Grammar Mutation for Testing Input Parsers (Registered Report)

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

ABSTRACT

Grammar-based fuzzing is an effective method for testing programs that consume structured inputs, particularly input parsers. A prerequisite of this method is to have a specification of the input format in the form of a grammar. Consequently, the success of a grammar-based fuzzing campaign is highly dependent on the available grammar. If the grammar does not accurately represent the input format, or if the system under test (SUT) does not conform strictly to that grammar, there may be an impedance mismatch between inputs generated via grammar-based fuzzing 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 fuzzing, by construction, will not yield such edge case inputs.

To overcome these limitations, we present GMUTATOR, an approach that mutates an input grammar and leverages the GRAMMARINATOR fuzzer to produce inputs conforming to the mutated grammars. As a result, GMUTATOR can find inputs that do not conform to the original grammar but are (wrongly) accepted by an SUT. In addition, GMUTATOR-generated inputs have the potential to increase SUT code coverage compared with the standard approach. We present preliminary results applying GMUTATOR to two JSON parsing libraries, where we are able to identify a few inconsistencies and observe an increase in covered code. We propose a plan for a full experimental evaluation over four different input formats—JSON, XML, URL and Lua—and twelve SUTs (three per input format).

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Grammar-based fuzzing, mutant grammars, input parsers

ACM Reference Format:
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 '23), July 17, 2023, Seattle, WA, USA*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3605157.3605170

1 INTRODUCTION

Thoroughly fuzz-testing a software system requires inputs that get past its parser. Invalid inputs that are rejected by the parser are valuable for testing error-handling code in the front-end, but cannot exercise deeper functionality of the system under test (SUT). A solution to the problem of generating valid inputs is *grammar-based fuzzing* [2, 14, 23, 25, 35, 37], where inputs are generated based on a grammar that should capture the intended input format.

A problem with grammar-based fuzzing is that there may be a mismatch between the grammar of the input format that the SUT is *supposed* to accept, and the implicit grammar associated with the inputs that the SUT *actually* accepts in practice. One reason for this is that writing a parser by hand is difficult and prone to errors; for instance, various parsers have been shown not to implement the JSON format correctly [21, 27]. Furthermore, as the SUT is updated over time, there is the potential for the front-end of an SUT to get out of sync with the input format it is supposed to accept. An important and under-explored application of fuzzing is to detect such inconsistencies—cases where the inputs actually accepted by an SUT differ from the inputs described by a grammar for the associated input format.

A related property of grammar-based fuzzing is that it is designed to only produce valid inputs. While in many ways this property is a *feature*—valid inputs have the potential to exercise deep parts of the SUT—it is also a limitation: grammar-based fuzzing does not produce "edge case" inputs that *almost* conform to the required grammar, but deviate from the grammar in small, seemingly innocuous ways. As well as the potential for such inputs to be incorrectly accepted by an SUT (as discussed above), they have the potential to trigger vulnerabilities in error-handling code paths in the SUT's front-end, and to find subtle defects that may not be triggered by more drastically-invalid inputs (such as purely random input strings, or inputs obtained by applying byte-level mutations to originally-valid inputs in the style of mutation-based fuzzers such as AFL [38] and libFuzzer [19]).

These problems with grammar-based fuzzing are particularly acute in the context of *blackbox* fuzzing [14, 28, 33] where, without any feedback signal from the SUT, one cannot learn the implicit grammar that the SUT accepts by trial and error. Even when it is possible to instrument the SUT, e.g. with coverage information, the use of an upfront grammar can accelerate the fuzzing process [35], but the use of such a grammar is subject to the above problems.

In this work, we investigate the idea of enhancing grammar-based blackbox fuzzing with *grammar mutations*, so that inputs are generated from slightly corrupted grammars. The aim is to generate inputs that diverge to some degree from the correct specification of an input format.

FUZZING '23, July 17, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0247-1/23/07.
https://doi.org/10.1145/3605157.3605170

In brief, our idea is as follows. Given a grammar G specifying the input format that an SUT is supposed to accept, we derive a *mutant grammar* G' by applying one or more *mutations* to the production rules of G. The grammar G' can then be used as a basis for generating inputs that do *not* conform to G, but—due to the close relationship between G and G', are in large part very similar to inputs that *do* conform to G. These *almost G-valid* inputs can then be applied to the SUT, with the potential to identify (a) inputs that the SUT accepts when it should not (the very existence of these inputs may constitute bugs in the SUT, and some of these inputs might reveal additional coverage or trigger crashes in deeper parts of the SUT), and (b) inputs that achieve additional coverage of the SUT's front-end error handling code paths (potentially revealing crashes in those code paths).

To experiment with these ideas, we present a new blackbox fuzzing tool, GMUTATOR, that takes as input an ANTLR [1] grammar specifying a target input format. The ANTLR tool is first used to produce a reference parser. Then, GMUTATOR mutates the grammar, producing multiple *mutant grammars*. The GRAMMARINATOR grammar-based fuzzing tool [13] is used to produce inputs for each mutant grammar. Inputs that are accepted by the SUT but rejected by the reference parser (i.e., they do not conform to the original grammar) are flagged for investigation: these represent invalid inputs that are accepted by the SUT. This has two benefits: first, it highlights discrepancies between input specification and implementation; secondly, it exercises the SUT in ways that would be missed by a conforming grammar-based fuzzer.

We present preliminary results investigating the effectiveness of our idea and tool for the JSON input format. Using GMUTATOR we have found JSON-like inputs that do not conform to the JSON grammar, but that are accepted by off-the-shelf JSON parsing libraries. We reported these issues to the library developers. One of them is under discussion as a potential bug, showing that GMUTATOR has the potential to find bugs that traditional grammar-based fuzzing would miss. The other bug was closed as intended behaviour, with the developer of the parsing library in question stating their intention to be permissive with respect to the inputs that are accepted; this shows that GMUTATOR can be useful for identifying cases where thorough grammar-based testing of a particular SUT would require a more permissive grammar. We also present experimental results showing that GMUTATOR is able to achieve additional code coverage on these JSON parsing libraries compared to using straightforward grammar-based fuzzing.

We then describe a planned set of controlled experiments to evaluate our idea at a larger scale with respect to four input formats in total, varying in their complexity (JSON, XML, URL and Lua) and three SUTs per input format. We will investigate the number and nature of discrepancies that GMUTATOR can find, and the effect on code coverage associated with grammar-based fuzzing using mutant grammars.

This registered report makes the following contributions:

(1) The idea of using mutant grammars to extend the reach of grammar-based fuzzing, allowing fuzzing with respect to multiple approximations of an input format, to enable the identification of discrepancies between the specification of the input format and the implementation of the SUT;
(2) The implementation of this idea as a blackbox fuzzing tool, GMUTATOR, that works on the widely-used ANTLR grammar format and leverages the GRAMMARINATOR grammar-based fuzzer;
(3) A preliminary investigation of the practical effectiveness of the idea using the JSON format, including details of identified discrepancies and developer responses to these discrepancies;
(4) A plan for a larger experimental evaluation over twelve SUTs covering four input formats in total.

The rest of the paper is structured as follows. We first illustrate our approach on the JSON format and discuss our preliminary results in §2. We then describe our GMUTATOR approach in detail in §3. We present the full evaluation plan in §4, including the research questions we want to address and the experimental methodology. We discuss related work in §5, and conclude the paper in §6.

2 PRELIMINARY RESULTS FOR JSON

Before going into details of our GMUTATOR tool, we illustrate the idea of grammar mutation and its effectiveness using an example.

JavaScript Common Object Notation (JSON) [6] is a standard format for representing structured data, and is commonly used as an interchange format between software tools. An ANTLR grammar for part of JSON is shown in Figure 1a, where the notation uses the Backus-Naur Form (BNF). 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, 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` numbers.

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.

For our preliminary evaluation, we used GMUTATOR to create such mutant grammars and employed them to generate inputs for testing two JSON parsers: cJSON [9] and PARSON [8], both written in C. The evaluation was run for one hour with 3 repetitions. For comparison, we also run GRAMMARINATOR [13], an open source blackbox grammar-based fuzzer, for the same amount of time on the same original JSON grammar.

GRAMMARINATOR generated 137,440 inputs in one hour. By contrast, GMUTATOR generated only 82,465 inputs, reflecting the additional overhead it incurs to create mutant grammars (1,268 for

```
json
   : value EOF ;
obj
   : '{' pair (',' pair)* '}'
   | '{' '}' ;
pair
   : STRING ':' value ;
arr
   : '[' value (',' value)* ']'
   | '[' ']' ;
value
   : STRING | NUMBER | obj | arr
   | 'true' | 'false' | 'null' ;
STRING
   : '"' (ESC | CHAR)* '"' ;
ESC
   : '\\' (["\\/bfnrt] | UNICODE) ;
UNICODE
   : 'u' HEX HEX HEX HEX ;
```

(a)

```
json
   : value (obj | EOF) ;
obj
   : '{' pair (',' pair)* '}'
   | '{' '}' ;
pair
   : STRING ':' value ;
arr
   : '[' value (',' value * )* ']'
   | '[' ']' ;
value
   : STRING | NUMBER | obj | arr
   | 'true' | 'false' | 'null' ;
STRING
   : '"' (ESC | CHAR)* '"' ;
ESC
   : '\\' (["\\/bfnrt] | UNICODE) ;
UNICODE
   : 'u' (STRING | HEX) HEX HEX HEX ;
```

(b)

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

this experiment). However, Gmutator managed to generate 7,793 inputs that are unique to the mutant grammars, i.e. they are not accepted by the original grammar.[1]

Extra coverage. The Grammarinator-generated inputs achieve branch coverage of 24% and 19% on cJSON and Parson, respectively. Despite generating fewer inputs, the inputs unique to mutant grammars allow Gmutator to increase branch coverage by 2% on cJSON and 3% on Parson.

Issues discovered. The inputs that are unique to the mutant grammars discovered several issues in the JSON parsers.

Mutation 3 led to the discovery of an issue in cJSON, which accepts invalid UNICODE values such as `ur282`. We are discussing this issue with the developers, who acknowledged that the accepted input has an invalid UTF-8 character.

Mutations 1 and 2 led to the discovery of two issues in Parson. Inputs such as `{} {}` and `[true, ]` are accepted by Parson, although they do not conform to the JSON format. However, while the developers acknowledged the issues, they have decided not to fix them, citing the robustness principle, also known as Postel's law [36]. 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 guideline cannot be adequately tested with a precise grammar, and a more permissive grammar is needed to exercise the full range of inputs. The examples discovered show how Gmutator can be useful in identifying where the JSON grammar used for grammar-based fuzzing of Parson would need to be more permissive.

3 GMUTATOR

We now describe Gmutator, our prototype tool for grammar mutation, and give details of the mutation operators that Gmutator incorporates.

Gmutator takes as input an ANTLR grammar, for example the (full version of the) grammar in Figure 1a. We chose to support the ANTLR format for two reasons: first, the ANTLR repository [1] features well-maintained grammar files for a number of input formats; secondly, the ANTLR tool can be used to automatically generate a reference parser for a given grammar, a feature that Gmutator makes use of as described below.

Gmutator applies a number of mutations to the rules of the given grammar, randomly selecting which types of mutation to apply and where to apply them. This leads to a new *mutant* grammar, as illustrated by the grammar in Figure 1b.

The following types of mutations are supported by Gmutator:

(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. `(...)?` ⟶ `(...)*`

---

[1] Despite the fact that each mutant grammar features at least one mutation, generating an input that is not accepted by the original grammar requires a mutated part of the grammar to be used during generation, and in a manner that actually causes a deviation from what the original grammar accepts.

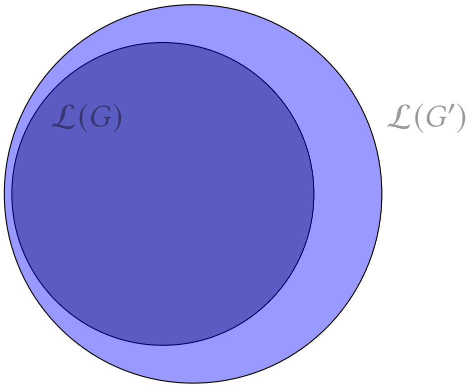

Figure 2: $\mathcal{L}(G)$ is the set of inputs derivable from grammar G, while $\mathcal{L}(G')$ is the set of inputs derivable from a mutant grammar G'. $\mathcal{L}(G')$ is a superset of $\mathcal{L}(G)$

(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.[2]
(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)`.

We chose these mutations because they are self-contained; that is, they do not introduce any new constructs that are not contained in the original grammar. This makes the mutation process dependent only on the original grammar. Additionally, self-contained mutations are normally subtle and do not lead to drastic changes, as they reuse existing grammar parts. In addition, the mutations are grammar-agnostic: they are straightforward to apply to an ANTLR description of any grammar. Our preliminary evaluation (§2) has already shown that these mutations can be effective at uncovering parsing discrepancies. In §6 we discuss the investigation of further mutations as an avenue for future work.

Given a grammar G, Gmutator produces a mutant grammar G' by randomly applying one or more mutations of the types described above. The default configuration is three mutations, which can be modified by the user. Each mutant grammar is transformed to a *generator* by Grammarinator, from which a sample of inputs can be generated. These inputs can then be executed on SUTs of interest. As is common in grammar-based fuzzing, Grammarinator (and thus also Gmutator) supports *blackbox* testing: the process of generating inputs is *not* guided by feedback from the SUT(s) in relation to previously-generated inputs.

By design, the mutations that Gmutator perform preserve the syntactical and semantic validity of ANTLR grammars; i.e. given an original well-formed ANTLR grammar, the mutant grammars that Gmutator produces are also well-formed and can thus be used by Grammarinator to drive the input generation process.

The grammar mutations we have designed have the property that each mutation monotonically increases the space of inputs that the grammar represents. As a concrete example, Mutation 1 in our running example (see §2) can generate both single-rooted JSON inputs and multi-rooted JSON inputs, while the original grammar can only generated single-rooted JSON inputs. As a more general example, the **concatenation** mutation outlined above does not replace a choice `foo | bar` with the concatenation `foo bar`; it instead adds `foo bar` as an *additional* choice, so that the grammar still has the potential to yield any input it could yield pre-mutation, as well as additional inputs corresponding to the new alternative.

More formally, given a grammar mutator M that takes as input a grammar G and produces a grammar G', M satisfies the monotonic acceptance property if and only if, for any string s:

$$s \in \mathcal{L}(G) \Rightarrow s \in \mathcal{L}(G'),$$

where $\mathcal{L}(G)$ denotes the set of inputs derivable from a grammar G. This is illustrated graphically in Figure 2.

The rationale for our design decision to use mutations that monotonically increase the input space that the grammar can generate 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 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.

4 PLANNED EVALUATION

We now detail the evaluation that we intend to conduct to assess the effectiveness of our techniques for grammar mutation and the Gmutator tool. We start by outlining the research questions our evaluation aims to answer (§4.1). We then discuss the target input formats we plan to study (§4.2), and the SUTs that consume these input formats that we plan to test (§4.3). Finally, we explain the procedure we plan to use for generating inputs and using generated inputs for testing (§4.4), and the metrics we plan to use in order to relate our findings back to our research questions (§4.5).

---

[2] This mutation could be generalised so that it would consider replacing any top-level regular expression with a more permissive one. We chose to implement the more restricted form of the mutation to maximise the chances of generating inputs that involve characters that are explicitly disallowed by the original grammar.

4.1 Research Questions

As a first 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 [26, 28, 32]. Importantly, it operates directly on the popular ANTLR format, which Gmutator also supports.

As a second baseline, we use the following approach, which we call Grammarinator+Mutations: Grammarinator is used to generate an input using a standard (non-mutated) grammar, then standard byte-level mutation operators, as implemented by fuzzers such as AFL, are used to modify the generated input before feeding it to an SUT.

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

- *RQ1*: To what extent can Gmutator and Grammarinator+Mutations 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 Grammarinator+Mutations 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

We plan to evaluate Gmutator with respect to grammars for four different input formats, with varying levels of complexity, ranging from regular to context-sensitive: URL, JSON, XML and Lua. We have obtained grammar definitions of these formats from the ANTLR GitHub repository [1].

The default ANTLR grammars are context-free and do not include any context-sensitive constraints. We have already started preliminary investigation into these grammars, and found that the lack of context-sensitivity makes Grammarinator generate many Lua and XML inputs that do not satisfy semantic validity constraints (even though they conform to the syntax specified by the grammar). This prompted us to add some constraints and make some simplifications to the XML and Lua grammars:

1. **XML:** We added constraints to the grammar to ensure that: a closing tag name must match the corresponding opening tag name; if the *declaration* tag is present (of the form `<?xml ...>`), then it must include the `version` attribute.
2. **Lua:** We added constraints to the grammar to ensure that: the `break` token can only appear inside loops; 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.

These adaptations and simplifications do not relate to our investigation of grammar mutation; they are needed even for standard grammar-based fuzzing to be useful for these input formats. When we present our full results, we will report on any further grammar adaptations that turn out to be needed.

4.3 Systems under Test

Original and mutated grammars for the input formats of §4.2 will be used to generate test inputs for a number of target SUTs. 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) as an indication of their varying complexity. We chose recent versions of SUTs and, for reproducibility, indicate which versions we will use in our full 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, especially when compiled with sanitisers); and choosing at least one SUT per input format 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 and Testing

The Gmutator tool serves as a complementary approach to existing grammar-based fuzzers. The primary objective of Gmutator 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 Gmutator is at discovering inputs that the SUT accepts but the original grammar does not, and whether it 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 generating overly large inputs or get stuck during execution, 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).

Grammarinator+Mutations generation. For this setup, which involves generating inputs using a standard grammar and subsequently mutating them, the same process will be used as for Grammarinator above, except that after each input is generated, between one and three random mutations will be applied to the input (where the number of mutations is also chosen at random). We will consider the following mutation operators, inspired by those used in coverage-guided fuzzers (such as AFL and libFuzzer):

- Deleting a randomly-chosen contiguous sequence of bytes from the input;
- Duplicating a randomly-chosen contiguous sequence of bytes at a random position in the input;

Table 1: The systems under test on which we plan to perform our full evaluation.

SUT	Input format	Language	Version	LOC	Notes
cJSON [9]	JSON	C	1.7.8	2,348	Ultralightweight JSON parser
Parson [8]	JSON	C	1.4	2,179	Lightweight JSON library
simdjson [18]	JSON	C++	3.2.0	10,356	Fast parser for large JSON files
luac [15]	Lua	C	5.4.4	17,327	Parser component of the official Lua implementation
LuaJIT [24]	Lua	C	2.1.0	49,725	Just-In-Time (JIT) compiler for the Lua programming language
py-lua-parser [7]	Lua	Python	3.1.1	3,823	Lua parser and AST builder written in Python
aria2 [31]	URL	C++	1.36.0	93,223	Utility for downloading files
curl [29]	URL	C	8.0.0	146,879	Command-line tool for transferring data with URLs
Wget [22]	URL	C	1.21.3	79,974	Program that retrieves content from web servers
fast-xml-parser [12]	XML	JavaScript	4.2.2	1,857	Tool that validates XML and parses XML to JS Object
libxml2 [30]	XML	C	20902	215,759	XML parser and toolkit originally developed for the GNOME Project
pugixml [16]	XML	C++	1.13	22,853	XML processing library

- Inserting a keyword drawn from an input format-specific dictionary at a random position in the input. The dictionary for each input format will be constructed based on fixed tokens appearing in the associated grammar.

Gmutator generation. Gmutator, on the other hand, 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.

Differential testing between the SUT and the parser generated from the original grammar. For Gmutator-, Grammarinator- and Grammarinator+Mutations-generated inputs, we will record how many are valid vs. invalid according to the original grammar. This will be achieved by attempting to parse each input using the ANTLR-generated parser derived from the original (non-mutated) grammar.

All Grammarinator-generated inputs should be valid, by construction. In contrast, Gmutator-generated inputs might not be valid, since the mutations that Gmutator applies to a grammar monotonically increase the set of inputs the grammar can generate (see §3). We expect many Grammarinator+Mutations-generated inputs to be invalid, but the mutations that are applied to Grammarinator-generated inputs are not guaranteed to affect validity.

For each SUT, we will then identify inputs for which the SUT and the ANTLR-generated parser disagree on validity. Cases where invalid inputs are accepted by an SUT are of particular interest. (The opposite case, where grammar-valid inputs are rejected by an SUT are more likely to be due to missing semantic constraints in the grammar.) This form of testing will allow *RQ1* above to be answered. Of particular interest will be the relative ability of our supposedly-smarter Gmutator approach vs. the simple Grammarinator+Mutations baseline in identifying discrepancies.

Differential testing across SUTs. Recall from §4.3 that we consider three SUTs per input format. This allows us to perform differential testing across SUTs, to identify cases where an input is accepted by some but not all of the SUTs, despite the fact the SUTs advertise that they consume the same input format. In practice, having performed differential testing between each SUT and the grammar-generated parser (as described above), we can mine these results to identify mismatches between SUTs.

This analysis adds colour to the findings for *RQ1*, as it is interesting to know whether mismatches are SUT-specific or common to multiple SUTs. It will also be useful in answering *RQ2*; for example, if all SUTs for an input format accept a particular input that does not conform to the original grammar, this may suggest that the original grammar is too strict, which we can then investigate. Alternatively, if a non-conforming input is accepted by one SUT but not the others, this will likely be a useful point to raise when reporting the issue to an SUT developer to get their feedback. Again, a comparison of discrepancies between SUTs identified by Gmutator vs. Grammarinator+Mutations will be of interest.

Manual investigation of discrepancies. The next step after performing differential testing is debugging. For every input format, we will first de-duplicate all instances of discrepancies between grammar and SUTs, or discrepancies between SUTs. This can be achieved by classifying inputs by their (mutant) grammar source. For each unique issue, we will investigate whether the fault is in the SUT or in the grammar, or if the issue is a false alarm, e.g. where an input is rejected by an SUT due to violating a semantic constraint. Our references will be the official specifications for JSON [6], Lua [15], URL [11], and XML [34]. When we are confident the discrepancy constitutes non-conformance to the specification, we will communicate the issue to the relevant SUT developers and report their answers.

This manual investigation will provide answers to *RQ2*.

Recording crashes and coverage. When running each SUT over the sets of inputs generated by GRAMMARINATOR, GMUTATOR and GRAMMARINATOR+MUTATIONS, we will record all crashes that occur. In addition, since we do have source code for each SUT, we will collect branch coverage information, using standard code coverage utilities for C/C++, JavaScript and Python.

The data we gather on crashes and coverage, for inputs generated using all three approaches, will allow us to answer *RQ3*. When we find inputs that lead to SUT crashes we will de-duplicate and report them. Cases where an input that crashes an SUT also turns out to be a non-conforming input (i.e. one that does not conform to the original grammar) will additionally contribute to answering *RQ2*.

Experimental settings. We will run experiments on a cluster of multicore Linux workstations (we will provide details of machine specifications when presenting our full results).

The SUTs will 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 GRAMMARINATOR+MUTATIONS), we will perform three 24 h runs. Each run will repeat 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, and (3) in the case where the generation tool is GMUTATOR or GRAMMARINATOR+MUTATIONS, attempting to parse the input using an ANTLR-generated parser for the original grammar (to record whether or not the input is valid). To account for inputs that trigger infinite loop bugs in our SUTs, or that lead to excessive SUT runtime, we will use a timeout of 3 seconds per input.

Performing three repeat runs allows us to present averaged coverage data, whilst keeping the CPU time required for our experiments tractable. Our planned experiments will 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.

Recall that 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, in the first 24 h run for a (generation tool, SUT) pair, the sequence of seeds $[0, 3, 6, 9, \ldots]$ will be used to initialise the generation tool. On the second and third repeat runs, the seed sequences $[1, 4, 7, 10, \ldots]$ and $[2, 5, 8, 11, \ldots]$ will be used, respectively. This means that, for example, the first repeat run in which GRAMMARINATOR is used to tests 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).

4.5 Evaluation Metrics

For clarity, we now recap the metrics that will be used to help in answering RQs 1–3, based on the data gathered from the generation and testing process described in §4.4.

For each benchmark, we plan to measure the following:

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. We will measure the number of accept-invalid inputs for each SUT, and categorise them, in order to answer *RQ1*.

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 this category is less important than the *accept-invalid* category above, because both GRAMMARINATOR and GMUTATOR have the potential to discover *reject-valid* inputs while only GMUTATOR has the potential to discover *accept-invalid* inputs. Still, reporting on *reject-valid* inputs is important to fully answer *RQ1*.

Cross-SUT disagreement. Recall that we consider three SUTs per input format (see Table 1). Counting and classifying the inputs for which there is disagreement between the three associated SUTs on whether the input should be accepted is important in the context of *RQ2*, because when reporting a discrepancy to developers it is informative to remark on cases where a comparable SUT does not exhibit the discrepancy.

Grammar-based faults. For each distinct category of *accept-invalid* or *reject-valid* inputs that we discover, we will investigate whether the issue is in fact due to a problem with the ANTLR grammar for the input format. This will inform *RQ2*.

SUT-based faults. If an *accept-invalid* or *reject-valid* input is not due to a problem with the grammar, we investigate if the error originates from the SUT. For every distinct class of issue,[3] based on feedback from developers we will classify the issue as one of: *intended* (the discrepancy is a feature of the SUT, e.g. as in the case for the discrepancy in Parson discussed in §2, where the parson developers cited Postel's law), *unintended* (the discrepancy is a fault in the SUT implementation, as appears to be the case with the issue discussed in §2 where cJSON accepts inputs that contain invalid UTF-8 characters), or *unresolved* (where despite discussion with developers, or due to lack of developer feedback, we do not manage to gain clarity on the issue). This classification of issues relates to answering *RQ2*.

Code coverage. For each input format, and for each set of associated inputs generated using Grammarinator, Gmutator and Grammarinator+Mutations, we will measure the total branch coverage achieved when each SUT for the input format is run across the input set. This will allow us to report, on average, how much Gmutator can improve on code coverage compared to Grammarinator and Grammarinator+Mutations, answering part of *RQ3*.

Program crashes. To assess the extent to which Gmutator can identify additional SUT crashes compared with Grammarinator and Grammarinator+Mutations, we will study logs of crashes identified when testing each SUT over the sets of inputs generated for each input format. Based on a best-effort de-duplication of crashes from details contained in their logs, we will report average results on the new crashes that Gmutator induces. It possible that Gmutator may *fail* to find certain crashes that Grammarinator exposes: even though Gmutator can generate any input that Grammarinator can generate in principle, it may do so with a lower probability in practice (due to the space of possible inputs that it can generate being larger). We will analyse whether such cases occur during our experiments. The data we collect on crashes will answer the remainder of *RQ3*.

5 RELATED WORK

Our approach builds upon grammar-based fuzzing [2, 14, 23, 25, 35, 37] and the Grammarinator [13] 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 [14, 37], 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 [20] 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 grammars from sample inputs [4, 5, 10, 17]. 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 builds on top of existing grammars, and attempts to generate inputs at the "edge" of what the input grammar allows.

A similar approach to ours is Ccoft [32], 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.

Unlike byte-level mutators, such as AFL [38], Gmutator performs mutations at the structure level. Random byte mutations typically break the syntax of an input, whereas we seek to produce inputs with similar structures.

6 CONCLUSION

We have presented Gmutator, a prototype tool that takes advantage of existing input grammars, and uses a combination of grammar mutation and grammar-based fuzzing as the basis for generating both well-formed and *almost*-well-formed inputs. Through a case study on JSON, we have presented preliminary evidence that the latter may be effective in identifying discrepancies between the inputs that systems under test actually accept vs. the inputs that they are supposed to accept according to the input format.

We have presented a plan for a larger experimental evaluation to assess the effectiveness of our idea on twelve different SUTs covering four different input formats. Through this evaluation we will measure and report on the rate of discrepancies Gmutator can discover, the improvement in code coverage and crash detection afforded by exploring a richer space of inputs, and the response from SUT developers to reports of inputs that trigger discrepancies.

In future work it would be interesting to extend our grammar mutation strategy to include new mutations, including experimenting with less conservative mutations. One direction would be the use of grammar fragments to derive new types of grammar mutations. Grammar fragments can be mined from the ANTLR repository, where over 200 grammars are available.

Another research direction is to exploit the fact that our monotonic mutation strategy (see §6 and Figure 2) enables the use of seed inputs, which have the potential to lead to more effective fuzzing.

7 DATA AVAILABILITY

The Gmutator implementation and experimental data are available at 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 grant EP/R006865/1.

---

[3] If a particular SUT turns out to suffer from a high rate of discrepancies we may report only a limited number of issues to avoid bombarding developers with reports.

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, 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

[6] Douglas Crockford. 2017. cjson. https://json.org.

[7] Eliott Dumeix. 2023. A Lua parser and AST builder written in Python. https://github.com/boolangery/py-lua-parser.

[8] Krzysztof Gabis. 2023. Lightweight JSON library written in C. https://github.com/kgabis/parson.

[9] Dave Gamble. 2023. Ultralightweight JSON parser in ANSI C. https://github.com/DaveGamble/cJSON.

[10] 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

[11] Network Working Group. 2005. Uniform Resource Identifier (URI): Generic Syntax. https://datatracker.ietf.org/doc/html/rfc3986.

[12] Amit Kumar Gupta. 2023. Fast XML Parser. https://github.com/NaturalIntelligence/fast-xml-parser.

[13] 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

[14] 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).

[15] Roberto Ierusalimschy, Waldemar Celes, and Luiz Henrique de Figueiredo. 2023. Lua. https://www.lua.org/manual/5.3/manual.html.

[16] Arseny Kapoulkine. 2022. Light-weight C++ XML processing library. https://pugixml.org.

[17] 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

[18] Daniel Lemire, Geoff Langdale, and John Keiser. 2023. Fast parser for large JSON files. https://simdjson.org.

[19] LLVM Project. Date Accessed July 21, 2022. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[20] 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).

[21] 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].

[22] Hrvoje Nikšić. 2023. Network utility to retrieve files from the World Wide Web. https://www.gnu.org/software/wget/.

[23] 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

[24] Mike Pall. 2022. Just-In-Time (JIT) compiler for the Lua programming language. http://luajit.org.

[25] 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

[26] 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).

[27] Nicolas Seriot. 2016. Parsing JSON is a minefield. https://seriot.ch/parsing_json.php [Online; accessed 15-Sep-2020].

[28] 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

[29] Daniel Stenberg. 2023. Command line tool and library for transferring data with URLs. https://curl.se.

[30] The GNOME Project. 2023. XML toolkit implemented in C. https://gitlab.gnome.org/GNOME/libxml2.

[31] Tatsuhiro Tsujikawa. 2021. Utility for downloading files. https://aria2.github.io.

[32] 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

[33] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. IFuzzer: An Evolutionary Interpreter Fuzzer Using Genetic Programming. In *Proc. of the (ESORICS'16)*. https://doi.org/10.1007/978-3-319-45744-4_29

[34] W3C. 2008. Extensible Markup Language (XML) 1.0 (Fifth Edition). https://www.w3.org/TR/xml/.

[35] 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

[36] Wikipedia. 2023. Robustness principle. https://en.wikipedia.org/wiki/Robustness_principle.

[37] 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

[38] Michal Zalewski. 2019. AFL Documentation. https://afl-1.readthedocs.io/_/downloads/en/latest/pdf/.

Received 2023-05-15; accepted 2023-06-12