# Grammar Mutation for Testing Input Parsers

BACHIR BENDRISSOU, Imperial College London, United Kingdom
CRISTIAN CADAR, Imperial College London, United Kingdom
ALASTAIR F. DONALDSON, Imperial College London, United Kingdom

Grammar-based fuzzing is an effective method for testing programs that consume structured inputs, particularly input parsers. However, if the available grammar does not accurately represent the input format, or if the system under test (SUT) does not conform strictly to the grammar, there may be an impedance mismatch between inputs generated via grammars and inputs accepted by the SUT. Even if the SUT *has* been designed to strictly conform to the grammar, the SUT parser may exhibit vulnerabilities that would only be triggered by slightly invalid inputs. Grammar-based generation, by construction, will not yield such edge case inputs. To overcome these limitations, we present two mutational-based approaches: Gmutator and G+M. Both approaches are built upon Grammarinator, a grammar-based generator. Gmutator applies mutations to the grammar input of Grammarinator, while G+M directly applies byte-level mutations to Grammarinator-generated inputs. To evaluate the effectiveness of these techniques (Grammarinator, Gmutator, G+M) in testing programs that parse various input formats, we conducted an experimental evaluation over four different input formats and twelve SUTs (three per input format). Our findings suggest that both Gmutator and G+M excel in generating edge case inputs, facilitating the detection of disparities between input specifications and parser implementations.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Grammar-based fuzzing, mutant grammars, input parsers

**ACM Reference Format:**
Bachir Bendrissou, Cristian Cadar, and Alastair F. Donaldson. 2024. Grammar Mutation for Testing Input Parsers. *ACM Trans. Softw. Eng. Methodol.* -, -, Article - (December 2024), 21 pages. https://doi.org/https://doi.org/10.1145/3708517

## 1 INTRODUCTION

Random program testing, or *fuzzing*, involves running a software system under test (SUT) on random inputs. For a testing campaign to be maximally effective, the inputs must exercise functionality in both the front-end and the back-end of the SUT. This requires that the inputs conform to the input specification. A common technique is to use a grammar-based generator, and randomised testing using a grammar-based generator is known as *grammar-based fuzzing* [2, 32, 41, 44, 54, 56]. A grammar encodes the input format, and in doing so ensures that the generated inputs are syntactically valid. This is a useful property because it yields inputs that make it past the parsing functionality of the SUT.

Authors' addresses: Bachir Bendrissou, Imperial College London, London, United Kingdom, b.bendrissou@imperial.ac.uk; Cristian Cadar, Imperial College London, London, United Kingdom, c.cadar@imperial.ac.uk; Alastair F. Donaldson, Imperial College London, London, United Kingdom, alastair.donaldson@imperial.ac.uk.


1049-331X/2024/12-ART- $15.00
https://doi.org/https://doi.org/10.1145/3708517

In practice however, a grammar may not correctly capture the intended input format. This is especially true when considering that grammar writing is a manual process and is prone to human error.

Another limitation with grammar-based generators is that they are unable to generate inputs not allowed by the grammar rules. While this limitation is by design, it may cause the test generator to miss exercising some program behaviours both during parsing and while testing the core functionality. For instance, the SUT implementation may deviate from the official grammar of an input format [39, 47], or the SUT may not implement all of the necessary input validation [27, 34]. It is therefore useful to generate test cases that slightly deviate from the input specification.

In this paper, we investigate two approaches that aim to mitigate the above limitations.

**Gmutator** The first approach, which we call Gmutator, is based on *grammar mutation*. This technique involves first mutating the grammar associated with an input format. The mutated grammar is then fed to a grammar-based generator, which yields inputs that conform to the mutated grammar. Crucially, mutation operations are applied to the grammar, at the level of grammar rules, rather than being applied directly to generated inputs, at the level of characters. This means that generated inputs exhibit a structure expressed by that of the mutated grammar. Our motivation for proposing and studying this technique is as follows. Grammar-level mutation may yield inputs that are invalid in interesting ways: because they arise from a mutated grammar they might not conform to the syntax of the original grammar, yet because they are nevertheless generated from a formal grammar, and in particular a formal grammar that is very similar to the original grammar, their syntax will *almost* conform to that of the original grammar. This may reveal interesting discrepancies where such almost-valid inputs are accepted by an SUT.

**G+M** The second approach, which we call G+M (short for "grammar plus mutations") involves using a grammar-based generator to produce an input (without applying mutations to the grammar), and then applying string-level mutations (similar to byte-level mutations employed by mutation-based fuzzers such as AFL [57]) to the generated input. A string-level mutation may involve the deletion or duplication of a random sequence, or the insertion of a keyword, where keywords are automatically collected from the grammar. As a result of these mutations, the syntax of the resulting inputs may deviate from the grammar rules. Our motivation for studying this technique is that it is a simpler and more straightforward idea compared with grammar mutation, and it is thus interesting to compare how effective the two techniques are. By using well-formed inputs (coming from the original grammar) as a starting point, G+M has the potential to generate almost-valid inputs, e.g. by removing a keyword or duplicating a region of text that happens to correspond to a complete structural element of the input format. As with Gmutator, this has the potential to reveal interesting discrepancies in relation to the inputs that an SUT actually accepts, compared with the inputs that the SUT *should* accept according to the official grammar for an input format. However, G+M is also likely to yield drastically-invalid inputs much of the time, e.g. by deleting portions of an input in a manner that completely destroys the the structure of the input.

We implement the Gmutator and G+M techniques on top of Grammarinator, an off-the-shelf grammar-based generator [31].

As discussed above, we observe that while the mutations implemented in both techniques may invalidate the input specifications, they have the potential to produce edge-case inputs: *slightly* invalid inputs that would not be produced by standard grammar-based generation. In the context of testing input parsers, producing edge-case inputs can offer two benefits. First, this class of inputs can help identify any mismatch between the language expressed by the original grammar and the

language accepted by the SUT. Second, these inputs can exercise code regions that otherwise are inaccessible by well-formed inputs.

Following the evaluation plan declared in our registered report [22], and in order to fully assess the two techniques, we have conducted a large experiment where we compare and analyse test case corpora produced by Grammarinator, Gmutator, and G+M. As target benchmarks, we selected twelve SUTs that process four distinct input formats (three SUTs per input format). We ran every (generator, SUT) pair for 24 hours.

We report on parsing discrepancies identified via these experiments: cases where an SUT and corresponding reference grammar do not agree on the validity of a given input. We also report on the code coverage achieved by the input corpora generated by the three techniques.

Both Gmutator and G+M were able to find parsing discrepancy bugs in several SUTs that cannot be found by Grammarinator because they are triggered by invalid inputs. Moreover, both approaches detected a memory bug in cJSON that was missed by Grammarinator. Our findings suggest that there is little difference between Gmutator and G+M in terms of their ability to find parsing discrepancies and crashes. With respect to code coverage, our results show that very similar levels of code coverage are achieved by Gmutator, G+M and Grammarinator.

In summary, we make the following contributions:

(1) We propose Gmutator, a novel grammar mutation technique, that helps generate different approximations of a given input grammar;
(2) We introduce a second technique G+M, where we apply string-level mutations to input strings produced by a grammar-based generator;
(3) We present a large evaluation comparing our two mutation-based approaches—Gmutator and G+M—against a standard grammar-based generator, Grammarinator. Our results show that both Gmutator and G+M yield comparable outcomes, but that the techniques are complementary with respect to the coverage they achieve on the codebases of various SUTs;
(4) We show the efficacy of Gmutator and G+M at identifying parsing discrepancies. We also discuss a sample of such parsing discrepancies, including developers' responses.

**Historical note:** In our registered report [22] we proposed using G+M (referred to as Grammarinator+Mutations in the registered report) merely as a baseline against which to compare Gmutator. Preliminary experiments quickly indicated that the G+M technique was more effective than we had anticipated, and might therefore be of wider interest to the software testing community than we predicted when we prepared our registered report. For this reason we decided to give Gmutator and G+M equal prominence in this paper.

## 2 ILLUSTRATIVE EXAMPLE

Before going into details of our mutation-based techniques, namely Gmutator and G+M, we first illustrate the ideas of grammar mutation and string-level mutation on which these techniques are based.

JavaScript Common Object Notation (json) [25] is a standard format for representing structured data, and is commonly used as an interchange format between software tools. A grammar for part of json is shown in Figure 1a, where the notation uses the Backus-Naur Form (BNF), and is expressed using the popular ANTLR format, which we discuss further in §3.1. To keep our grammar examples compact and simple to read, we omit rules and constructs that are not relevant for this example. The highlighted parts of the grammar relate to mutations that we will describe in due course.

As shown in the grammar, a json document consists of a `value`, which can be of multiple types. For example, an array (`arr`) is defined as a sequence of one or more values, separated by commas,

```
1  json
2      : value EOF ;
3  obj
4      : '{' pair (',' pair)* '}'
5      | '{' '}' ;
6  pair
7      : STRING ':' value ;
8  arr
9      : '[' value (',' value)* ']'
10     | '[' ']' ;
11 value
12     : STRING | NUMBER | obj | arr
13     | 'true' | 'false' | 'null' ;
14 STRING
15     : '"' (ESC | CHAR)* '"' ;
16 ESC
17     : '\\' (["\\/bfnrt] | UNICODE)
         ;
18 UNICODE
19     : 'u' HEX HEX HEX HEX
         ;
```

(a)

```
1  json
2      : value (obj | EOF) ;
3  obj
4      : '{' pair (',' pair)* '}'
5      | '{' '}' ;
6  pair
7      : STRING ':' value ;
8  arr
9      : '[' value (',' value*)* ']'
10     | '[' ']' ;
11 value
12     : STRING | NUMBER | obj | arr
13     | 'true' | 'false' | 'null' ;
14 STRING
15     : '"' (ESC | CHAR)* '"' ;
16 ESC
17     : '\\' (["\\/bfnrt] | UNICODE)
         ;
18 UNICODE
19     : 'u' (STRING |
        HEX) HEX HEX HEX ;
```

(b)

Fig. 1. Simplified version of the JSON grammar (left) and one of its mutant grammars (right). The highlighted parts show the mutations applied.

and surrounded by square brackets. As a second example, `UNICODE` encodes a UNICODE character, which is specified by the letter 'u' followed by four `HEX` digits.

### 2.1 Illustration of Grammar Mutation (used by GMUTATOR)

Given this grammar, GMUTATOR applies mutations to its rules to construct mutant grammars. Figure 1b shows one of the possible mutant grammars, which was obtained via three mutations:

- **Mutation 1 (line 2):** With this mutation applied, a JSON file can now consist of multiple roots rather than a single one. For instance, the input `{} {}` is accepted by the mutant grammar, but not by the original one.
- **Mutation 2 (line 9):** With this mutation applied, an array may have values that are not correctly comma-separated. For instance, the input `[true, ]` is accepted by the mutant grammar, but not by the original one.
- **Mutation 3 (line 19):** With this mutation applied, a UNICODE value may start with an arbitrary string. E.g., the input `"\ur282"` is accepted by the mutant grammar, but not by the original one.

Differences between the two grammars will inevitably reflect in differences in the languages they represent. An input generated from the mutated grammar can have a syntax different from the correct JSON format. For instance, consider mutation 2. This mutation can lead to the derivation of the input `[1, 2 "foo"]`.

### 2.2 Illustration of String-level Mutation (used by G+M)

In contrast to GMUTATOR, the G+M approach that we also investigate uses a grammar-based generator to produce an input using the original (non-mutated) grammar for an input format, and then applies string-level mutations to the resulting generated input.

String-level mutations are similar to byte-level mutations employed by mutation-based fuzzers such as AFL [57]. They can be applied to arbitrary strings in a manner that is agnostic to the input format of the SUT—i.e. they do not rely on any knowledge of a grammar.

We use two examples to illustrate string-level mutations (which we discuss in detail in §3.2):

- Given a well-formed JSON string such as `{"name": foo}`, a string-level mutation might delete the substring "na", yielding the string `{"me": foo}`. In this case, the result happens to be valid JSON.
- Alternatively, a mutation might duplicate the characters "`: foo}`", leading to an invalid input as follows (the duplicated characters are shown in blue):

`{"name": foo}` ⟶ `{"name": foo}: foo}`

### 2.3 Discussion

A key difference to note is that grammar mutation (employed by GMUTATOR) involves making changes to the grammar, so that a grammar-based generator has the potential to generate inputs from the mutated grammar that cannot be generated by the original grammar, while string-level mutation (employed by G+M) involves generating a valid input using an unmodified grammar and then treating this input as a string and mutating it in a grammar-agnostic way.

In the context of JSON, a single mutation to the grammar rule that describes integers may cause a generated input to feature malformed integer tokens exactly where integer tokens would normally be generated, and has the potential to affect many places where well-formed integers would usually be generated. In contrast, a string-level mutation applied to a well-formed JSON input might, by chance, modify the characters that happen to be associated with one particular integer value. The probability that string-level mutations would make localised changes to multiple distinct integers (without affecting other parts of the input) is very low.

## 3 APPROACH

This section presents in detail our two mutation-based techniques for producing edge case inputs: grammar mutation (§3.1) and string-level mutation (§3.2).

### 3.1 Grammar Mutation

We start by discussing how our grammar mutation technique GMUTATOR is designed, and how it can be applied to any ANTLR grammar.

The ANTLR grammar format [43] is a popular format in the software developer community, that is used to write input grammars. The open-source ANTLR repository [1] provides more than 200 grammars, specifying a variety of languages and formats. Moreover, the ANTLR tool is a parser generator that converts ANTLR grammars to parsers, which we find very useful in our experiments.

An ANTLR grammar consists of a set of rules that describe the syntax of a language. Every non-terminal symbol in the grammar is defined in terms of other non-terminals or terminals, whereas a terminal symbol defines a token as a regular expression.

Given an ANTLR grammar, GMUTATOR works as follows. First, a grammar rule is selected. Then GMUTATOR chooses one mutation operator from a set of predefined operators. The mutation operator is applied to the rule, resulting in a new mutated grammar. This process can be repeated

multiple times, so that multiple changes to the grammar are made, depending on how drastic we want the changes to be. We set the default to be three mutations, which we have found to produce noticeable changes in inputs generated by the final grammar.

We designed four types of mutations, described below:

(1) **Repetition:** Change the number of allowed repetitions of an expression to zero-or-more. This can be done by changing an existing repetition operator to `*`, or introducing `*` when there is no existing repetition operator. Examples of this mutation are:
   - Repeat a terminal, e.g. `']'` ⟶ `']'*`
   - Change the number of repetitions of a non-terminal from one-or-more to zero-or-more, e.g. `foo+` ⟶ `foo*`
   - Change an optional subrule to zero-or-more repetitions of the subrule, e.g. `(...)?` ⟶ `(...)*`

(2) **Concatenation:** Allow the concatenation of two rules that would normally be alternatives, e.g. change `foo | bar` to `foo | bar | foo bar`, so that in addition to the choice of `foo` or `bar`, the sequence `foo bar` is allowed.

(3) **Relax excluded character set:** Replace a top-level regular expression defining the complement of a set of characters—i.e. a regular expression of the form `~`$R$ where $R$ defines a set of characters—with the full character range; e.g. `~[0-9]` ⟶ `.`, which changes the regular expression that accepts any non-digit character to a regular expression that accepts any character.[1]

(4) **Introduce choice:** Replace a use of a lexer/parser rule with a choice between this or another lexer/parser rule appearing in the grammar; e.g. `HEX` ⟶ `(STRING | HEX)`.

The mutations outlined were crafted to adhere to three specific properties. Mutations applied to an ANTLR grammar do not invalidate the ANTLR syntax of the grammar. This is necessary in order for Grammarinator to successfully parse the grammar and produce a generator. Second, mutations are self-contained in relation to the source grammar. This implies that mutations are derived from the rules of the source grammar. In other words, we don't import constructs from external grammars or sources. A third property is monotonicity. We say that a given mutation is monotonic, if after applying the mutation to grammar G to derive grammar G', the language expressed by G' subsumes the language expressed by G. More formally:

$$\mathcal{L}(G) \subseteq \mathcal{L}(G'),$$

where $\mathcal{L}(G)$ denotes the set of inputs derivable from a grammar $G$.

The rationale for our design decision to use mutations that monotonically increase the input space that the grammar describes is to allow localised mutations of any inputs that could be generated by the original grammar. Considering the **concatenation** mutation type again, suppose we have a grammar that defines add expressions, using the rule `AddExpr -> AddExpr + AddExpr | Const` where `Const` defines a numerical constant. If the rule would be mutated to `AddExpr -> AddExpr + AddExpr Const` instead of `AddExpr -> AddExpr + AddExpr | Const | AddExpr + AddExpr Const` then one could not generate large, mostly-valid expressions where only *some* subtrees would be mutated. For instance, the expression `1 + 3 + 4 + 5 4`, which uses a mixture of the original and mutated rule, could not be generated.

Another advantage of our design decision that could prove useful in the future is that it leaves the door open for grammar-based fuzzing that operates on initial seed inputs. For seed inputs to be usable when fuzzing with a mutant grammar, it must be the case that they can be expressed using

[1]This mutation could be generalised so that it would consider replacing *any* top-level regular expression (not merely a complement expression) with a more permissive one. We chose to implement the more restricted form of the mutation (that only applies to complements) to maximise the chances of generating inputs that involve characters that are explicitly disallowed by the original grammar.

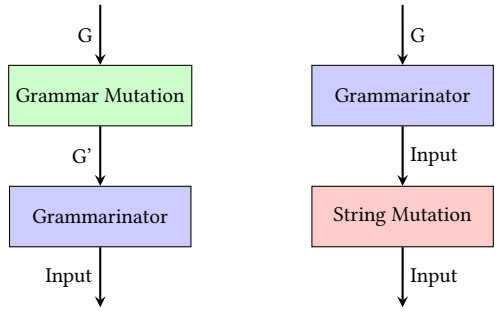


Fig. 2. Grammar mutation vs. string-level mutation.

the mutant grammar so that they can be parsed. While GMUTATOR does not yet make use of seed inputs, our approach ensures that this would be possible to support in the future.

### 3.2 String-level Mutation

We now describe the string-level mutation technique used by G+M. Recall from §2.2 that G+M works by using a grammar-based generator to produce a string that corresponds to a well-formed input according to the grammar. It then applies mutations to the string at random to produce a new string that may not be (and in practice is unlikely to be) well formed according to the grammar.

The mutation strategy here is a dumb approach, as it does not use any knowledge of the input specification. Byte-level mutators have proven to be very effective in automated software testing, and are adopted by many modern fuzzers, such as libFuzzer [37] and AFL [57]. We implement a simplified version of this approach in a tool called G+M. Given an input grammar, we derive its input generator using GRAMMARINATOR. Using the generator, we produce a well-formed input. Next, we apply string-level mutations to the input, leading to the creation of a new input. The mutations we implement are inspired by fuzzers such as AFL. In particular, we implement three types of mutations:

(1) Duplication: We select a sequence of contiguous bytes of a random size at a random position in the input string. We insert a duplicate of the sequence at the end of the original sequence.
(2) Deletion: We delete a sequence of contiguous bytes of a random size at a random position in the input string.
(3) Token insertion: We insert a random grammar token at a random position in the input string. Tokens are keywords extracted from the input grammar, meaning that they are specific to the input language described by this grammar.

Figure 2 highlights the difference between the workflow of grammar mutation and the workflow of string-level mutation. In the grammar mutation approach, the mutation takes place *before* the GRAMMARINATOR phase, while in the string mutation approach, the mutation follows *after* GRAMMARINATOR. This arrangement is essential due to the dependencies involved: The mutated grammar is used as input by GRAMMARINATOR, whereas string mutation requires a string input from GRAMMARINATOR.

## 4 EVALUATION

To study the implications of incorporating mutations to grammar-based generation, we have designed and conducted several experiments. The goal of these experiments is to compare how

Grammarinator and G+M perform against standard grammar-based fuzzing in the context of testing input parsers.

We start by outlining the research questions our evaluation aims to answer (§4.1). We then discuss the target input formats we study (§4.2), and the SUTs consuming these input formats that we test (§4.3). Next, we explain the procedure we use for generating inputs and using generated inputs for testing (§4.4), and other experimental settings (§4.5). Finally, we present and discuss our results (§4.6).

### 4.1 Research Questions

As a baseline for our experiments, we use Grammarinator, because it is open source and well maintained, and has been used in several recent papers related to grammar-based fuzzing [45, 48, 52]. Grammarinator operates directly on the popular ANTLR format [43], which our tools also support.

We design our evaluation experiments to answer the following research questions:

*RQ1*: To what extent can Gmutator and G+M identify discrepancies between the inputs that an SUT accepts, and inputs that conform to the grammar associated with the input format the SUT claims to consume?

*RQ2*: What are the reasons for such discrepancies, and in particular do they relate to unintended acceptance of invalid inputs by the SUT, intentional acceptance due to the SUT being permissive by design, or a lack of precision in the available ANTLR grammar for the input format?

*RQ3*: How does grammar-based fuzzing using Grammarinator, Gmutator and G+M compare in terms of the SUT code coverage that is achieved, and in terms of the SUT crashes that are identified?

### 4.2 Target Input Formats

For our experiments, we select four input formats, that have varying complexity: json, lua, url, and xml. The ANTLR grammars defining these formats are extracted from the ANTLR GitHub repository [1]. The grammars encode the syntax rules of a language. Consequently, these grammars are well suited for parsing the syntax of input strings. However, using these grammars to generate well-formed inputs can be ineffective when the language has semantic properties. Indeed, when generating inputs from these grammars using Grammarinator, we observe a high rejection rate by the target SUTs. For these reasons, we decided to make the following modification to our grammars, to make them suitable for generating semantically-valid inputs:

**json:** No modifications were made to the json grammar.

**lua:** We added constraints to the grammar to ensure that: (i) the `break` token can only appear inside loops and (ii) the `<close>` attribute should not appear more than once in an attribute name list. We also simplified the grammar by removing the `goto` construct, as adding constraints to model it fully would have complicated the grammar.

**url:** IPv6 addresses consist of exactly 8 hexadecimal numbers, separated by colons. Accordingly, we changed the grammar to specify 8 hexadecimals instead of an arbitrary number of hexadecimals. Additionally, we noted that different url SUTs implement different url schemes. Therefore, we limited the allowed url schemes to the three most common schemes: `ftp`, `http`, and `https`.

**xml:** We added constraints to the grammar to ensure that: (i) a closing tag name must match the corresponding opening tag name and (ii) if the *declaration* tag is present (of the form `<?xml ...>`), then it must include the `version` attribute.

Table 1. The systems under test.

| SUT | Input format | Language | Version | LOC | Notes |
|---|---|---|---|---|---|
| cJSON [28] | JSON | C | 1.7.8 | 2,348 | Ultralightweight JSON parser |
| Parson [27] | JSON | C | 1.4 | 2,179 | Lightweight JSON library |
| simdjson [36] | JSON | C++ | 3.2.0 | 10,356 | Fast parser for large JSON files |
| Luac [33] | LUA | C | 5.4.4 | 17,327 | Parser component of the official Lua implementation |
| LuaJIT [42] | LUA | C | 2.1.0 | 49,725 | Just-In-Time (JIT) compiler for the Lua programming language |
| py-lua-parser [26] | LUA | Python | 3.1.1 | 3,823 | Lua parser and AST builder written in Python |
| Aria2 [51] | URL | C++ | 1.36.0 | 93,223 | Utility for downloading files |
| Curl [49] | URL | C | 8.0.0 | 146,879 | Command-line tool for transferring data with URLs |
| Wget [40] | URL | C | 1.21.3 | 79,974 | Program that retrieves content from web servers |
| fast-xml-parser [30] | XML | JavaScript | 4.2.2 | 1,857 | Tool that validates XML and parses XML to JS Object |
| Libxml2 [50] | XML | C | 20902 | 215,759 | XML parser and toolkit originally developed for the GNOME Project |
| Pugixml [34] | XML | C++ | 1.13 | 22,853 | XML processing library |

The modifications described above improve the quality of our input generators, while being simple enough to implement. Note that these changes do not have any effect on the validity of our evaluation. They are needed even for standard grammar-based fuzzing to be useful for these input formats.

### 4.3 Systems under Test

For each input format, we have identified three relevant SUTs, summarised in Table 1. Even though our approach is a blackbox method, we show the total number of lines of code (LOC) for each SUT (gathered using the cloc tool [24]) as an indication of their varying complexity. We chose recent versions of SUTs and, for reproducibility, indicate which versions we use in our evaluation.

Our choice of SUTs was guided by: restricting to open-source software (for ease of communication with developers, and so that we can gain insight into fixes to bugs that we report); including some programs written in C/C++ (the unsafe nature of C/C++ means that SUTs written in C/C++ have the potential to benefit greatly from fuzzing); and for each input format choosing at least one SUT that is widely-used (in particular, cJSON has 8.8k stars on GitHub, Luac is part of the official implementation of the widely-used lua language, Curl is a standard tool for URL-based data transfer, and Libxml2 has been actively developed and maintained for more than two decades).

### 4.4 Procedure for Generation

The two mutational-based tools discussed in this paper, Gmutator and G+M, aim to complement existing grammar-based generators. The primary objective of these tools is to explore the input space of a given program that is at the "edge" of what is defined by the input grammar. In particular, we seek to evaluate how effective the tools are at discovering inputs that the SUT accepts but the

original grammar does not, and whether they can reach program code that is unreachable with inputs generated from the original grammar.

**Grammarinator generation.** Grammarinator takes an ANTLR grammar and transforms it into generator code written in Python, then it produces inputs using the generator. The tool supports a *maximum depth* option, which sets the maximum length of any generation path from the root node to a leaf in the tree. To avoid generator timeouts, as well as parsing timeouts arising from overly large inputs, we set the maximum depth to 60 for all input formats except lua, for which we use a maximum depth of 20 (due to the more complex nature of this grammar).

**Gmutator generation.** Gmutator repeats the process of creating a mutant grammar and then generating inputs using that grammar. Each mutant grammar is obtained by applying three mutations to the original grammar, at random. A mutant grammar is used to generate 40 inputs before Gmutator moves to the next mutant grammar. We have found that this number of inputs typically allows all the rules of the ANTLR grammars we have experimented with to be exercised at least once. Again, we set maximum depth to 20 for lua and 60 for other input formats.

**G+M generation.** For this setup, which involves generating inputs using a standard grammar and subsequently mutating them, the same process is used as for Grammarinator above, except that after each input is generated, between one and three random string-level mutations will be applied to the input (where the number of mutations is also chosen at random).

### 4.5 Experimental Settings

We run experiments on a Linux Ubuntu 20.04 x86_64 machine with 40 CPU cores, 16 GiB of RAM, and 252 GiB of storage. The SUTs are run in a Docker container without a network connection, so that our url-processing SUTs (Curl, Wget and Aria2) will be expected to terminate gracefully with a "no network connection" error if they do manage to parse a given input successfully.

For each (generation tool, SUT) pair (where the generation tools are Grammarinator, Gmutator and G+M), we perform three 24 h runs. Each run repeats the process of:

(1) Generating an input using the generation tool.
(2) Running the input against a coverage-instrumented version of the SUT, logging the output and exit code for subsequent analysis. To account for inputs that trigger infinite loop bugs in our SUTs, or that lead to excessive SUT runtime, we use a timeout of 3 s per input.
(3) Attempting to parse the input using an ANTLR-generated parser for the original grammar (to record whether or not the input is valid).

Performing three repeat runs allows us to present averaged coverage data, whilst keeping the CPU time required for our experiments tractable. Our experiments require (4 input formats) × (3 SUTs per input format) × (3 generation tools) × (3 repeat runs) × (24 h per repeat run) = 2,592 hours of CPU time.

An important part of our evaluation involves looking for discrepancies between different SUTs that accept the same input format. It is therefore important that we generate identical sequences of inputs for the SUTs we wish to compare. Each generation tool can be made deterministic by being provided with a pseudo-random number generator seed. To ensure that SUTs are tested with identical inputs we use the following strategy. In the first 24 h run for a (generation tool, SUT) pair, the pseudo-random number generator of the generation tool is initialised using the sequence of seeds $[0, 3, 6, 9, \dots]$. On the second and third 24 h repeat runs, the seed sequences $[1, 4, 7, 10, \dots]$ and $[2, 5, 8, 11, \dots]$ are used, respectively. This means that, for example, the first repeat run in which Grammarinator is used to test cJSON (one of our json-consuming SUTs; see Table 1) will involve exactly the same generated inputs as for the first repeat run in which Grammarinator is

used to test Parson (another of our json-consuming SUTs), allowing the results of these runs to be compared across the SUTs.

A downside of this method is that CPU time will be devoted to redundantly generating identical inputs to feed to different SUTs, and checking whether these inputs are valid. However, these overheads are part of the true cost associated with testing via our method, so it is fair that they absorb part of the time budget associated with each run. The approach also avoids the need to guess in advance an upper bound on how many inputs it will be possible to generate and process within a 24 h time period (which may vary across input formats and SUTs), and also avoids the problem of testing proceeding at the speed of the slowest SUT (which would be a problem if we instead generated an input and then executed the input against all relevant SUTs before moving on to the next input).

### 4.6 Results

In this section, we present and discuss our evaluation results. We first report on our findings on differential testing of SUTs and their reference grammars (§4.6.1), including the performance of different tools in identifying parsing discrepancies. Next, we illustrate the extent of disagreement among different parser implementations for a single input format and provide some examples (§4.6.2). Following this, we manually investigate specific instances of parsing discrepancies and explain the underlying reasons (§4.6.3). Finally, we report code coverage achieved by each tool across all SUTs, and discuss the reasons for the coverage differences and a crash detected by one of the tools (§4.6.4).

#### *4.6.1 Differential testing between the SUT and the parser generated from the original grammar.*

For Grammarinator-, Gmutator- and G+M-generated inputs, we record how many are valid vs. invalid according to the original grammar. This is achieved by attempting to parse each input using the ANTLR-generated parser derived from the original (non-mutated) grammar.

For each SUT, we then identify inputs for which the SUT and the ANTLR-generated parser disagree on validity. These parsing discrepancy issues can be classified into two categories, as follows:

**Accept-invalid.** An *invalid* input is an input that is rejected by the original grammar, that is, it is not derivable by this grammar. An *accept-invalid* input is an invalid input that is accepted by an SUT for the associated input format.

**Reject-valid.** A *valid* input is an input accepted by the original grammar. Another interesting measurement is the number of valid inputs that are rejected by an SUT—we call these *reject-valid* inputs. With respect to evaluating Gmutator and G+M, this category is less important than the *accept-invalid* category above. This is because every generator tool has the potential to generate valid inputs, and thus the potential to discover *reject-valid* inputs. In contrast, only Gmutator and G+M can generate invalid inputs, so only these tools have the potential to discover *accept-invalid* inputs.

**Discussion of results.** To address *RQ1*, we examine Table 2, which shows the number of discrepancies identified by each tool on each SUT run. We observe that Grammarinator produces on average more reject-valid than other tools. This is mainly due to the high rate of valid inputs produced by Grammarinator. On the other hand, we see no accept-invalid instances generated by Grammarinator. This is expected, since Grammarinator generates only valid inputs by construction. On an SUT level, we find that reject-valid cases are more frequent with lua and url subjects. As discussed above, we consider reject-valid cases less important than accept-invalid ones.

Table 2. Average number of parsing issues recorded by all tools over three 24-hour runs.

| SUT | Grammarinator | | | Gmutator | | | G+M | | |
|---|---|---|---|---|---|---|---|---|---|
| | Inputs | Accept-invalid | Reject-valid | Inputs | Accept-invalid | Reject-valid | Inputs | Accept-invalid | Reject-valid |
| cJSON | 138,069 | 0 | 753 | 138,178 | 1,645 | 366 | 127,379 | 1,927 | 95 |
| Parson | 138,082 | 0 | 7,890 | 138,168 | 7,574 | 4,157 | 127,370 | 31,962 | 686 |
| Simdjson | 137,372 | 0 | 2,292 | 137,491 | 0 | 1,355 | 126,776 | 0 | 296 |
| Luac | 37,836 | 0 | 2,483 | 61,489 | 309 | 205 | 45,526 | 38 | 155 |
| LuaJIT | 37,739 | 0 | 35,105 | 61,105 | 10 | 11,081 | 44,968 | 13 | 2,755 |
| Py-lua | 32,256 | 0 | 27,979 | 47,941 | 67 | 6,150 | 37,321 | 894 | 1,934 |
| Aria2 | 108,222 | 0 | 33,162 | 104,898 | 17,716 | 18,021 | 104,385 | 24,192 | 5,256 |
| Curl | 125,305 | 0 | 17,421 | 118,985 | 22,715 | 9,382 | 119,157 | 37,245 | 3,549 |
| Wget | 132,404 | 0 | 13,626 | 124,287 | 24,457 | 6,507 | 123,099 | 57,894 | 2,125 |
| Fast-xml | 4,028 | 0 | 114 | 4,017 | 116 | 69 | 3,989 | 240 | 119 |
| Libxml2 | 119,595 | 0 | 12,653 | 120,221 | 138 | 9,060 | 114,045 | 25 | 9,203 |
| Pugixml | 125,316 | 0 | 0 | 123,698 | 9,356 | 728 | 118,734 | 8,370 | 4,547 |

First, because unlike accept-invalid, inputs resulting in reject-valid cases can be produced by all generators. Second, as we shall see later, many reject-valid cases are caused by semantic violations, while our context-free grammars do not include all semantic rules.

From the table, we can see that Gmutator and G+M tools were able to identify accept-invalid instances in most SUTs. The mutations performed by these tools allowed the creation of inputs that are not recognisable by the original grammar, but nevertheless are accepted by a corresponding SUT. These edge cases highlight a mismatch between an SUT and its reference grammar. The url SUTs exhibit the most instances of this type, indicating that they are highly permissive compared with the reference ANTLR grammar.

We observe a notable trend: G+M usually produces more accept-invalid instances compared to Gmutator. This disparity can be attributed to the substantial mutation impact (throughput) of G+M. Each input generated by G+M is guaranteed to undergo a mutation, resulting in a higher likelihood of being invalid. Conversely, inputs generated by a mutated grammar may turn out to be valid if the mutated part of the grammar was not exercised during generation. Additionally, the prevalence of short input strings generated by Grammarinator increases the probability of string-level mutations disrupting input structure. This is further compounded by the fact that short strings often indicate limited grammar coverage, reducing the likelihood of exercising the mutated rule. Upon thorough manual inspection, we confirmed that neither of the Gmutator and G+M tools identified any issues that the other did not also identify.

As discussed above, we attribute the performance difference in Table 2 between G+M and Gmutator in terms of parsing discrepancies detection to the low disruptive level of grammar mutations. We confirmed this by running a small scale one hour experiment, in which we increase the number of grammar mutations from three to nine. While this experiment is short, it was designed to assess the tools' behaviour during their most active phase of defect discovery, which typically occurs early in the execution process. Under these new settings, Gmutator finds more *accept-invalid* instances than G+M in five out of the twelve SUTs. Furthermore, with respect to one cJSON crash that we discuss in §4.6.4 below: Gmutator finds the cJSON crash within the first hour, while G+M does not. Therefore, the differences in the results appear to relate at least partly to differences in the way the tools are configured.

Table 3. Unique `accept-invalid` and ANTLR issues discovered by GMUTATOR and G+M.

| # | SUT | Description | Status | # |
|---|---|---|---|---|
| 1 | PY-LUA-PARSER | Fail to reject unassigned global variable | Fixed [12] | 9 |
| 2 | PY-LUA-PARSER | Fail to reject an invalid escape sequence | Fixed [9] | |
| 3 | PY-LUA-PARSER | Parsing standalone name tokens | Fixed [15] | |
| 4 | PY-LUA-PARSER | Missing function call arguments | Fixed [14] | |
| 5 | PY-LUA-PARSER | Fail to parse chained comparisons | Fixed [13] | |
| 6 | WGET | Semicolon incorrectly handled in userinfo | Fixed [21] | |
| 7 | ANTLR | Parsing LUA long comment as short comment | Fixed [18] | |
| 8 | ANTLR | Ambiguity in URL grammar | Fixed [7] | |
| 9 | ANTLR | Underscore not allowed in XML NameStartChar | Fixed [19] | |
| 10 | CJSON | Fail to reject invalid escape character | Confirmed [10] | 4 |
| 11 | FAST-XML-PARSER | Fail to reject multiple root nodes | Confirmed [11] | |
| 12 | FAST-XML-PARSER | Validation of invalid XML declarations | Confirmed [20] | |
| 13 | FAST-XML-PARSER | Parsing an invalid XML element | Confirmed [17] | |
| 14 | PARSON | Accepting invalid array | Rejected [5] | 2 |
| 15 | PARSON | EOF not enforced | Rejected [8] | |
| 16 | CJSON | Accepting invalid integers | Reported [6] | 2 |
| 17 | PY-LUA-PARSER | Literal string gets parsed as LUA code | Reported [16] | |

#### 4.6.2 *Differential testing across SUTs.*

The metrics discussed earlier can be useful for measuring the gap between the language accepted by an SUT and the language expressed by its grammar, partially answering *RQ2*. These measurements can serve as an indication of the effectiveness of grammar-based fuzzing on a given SUT. Alternatively, they can help reveal how mature a given parser implementation (or grammar) is, with regard to compliance with an input specification. In our case where multiple SUTs of one input format are available, we can compare the degree of mismatch among the different SUTs. For instance, we observe very low mismatch scores in LUAC compared to LUAJIT and PY-LUA-PARSER. This reflects the maturity level of the LUAC implementation, which is the de facto parser for LUA. For the JSON SUTs, PARSON stands out as having a high number of discrepancies. For XML, it is PUGIXML which has a high number of discrepancies.

The SUTs for the URL format were surprisingly tolerant to malformed inputs. As an example, CURL allows one to three slashes after the scheme component to still be considered a valid URL.[2] Moreover, the SUTs accept paths that have redundant slashes as valid, whereas this is not permitted by the reference ANTLR grammar. Another difference we observe is that SUTs do not implement the same URL schemas. Both of these factors (permissiveness and disagreements) contributed to the relatively high number of issues.

#### 4.6.3 *Manual investigation of discrepancies.*

To fully address *RQ2*, we decided to manually inspect the issues found by our differential testing. On inspection, we noted a significantly high occurrence of duplicate issues. Generally, each system under test (SUT) presents no more than six distinct issues. Our initial step involved deduplicating all identified issues. Six false positives were identified and discarded. A false positive is determined by scrutinising the specification of the system under test (SUT). Specifically, an issue is classified

[2] https://curl.se/docs/url-syntax.html

as a false positive if the SUT specification is found to account for the observed discrepancy. We identified several reject-valid cases with LUA and URL subjects, see Table 2. On inspection, we find that most inputs rejected by URL SUTs were due to semantic errors. For instance, port numbers that are out of range were rejected by the SUTs. ARIA2 requires URLs with the FTP scheme to include file paths. We find a similar case with LIBXML2. The LIBXML2 parser rejects XML inputs containing invalid character reference values such as `&#2;`. On the other hand, the LUAJIT and PY-LUA-PARSER parsers were not implemented fully, as a result they were unable to parse certain valid LUA constructs. For example, LUAJIT rejects programs that contain floor division `//` or bitwise exclusive OR `~` symbols.[3]

We reported the remaining accept-invalid cases to the developers, as summarised in Table 3. Out of the 17 issues reported, we have received responses for 15 reports so far: 9 have been fixed, 4 have been confirmed, and 2 have been rejected. As a result of one of our reports, one critical CVE has been issued.

Our testing found problems in the grammars for LUA, XML, and URL provided in the ANTLR repository. We reported three such issues, and all of them have been fixed. For instance, we encountered an issue with the XML grammar, where tag names were not permitted to start with the underscore symbol. This contradicted the XML specification [53]. Upon identifying the discrepancy, we promptly reported and corrected the relevant grammar rule. This goes to show that hand writing grammars is prone to error and that fuzzing that deviates slightly from inputs expected by the grammar can be valuable in finding such errors.

We reported issues demonstrating that PY-LUA-PARSER and FAST-XML-PARSER are too permissive in relation to their ANTLR grammars. This is a typical feature of newly developed parsers. For instance, PY-LUA-PARSER incorrectly allows uninitialised declarations of global variables.

URLs containing semicolons in the userinfo part, such as `http://a;b:c@xyz` , were rejected by WGET but accepted by CURL and ARIA2. The userinfo segment is incorrectly parsed by WGET as part of the hostname, leading to DNS resolution errors. This bug in WGET's URL parsing leads to security vulnerabilities due to its incorrect handling of semicolons in the userinfo segment, which violates the URI standard defined in RFC 3986. This misinterpretation can cause authentication failures and expose sensitive credentials. It opens the door to phishing and spoofing attacks by allowing attackers to manipulate host headers and redirect users to malicious servers. Additionally, it can result in incorrect DNS resolution, making systems vulnerable to man-in-the-middle attacks where attackers can intercept and manipulate data. Furthermore, the insecure handling of userinfo data can lead to unintended exposure of sensitive information, such as usernames and passwords, in logs or error messages. Given the security implications of this bug, it was confirmed and fixed by the developer,[4] and a critical CVE was subsequently assigned to it.[5]

Additionally, we found that CJSON accepts invalid UNICODE values such as `ur282`. The CJSON developers have acknowledged that the accepted input has an invalid UTF-8 character. For PARSON, we reported two issues relating to invalid inputs such as `{} {}` and `[true, ]` being accepted despite not conforming to the JSON format. However, while the developers acknowledged the issues, they have decided not to fix them,[6] citing the robustness principle, also known as Postel's law [55]. According to this principle, programs should be permissive in what they accept, and conservative (format-conforming) in what they generate. We argue here that programs that follow this design

[3] LUAJIT developers acknowledge that LUAJIT is fully upwards-compatible with LUA 5.1, and that some missing operators will be included in future releases: https://github.com/LuaJIT/LuaJIT/issues/1158

[4] https://lists.gnu.org/archive/html/bug-wget/2024-06/msg00005.html

[5] CVE-2024-38428, https://nvd.nist.gov/vuln/detail/cve-2024-38428

[6] https://github.com/kgabis/parson/issues/194

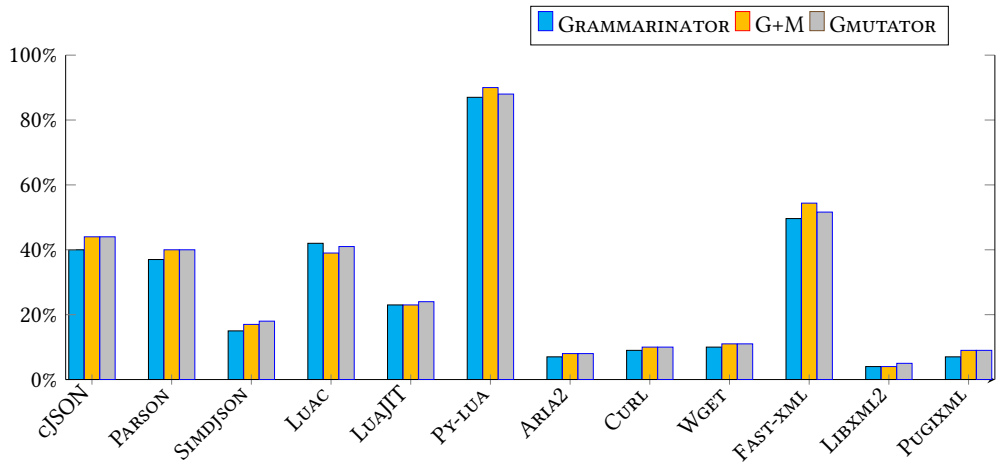


Fig. 3. Total branch coverage achieved in a 24 h run for every (generation tool, SUT) pair.

guideline cannot be adequately tested with a precise grammar, and a more permissive grammar is needed to exercise the full range of inputs.

#### 4.6.4 *Crashes and coverage.*

To answer *RQ3*, for each SUT, we recorded the total branch coverage achieved by every tool on a 24 h run. The results are shown in Figure 3. Overall, we see that the total coverage achieved on average is small. This is partly because the codebases of SUTs include unreachable code, but mainly because our test drivers and generators do not target semantic analysis code. G+M covers more branches in Py-lua-parser and Fast-xml-parser. These programs are purely parsing programs. As a result, G+M is sufficiently capable of covering significant portions of parser code, and error handling code as well. In contrast, Gmutator has a slight advantage in Luac, LuaJIT, Simdjson, and Libxml2. This is because Gmutator generates both well-formed and malformed inputs. Subsequently, Gmutator covers more error handling code (at the parsing stage) than Grammarinator, and more core functionality code than G+M.

G+M found a crash in cJSON. Re-running the crash-inducing input with a version of cJSON compiled with AddressSanitizer [46] revealed that this was due to a heap buffer overflow when reading strings with unbalanced double quotation marks, such as `"""` . Because our fuzzing campaign was conducted without the use of sanitisers, Gmutator happened not to trigger this crash. However, we confirmed that Gmutator did generate inputs that reveal this problem when AddressSanitizer is enabled. We did not report this bug, since it is no longer present in the latest version of cJSON.

Figure 3 shows that there are overall differences between the code coverage achieved by Gmutator and G+M on the SUTs used in our study, including cases where Gmutator is the overall winner (e.g. LuaJIT) and cases where G+M is the overall winner (e.g. Py-lua-parser). However, these results do not indicate the extent to which the tools complement one another with respect to the coverage that they achieve. In relation to complementarity, Table 4 reports differential coverage for Gmutator and G+M.

The table shows that the tools almost always complement one another to some extent, and that for some SUTs one tool achieves substantial additional coverage compared with the other (e.g. Gmutator markedly outperforms G+M on LuaJIT and Libxml2, while the converse is true for

Table 4. Differential line coverage between G+M and GMUTATOR. The highest value for each SUT is in bold.

| Tool | CJSON | PARSON | SIMDJSON | LUAC | LUAJIT | PY-LUA | ARIA2 | CURL | WGET | FAST-XML | LIBXML2 | PUGIXML |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **GMUTATOR** | **10** | **4** | **116** | **79** | **131** | 7 | 0 | 5 | 3 | 1 | **183** | **8** |
| **G+M** | 6 | 1 | 33 | 20 | 22 | **16** | **58** | **192** | **9** | **16** | 88 | 2 |

ARIA2 and CURL). GMUTATOR is the overall winner with respect to the "uniquely-covered lines" metric, in that it achieves more uniquely-covered lines than G+M for more SUTs.

To gain a deeper understanding of the observed coverage differences, we manually inspected the unique lines covered by each tool and mapped them to the relevant input features that induced this coverage. We summarise the primary factors contributing to GMUTATOR's enhanced coverage in two points:

(1) **Reachability of grammar rules**: The ANTLR format features auxiliary rules used to ignore whitespace and comments during lexical analysis. A special rule named `WS` is used to define the form of whitespace. E.g., for JSON this special rule indicates that tabs should be considered whitespace. Another special rule, `COMMENT`, is used to define the form of comments. For example, LUA's multi-line comments, which comprise strings starting with `--[` and ending with `]`, such as `--[=[ abc ]=]`, are defined via the following two rules:

```
COMMENT    : '--[' NESTED_STR ']' ;
NESTED_STR : '=' NESTED_STR '=' | '=[' .*? ']=' ;
```

These rules are defined in the ANTLR grammar for a given language, but they are not explicitly referenced from any other rules of the grammar. In particular, a random walk of the grammar rules commencing at the grammar's "start" rule would not traverse whitespace or comment rules. Therefore, G+M would have a hard time producing such syntactic elements. During the first stage, its GRAMMARINATOR component (which involves randomly walking an input grammar in this way) would not be able to use the `WS` and `COMMENT` rules when generating inputs, meaning the corresponding syntactic elements are absent from generated inputs. In its second stage, clearly the random mutations that G+M applies would be extremely unlikely to introduce such syntactic elements. By contrast, the "introduce choice" grammar mutation of GMUTATOR (see §3.1) can make the `WS` and `COMMENT` rules reachable from the grammar's "start" rule, by changing a use of some existing rule to a choice between the existing rule and one of these helper rules. Our manual analysis shows that this leads to coverage of additional lines in JSON and LUA SUTs.

(2) **Generation of specific input sequences**: Certain input sequences are more easily produced through the grammar mutations of GMUTATOR than through the random string mutations of G+M. For example, a sequence of characters following a valid JSON object is highly unlikely to be generated by random string mutations, but is readily achievable with grammar mutations. A concrete example is the invalid input `{age:40} true`. This is readily generated by GMUTATOR due to its "concatenation" rule (see §3.1), but would be very unlikely to be generated by G+M. This type of input resulted in the coverage of new lines in CJSON with GMUTATOR.
A similar case was observed with LUAC, where the LUA grammar does not allow for negative integers, yet the grammar mutations of GMUTATOR successfully generate such sequences. A special instance of this type of LUA input is `x=-2147483648/-1`. This input has the potential to trigger an integer overflow. Specifically, integer `-2147483648` is the minimum value for a

```
1  lua_Integer luaV_idiv (lua_State *L, lua_Integer m, lua_Integer n) {
2    if (l_unlikely(l_castS2U(n) + 1u <= 1u)) {  /* special cases: -1 or 0 */
3      if (n == 0)
4        luaG_runerror(L, "attempt to divide by zero");
5      return intop(-, 0, m);   /* n==-1; avoid overflow with 0x80000...//-1 */
6    }
7    else {
8      lua_Integer q = m / n;  /* perform C division */
9      if ((m ^ n) < 0 && m % n != 0)  /* 'm/n' would be negative non-integer? */
10       q -= 1;  /* correct result for different rounding */
11     return q;
12   }
13 }
```

Listing 1. Function from the source code of Luac that divides two integers. Special cases are handled separately, such as when the denominator is 0 or -1.

32-bit signed integer, and the result of dividing it by -1, `2147483648`, exceeds the maximum value for a 32-bit signed integer, `2147483647`. To handle such cases, the code in Luac has a special check in function `luaV_idiv`. Listing 1 shows the implementation of this function. On line 5, the code specifically handles the case when the denominator `n` is -1. If the denominator is -1, the code returns the negation of the numerator `m`, effectively preventing the overflow that would occur from the division. Gmutator was able to repeatedly exercise this particular branch of code by generating inputs that would trigger this check. However, it is unlikely that G+M would produce such an input. This is because generating such an input requires two specific conditions to be met: the presence of the minus symbol and a division expression with 1 as the denominator. Additionally, a string mutation must place the minus sign immediately before the denominator. The probability of both conditions occurring together is low, making this input less likely to be generated by G+M.

From Table 4, we observe that G+M outperforms Gmutator for all url programs, Py-lua-parser and Fast-xml-parser. A detailed investigation reveals the following insights:

(1) **Error handling code**: A significant portion of the unique line coverage achieved by G+M on url programs is in the code responsible for parsing program arguments. Since the URL input is supplied as a command-line option, parts of the URL may be mistakenly processed as separate program options, triggering the activation of error-handling code for program options. For instance, URLs that begin with the dash symbol such as `-http://example.com` were parsed as program options. Py-lua-parser and Fast-xml-parser are relatively small SUTs. These programs are mostly parsers and contain relatively large portions of error handling code. The generation of highly invalid inputs allowed G+M to cover significant error handling code for these targets. For instance, Fast-xml-parser includes dedicated code to handle invalid XML inputs containing sequences of elements with identical tag names, such as `<A/><A/>`. By employing string duplication mutations, G+M can easily generate these sequences. In contrast, Gmutator is unlikely to produce two xml elements with identical tag names.

(2) **Generation of specific input sequences**: The string mutations employed by G+M were able to generate certain unique inputs. For example, zone identifiers that can trail IPv6 addresses are not part of the URL grammar but can be easily produced by a duplication mutation. An example of such mutation is the following:

`https://%29%AB@[::]` ⟶ `https://%29%AB@[::%AB@[::]`

The duplicated characters in blue represent an invalid zone identifier. This mutated URL triggered the execution of code responsible for processing zone identifiers. The current grammar mutations employed by GMUTATOR are not capable of generating such elements. A simple solution to remedy this would be to introduce a new grammar mutation that allows random insertion and duplication of non-terminals and tokens.

Considering the results in Table 4 and our discussion above, GMUTATOR uniquely covers slightly more code than G+M. More importantly, we find that the two tools complement each other.

## 5 RELATED WORK

Our approach builds upon grammar-based fuzzing [2, 32, 41, 44, 54, 56] and the GRAMMARINATOR [31] tool in particular. The effectiveness of grammar-based fuzzing depends on the quality of the grammar; because the generator is blackbox, it is unable to exploit knowledge of the program's implementation. While it is possible to build grammar-based fuzzers that enforce semantic constraints to generate valid inputs [32, 56], this requires significant effort and such fuzzers are only available for a few domains.

Automatically mining input grammars from programs can reduce the manual effort required in building grammars. With no specification or seed inputs, pFuzzer [38] can mine an input grammar of a program, by instrumenting the program and tracking byte comparisons at runtime. However, pFuzzer only works with recursive descent parsers, written in C. Grammars can also be synthesised from sample inputs [4, 23, 29, 35]. These techniques are useful when the program source code is not available, however a seed corpus exercising all of the target grammar rules is needed.

Instead of mining grammars from scratch, GMUTATOR and G+M build on top of existing grammars, and attempt to generate inputs at the "edge" of what the input grammar allows.

A similar approach to ours is Ccoft [52], which is a mutator that operates on Protobuf objects. Ccoft converts an input grammar into a Protobuf format, then uses the libprotobuf-mutator to mutate instantiations of the Protobuf format. Although the tool detected many *reject-valid* and *accept-invalid* bugs, it was only targeted towards testing of C++ compiler front-ends and was not shown to generalise beyond C++ subsets.

FuzzTruction [3] is another approach that introduces subtle mutations to generator applications. The approach is successful at generating almost-valid inputs. The tool is effective with highly structured formats, especially those that go through complex transformations like compression and encryption. However, the approach only works when a generator application is available, which is not the case for most program parsers.

AFLSmart [44] extends AFL with smart mutators. Using a structure template, it performs structure-level mutations on inputs to generate new valid inputs. By relaxing the structural integrity of inputs, AFLSmart was able to expose a critical vulnerability in a popular parser library. GMUTATOR draws inspiration from these findings and builds a more general solution by synthesising approximations of input grammars.

An alternative class of solutions aimed at generating edge case inputs involves byte-level mutation fuzzers. The AFL fuzzer [57], for instance, operates by mutating well-formed seed inputs and is particularly effective for programs that consume chunk-based binary formats. Similarly, libFuzzer [37] functions akin to AFL but is better suited for fuzzing libraries and their APIs. G+M, drawing inspiration from AFL and libFuzzer, implements a subset of their mutations. However, unlike AFL and libFuzzer, G+M does not rely on coverage feedback or seed inputs. Instead, it utilises a grammar-based generator to produce valid inputs and subsequently applies mutations to them.

## 6 CONCLUSION

Grammar-based fuzzing is a technique used to generate well-formed inputs. In this work, we explored some limitations of this technique and presented two mutation-based approaches that complement it: grammar mutation and string-level mutation.

We introduce the novel concept of grammar mutation alongside string-level mutation, offering two distinct yet complementary approaches to input generation. In contrast to string-level mutations, grammar mutations apply mutations at the grammar level, thereby creating approximations of a reference grammar. The mutated grammar is then used for input generation. String-level mutations, on the other hand, apply byte-level mutations to grammar-generated inputs. We demonstrated how both grammar mutation and string-level mutation are useful at producing edge test cases. We identified several parsing discrepancies that are the product of incomplete or inaccurate implementations, as well as imprecise grammars. Most of our reports have resulted in bug fixes, and in a critical CVE security vulnerability being issued

Although the concept of grammar mutation works fundamentally differently from string-level mutation, our findings did not reveal significant differences in the performance of either approach. However, we acknowledge that this concept is still in its infancy and holds promise for further exploration. In future work, we aim to extend the application of grammar mutation beyond ANTLR to encompass a broader spectrum of input specification formats. For input specifications that include semantic constraints, we plan to target our mutations towards the semantics definitions. Another research direction would be to develop new grammar mutations and utilise code coverage feedback to identify the most effective ones.

## 7 DATA AVAILABILITY

The complete implementation and experimental data can be accessed at https://doi.org/10.5281/zenodo.10781796 and https://srg.doc.ic.ac.uk/projects/gmutator.

## ACKNOWLEDGEMENTS

This project has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation program (grant agreement 819141) and from the UK Engineering and Physical Sciences Research Council through a PhD studentship and grant EP/R006865/1.

## REFERENCES

[1] Antlr. 2020. ANTLR v4 Grammars. https://github.com/antlr/grammars-v4. Online; accessed 7 May 2023.
[2] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *Proc. of the 26th Network and Distributed System Security Symposium (NDSS'19)* (San Diego, CA, USA). https://doi.org/10.14722/ndss.2019.23412
[3] Nils Bars, Moritz Schloegel, Tobias Scharnowski, Schiller Nico, and Thorsten Holz. 2023. Fuzztruction: Using Fault Injection-based Fuzzing to Leverage Implicit Domain Knowledge. In *Proc. of the 32nd USENIX Security Symposium (USENIX Security'23)* (Boston, MA, USA).
[4] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing Program Input Grammars. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'17)* (Barcelona, Spain). https://doi.org/10.1145/3062341.3062349
[5] Bachir Bendrissou. 2023. *Accepting invalid array.* https://github.com/kgabis/parson/issues/194
[6] Bachir Bendrissou. 2023. *Accepting invalid integers.* https://github.com/DaveGamble/cJSON/issues/718
[7] Bachir Bendrissou. 2023. *Ambiguity in url grammar.* https://github.com/antlr/grammars-v4/pull/3718
[8] Bachir Bendrissou. 2023. *EOF not enforced.* https://github.com/kgabis/parson/issues/195
[9] Bachir Bendrissou. 2023. Fail to reject an invalid escape sequence. https://github.com/boolangery/py-lua-parser/issues/30
[10] Bachir Bendrissou. 2023. *Fail to reject invalid escape character.* https://github.com/DaveGamble/cJSON/issues/736

[11] Bachir Bendrissou. 2023. *Fail to reject multiple root nodes.* https://github.com/NaturalIntelligence/fast-xml-parser/issues/542
[12] Bachir Bendrissou. 2023. Fail to reject unassigned global variable declaration. https://github.com/boolangery/py-lua-parser/issues/29
[13] Bachir Bendrissou. 2023. *Failure to parse chained comparisons.* https://github.com/boolangery/py-lua-parser/issues/56
[14] Bachir Bendrissou. 2023. *Failure to reject incorrect inputs: missing function call arguments.* https://github.com/boolangery/py-lua-parser/issues/50
[15] Bachir Bendrissou. 2023. *Failure to reject incorrect inputs: name token.* https://github.com/boolangery/py-lua-parser/issues/49
[16] Bachir Bendrissou. 2023. *Literal string gets parsed as Lua code.* https://github.com/boolangery/py-lua-parser/issues/51
[17] Bachir Bendrissou. 2023. *Parsing an invalid XML element.* https://github.com/NaturalIntelligence/fast-xml-parser/issues/618
[18] Bachir Bendrissou. 2023. *Parsing Lua long comment as short comment.* https://github.com/antlr/grammars-v4/issues/3741
[19] Bachir Bendrissou. 2023. *Underscore symbol not allowed in XML NameStartChar.* https://github.com/antlr/grammars-v4/issues/3758
[20] Bachir Bendrissou. 2023. *Validation of invalid XML declarations.* https://github.com/NaturalIntelligence/fast-xml-parser/issues/616
[21] Bachir Bendrissou. 2024. *Semicolon not allowed in userinfo.* https://lists.gnu.org/archive/html/bug-wget/2024-06/msg00005.html
[22] Bachir Bendrissou, Cristian Cadar, and Alastair F. Donaldson. 2023. Grammar Mutation for Testing Input Parsers (Registered Report). In *Proceedings of the 2nd International Fuzzing Workshop, FUZZING 2023, Seattle, WA, USA, 17 July 2023*, Marcel Böhme, Yannic Noller, Baishakhi Ray, and László Szekeres (Eds.). ACM, 3–11. https://doi.org/10.1145/3605157.3605170
[23] Bachir Bendrissou, Rahul Gopinath, and Andreas Zeller. 2022. "Synthesizing Input Grammars": A Replication Study. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'22)* (San Diego, CA, USA). https://doi.org/10.1145/3519939.3523716
[24] CLOC - count Lines of Code [n. d.]. CLOC - Count Lines of Code. http://cloc.sourceforge.net/.
[25] Douglas Crockford. 2017. cjson. https://json.org.
[26] Eliott Dumeix. 2023. A Lua parser and AST builder written in Python. https://github.com/boolangery/py-lua-parser.
[27] Krzysztof Gabis. 2023. Lightweight JSON library written in C. https://github.com/kgabis/parson.
[28] Dave Gamble. 2023. Ultralightweight JSON parser in ANSI C. https://github.com/DaveGamble/cJSON.
[29] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining Input Grammars from Dynamic Control Flow. In *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'20)* (Online). https://doi.org/10.1145/3368089.3409679
[30] Amit Kumar Gupta. 2023. Fast XML Parser. https://github.com/NaturalIntelligence/fast-xml-parser.
[31] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. Grammarinator: A Grammar-Based Open Source Fuzzer. In *Proc. of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST'18)* (Lake Buena Vista, FL, USA). https://doi.org/10.1145/3278186.3278193
[32] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proc. of the 21st USENIX Security Symposium (USENIX Security'12)* (Bellevue, WA, USA).
[33] Roberto Ierusalimschy, Waldemar Celes, and Luiz Henrique de Figueiredo. 2023. Lua. https://www.lua.org/manual/5.3/manual.html.
[34] Arseny Kapoulkine. 2022. Light-weight C++ XML processing library. https://pugixml.org.
[35] Neil Kulkarni, Caroline Lemieux, and Koushik Sen. 2021. Learning Highly Recursive Input Grammars. In *Proc. of the 36th IEEE International Conference on Automated Software Engineering (ASE'21)* (Melbourne, Australia). https://doi.org/10.1109/ASE51524.2021.9678879
[36] Daniel Lemire, Geoff Langdale, and John Keiser. 2023. Fast parser for large JSON files. https://simdjson.org.
[37] LibFuzzer 2022. LibFuzzer website. http://llvm.org/docs/LibFuzzer.html.
[38] Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. 2019. Parser-Directed Fuzzing. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'19)* (Phoenix, AZ, USA).
[39] Jake Miller. 2021. An Exploration of JSON Interoperability Vulnerabilities. https://labs.bishopfox.com/tech-blog/an-exploration-of-json-interoperability-vulnerabilities [Online; accessed 12-May-2021].
[40] Hrvoje Nikšić. 2023. Network utility to retrieve files from the World Wide Web. https://www.gnu.org/software/wget/.

[41] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'19)* (Beijing, China). https://doi.org/10.1145/3293882.3330576
[42] Mike Pall. 2022. Just-In-Time (JIT) compiler for the Lua programming language. http://luajit.org.
[43] Terence Parr. 2013. *The Definitive ANTLR 4 Reference* (2nd ed.). Pragmatic Bookshelf.
[44] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. 2021. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering (TSE)* 47, 9 (2021), 1980–1997. https://doi.org/10.1109/TSE.2019.2941681
[45] Hamad Ali Al Salem and Jia Song. 2019. A Review on Grammar-Based Fuzzing Techniques. *International Journal of Computer Science and Security* 13, 3 (June 2019).
[46] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proc. of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)* (Boston, MA, USA).
[47] Nicolas Seriot. 2016. Parsing JSON is a minefield. https://seriot.ch/parsing_json.php [Online; accessed 15-Sep-2020].
[48] Prashast Srivastava and Mathias Payer. 2021. Gramatron: Effective grammar-aware fuzzing. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'21)* (Online). https://doi.org/10.1145/3460319.3464814
[49] Daniel Stenberg. 2023. Command line tool and library for transferring data with URLs. https://curl.se.
[50] The GNOME Project. 2023. XML toolkit implemented in C. https://gitlab.gnome.org/GNOME/libxml2.
[51] Tatsuhiro Tsujikawa. 2021. Utility for downloading files. https://aria2.github.io.
[52] Haoxin Tu, He Jiang, Zhide Zhou, Yixuan Tang, Zhilei Ren, Lei Qiao, and Lingxiao Jiang. 2023. Detecting C++ Compiler Front-End Bugs via Grammar Mutation and Differential Testing. *IEEE Transactions on Reliability* 72, 1 (March 2023), 1–15. https://doi.org/10.1109/TR.2022.3171220
[53] W3C. 2008. Extensible Markup Language (XML) 1.0 (Fifth Edition). https://www.w3.org/TR/xml/.
[54] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing. In *Proc. of the 41st International Conference on Software Engineering (ICSE'19)* (Montreal, Canada). https://doi.org/10.1109/ICSE.2019.00081
[55] Wikipedia. 2023. Robustness principle. https://en.wikipedia.org/wiki/Robustness_principle.
[56] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'11)* (San Jose, CA, USA). https://doi.org/10.1145/1993498.1993532
[57] Michal Zalewski. [n. d.]. Technical "whitepaper" for afl-fuzz. http://lcamtuf.coredump.cx/afl/technical_details.txt.