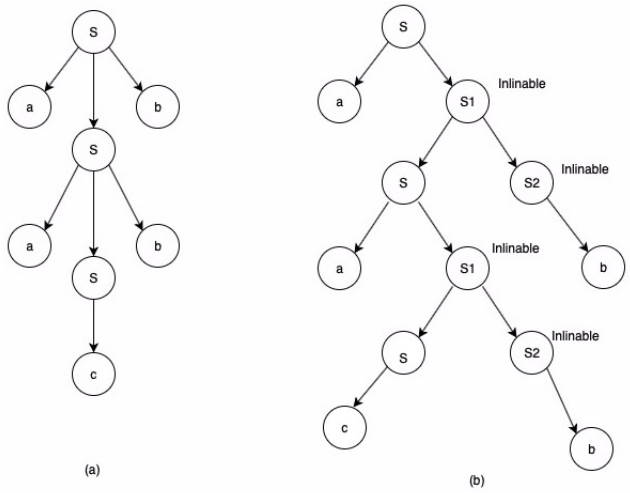

Fig. 16. Equivalence Modulo Checker example.

In case of the unroller mutation, the technique of inlining can still be applied to those parse trees to decide if they are equivalent. The only difference is that the nodes on the original grammar's parse tree get labelled as inlinable according to the unrolling mutation.

H. Inlining the tagged nodes reduces Fig. 16 (b) to Fig. 16 (a). The parse-tree comparison algorithm compares the two corresponding nodes in the two parse trees. If they refer to the same terminal or nonterminal, comparison proceeds to their children. If they are different but one is inlinable, inlining is performed and comparison proceeds as before after inlining. The time complexity of the algorithm is O(n), where n is the number of nodes in the parse tree. If the two parse trees are not equivalent by the aforementioned process, EMC reports a possible bug by generating a bug file with the culprit grammar and the input set. The developer can then use the bug file to debug the parsing library in order to isolate the bug.