Grammarinator Meets LibFuzzer: A Structure-Aware In-Process Approach

Renáta Hodován[a] and Ákos Kiss[b]
Department of Software Engineering, University of Szeged, Árpád tér 2, Szeged, Hungary

Keywords: Fuzzing, Grammar-Based, Coverage-Guided, In-Process, Grammarinator, LibFuzzer.

Abstract: Fuzzing involves generating a large number of inputs and running them through a target application to detect unusual behavior. Modern general-purpose guided fuzzers are effective at testing various programs, but their lack of structure awareness makes it difficult for them to induce unexpected behavior beyond the parser. Conversely, structure-aware fuzzers can generate well-formed inputs but are often unguided, preventing them from leveraging feedback mechanisms. In this paper, we introduce a guided structure-aware fuzzer that integrates Grammarinator, a structure-aware but unguided fuzzer, with LibFuzzer, a guided but structure-unaware fuzzer. Our approach enables effective testing of applications with minimal setup, requiring only an input format description in the form of a grammar. Our evaluation on a JavaScript engine demonstrates that the proposed fuzzer achieves higher code coverage and discovers more unique bugs compared to its two predecessors.

1 INTRODUCTION

In our increasingly fast-paced world, software supports every aspect of our lives, from our cars to our TVs, medical instruments, and even smart toothbrushes. This means writing a lot of new code and even more tests. At least, that would be the case in an ideal world. However, keeping up with this volume of code with high-quality test suites is very challenging. It is common practice to write tests that check the expected behavior of programs, but tests for handling unexpected behavior are much rarer. Yet, or precisely because of this, improper handling of unexpected inputs can be a perfect attack vector for a malicious user. Writing negative tests, however, is not an easy task, as it is much simpler to list the cases where the program should work than to enumerate all the faulty possibilities it should handle correctly. Therefore, it is not surprising that an automatic approach to negative testing, known as random testing or fuzzing, has emerged (Miller et al., 1990; Miller, 2008).

The essence of fuzzing is to generate a large number of inputs that are run through the tested application while monitoring its execution. If something unusual is detected, it is saved as a potential bug to be validated by a human tester. Unusual behavior could be a crash, an unusually long runtime, or an output deemed incorrect (if an oracle is available during monitoring). Nowadays, there are many types of fuzzers available, but among them, general-purpose guided fuzzers are particularly popular (e.g., AFL (Zalewski, nd) and its variants (Böhme et al., 2016; Lemieux and Sen, 2018; Lemieux et al., 2018; Atlidakis et al., 2020; Pham et al., 2021; Fioraldi et al., 2020), or LibFuzzer (LLVM Project, nd)) because a single well-written fuzzer can test hundreds of programs effectively, finding many bugs. These fuzzers are based on simplicity and speed: starting from a seed corpus (in the extreme case, perhaps even from an empty one), they generate new tests by making minimal byte-level changes to the elements of the corpus, while prioritizing those elements that have already proven useful based on certain metrics. While these fuzzers do find many bugs, most of those bugs are parser-related because such simple mutators often corrupt the structure of the input, rarely driving deeper code sections beyond the parser.

Targeted fuzzers offer a solution to this problem by providing automated testing for specific input formats (e.g., CSmith (Yang et al., 2011), GrayC (Even-Mendoza et al., 2023), Domato (Fratric, nd), Echidna (Grieco et al., 2020), jsfunfuzz (Mozilla Corporation, nd), or IFuzzer (Veggalam et al., 2016)). These are effective for one given format, but not for other targets. To mitigate this prob-

[a] https://orcid.org/0000-0002-5072-4774
[b] https://orcid.org/0000-0003-3077-7075

Hodován, R. and Kiss, Á.
Grammarinator Meets LibFuzzer: A Structure-Aware In-Process Approach.
DOI: 10.5220/0013571500003964
In *Proceedings of the 20th International Conference on Software Technologies (ICSOFT 2025)*, pages 178-189
ISBN: 978-989-758-757-3; ISSN: 2184-2833

lem, there are fuzzers that, with the help of an input format description—which can be a grammar, an IDL definition, an XML schema, or a completely custom format—can generate structure-aware tests through generation or mutation (e.g., Grammarinator (Hodován et al., 2018), LangFuzz (Holler et al., 2012), Dharma (Diehl, 2015), Peach (GitLab B.V., nd), or Skyfire (Wang et al., 2017)), making them usable for multiple input formats with relatively little effort. However, most of these are not guided, thus missing out on very useful feedback information.

The solution may lie in guided, structure-aware fuzzers, which enable a single fuzzer to generate or mutate tests for any application in a guided manner, requiring only a format description. Examples of such fuzzers include Nautilus (Aschermann et al., 2019), Superion (Wang et al., 2019), and libprotobuf-mutator (Google Inc., nd); and the tool presented in this work.

In this paper, we present a new tool resulting from the integration of Grammarinator and LibFuzzer, which combines their strengths while mitigating their weaknesses. During the design of the integration, we ensured that existing functionalities were preserved, and we also introduced new features inspired by the two components. The new tool can work with and mutate a seed corpus like its predecessors, but it can also generate completely new tests or parts thereof based on a grammar. Thanks to the popularity of ANTLR4, hundreds of grammars are available online (Parr, 2013), removing or reducing the effort needed for writing grammars. The new tool can apply structure-aware mutations to the input but also has access to the tried and tested mutators of LibFuzzer. Moreover, since Grammarinator offers numerous customization points, those can be utilized to define logic that is beyond the expressive power of grammars.

The rest of this paper is organized as follows: In Section 2, we give an overview of the two tools our work is based on, i.e., Grammarinator and LibFuzzer. In Section 3, we list the challenges we have faced during their integration and present our solution. In Section 4, we present the scope and results of the evaluation of the new tool. In Section 5, we overview the related works from the literature, and finally, in Section 6, we conclude our paper.

2 BACKGROUND

2.1 LibFuzzer

LibFuzzer (LLVM Project, nd) is an efficient coverage-guided in-process fuzzing engine integrated with LLVM. It is a general-purpose fuzzer, which is applicable to any input format. It is based on a genetic algorithm that, in each iteration, selects a test from its corpus, mutates it, and then runs the resulting test with the fuzz target. Thanks to its integration with the compiler, the fuzz target is instrumented during compilation so it can monitor the parts of the code covered by the test during execution. If this indicates the discovery of new code sections, the test is deemed useful and added to the corpus for further testing. Although fuzzing can start with an empty corpus, the quality of the corpus significantly affects the efficiency of the fuzzing process. A high-quality initial corpus is necessary because LibFuzzer generates new tests using very simple mutations, making it challenging to create tests that match a complex input format starting from an empty corpus. However, these simple mutations make LibFuzzer applicable to any input format and allow for the rapid generation of new tests, which is a critical requirement for effective fuzzing.

Currently, LibFuzzer has 13 built-in mutators and 2 extension points that allow for user-defined custom mutators, which are summarized in Table 1. With one exception, all mutators operate on a single corpus element, making changes to it. These mutations are typically applied at randomly chosen positions in the input: they delete, insert, repeat, or rearrange bytes. They move randomly selected chunks of the test case to different locations. They insert strings from a manually specified or automatically collected dictionary. Or, they alter numeric values. The only mutation that works with two corpus elements simultaneously is the cross-over mutation, which randomly selects one or more chunks from the donor element and inserts them into random positions in the host element or replaces random parts of it.

As can be seen, these mutations can generate new tests very quickly because—with minimal exceptions—they do not attempt to interpret the input; they just work at random positions. The sole exception is the mutation of numbers, where the input is scanned once, and positions containing digits are recorded.

This simplicity, generality, and speed are both the strengths and weaknesses of LibFuzzer. It can test applications with any input format out-of-the-box, but since most of the generated tests are syntactically incorrect, they primarily drive and test only the parsers of the applications being tested. In the case of more complex programs, however, this only scratches the surface of the codebase, leading to a demand for structure-aware mutators years ago.

Table 1: LibFuzzer Mutators and Their Descriptions.

Mutator	Description
Erase Bytes	Delete bytes at randomly chosen positions from the input.
Insert Byte	Insert a single byte at a randomly chosen position in the input.
Insert Repeated Bytes	Insert a sequence of repeated bytes at a randomly chosen position in the input.
Change Byte	Change a byte at a randomly chosen position in the input.
Change Bit	Change a bit at a randomly chosen position in the input.
Shuffle Bytes	Shuffle bytes in a randomly chosen segment of the input.
Change ASCII Int	Change an ASCII integer at a randomly chosen position in the input.
Change Bin Int	Change a binary integer at a randomly chosen position in the input.
Copy Part	Copy a randomly chosen chunk of the input to another position within the same input.
Cross-Over	Copy a randomly chosen chunk from the donor element and insert it into a random position or replace a randomly chosen chunk in the host element.
Manual Dict	Insert strings from a manually specified dictionary.
Persistent Auto Dict	Insert strings from an automatically collected dictionary.
CMP	Insert strings from recent compares.
Custom Mutator	Apply custom user-defined mutations on the input.
Custom Cross-Over	Apply custom user-defined cross-over mutations on the input.

2.2 Grammarinator

Grammarinator (Hodován et al., 2018) is a structure-aware test generation tool that produces tests in a black-box manner without any feedback, using ANTLR4-format grammar descriptions. This format was designed for the popular ANTLR parser generator tool (Parr, 2013), which has developed a significant ecosystem over the years. The grammar format essentially describes extended context-free grammars but includes many additional features, such as actions embedded in the grammar, predicates, special wildcards, etc. Due to its popularity, there are currently more than 300 publicly available grammars in the official grammar repository[1].

To avoid the labor-intensive task of writing grammars for a proprietary format, Grammarinator chose the ANTLR4 grammar format as the model for its test generator. In the first step, it processes the chosen grammar and then creates a Python test generation code from it. This generator can later be used to produce any number of tests. The generated code traverses the grammar and through a series of random decisions, selects between alternatives and decides how many times to repeat quantified subexpressions. The result of the generation is a tree much like parse trees, where each internal node represents a grammar rule and each leaf node represents a token. This resulting tree is then processed by a serializer, which combines the leaf nodes—while possibly applying further formatting, at least inserting spaces between the leaves—to produce the final test.

In addition to generating grammar-conforming tests, Grammarinator can create mutants from existing tests. It supports two types of mutation operators: it can either regenerate a randomly chosen subtree according to the grammar rule corresponding to the subtree's root, or it can swap nodes of the same type (i.e., belonging to the same grammar rule) between two tests. Mutations are typically performed on elements of a tree corpus (called a population in Grammarinator's terminology). Such populations can either be generated by Grammarinator itself or created from existing tests using a test-to-tree conversion utility that is part of the Grammarinator toolchain (utilizing ANTLR4, parsing the tests with the same grammar that is used for generation, and converting the parse trees to the tree representation of Grammarinator). It is important to note that the selection of trees to mutate is entirely random, without any guidance. The reason for this is that Grammarinator is purely a test generator; it does not have an integrated fuzzing harness, so it cannot run tests or collect feedback on its own.

Beyond the core functionality described above, Grammarinator supports several customization features, such as subclassing generator classes, defining model classes to control alternative selection and quantification, defining listeners, inserting inline code and predicates into the grammar, post-processing the generated trees with transformers, supporting custom serializers, and more. These features are discussed in detail in the original paper and the online documentation of the project (Hodován et al., 2018).

[1] https://github.com/antlr/grammars-v4

3 INTEGRATING GRAMMARINATOR AND LIBFUZZER

3.1 Challenges and Solutions

Recognizing the strengths and weaknesses of Grammarinator (good for structured inputs, but lacks guidance) and LibFuzzer (good at guidance, but struggles with heavily structured input formats), the need arose to combine the two tools to create a more efficient fuzzer that keeps the best of both. In the following, we will cover the necessary steps for integration, the challenges encountered, and the resulting developments[2].

Technically, the integration is made possible by the two extension points of LibFuzzer: the custom mutator and cross-over operations. If these two are defined, then they override the built-in mutators of LibFuzzer. Thus, we had a way to let LibFuzzer manage the corpus, guide the fuzzing, and channel the selected test cases to Grammarinator for structure-aware mutation.

The first challenge to overcome was that Grammarinator generated fuzzers in Python, while LibFuzzer primarily expects the custom operators to be implemented in compiled languages, e.g., C/C++. (Although we are aware that LibFuzzer is not limited to testing applications written in C or C++, and can be applied to targets in languages such as Python, Java, Rust, or Go through various wrappers, C and C++ are the most common. Therefore, we focus on these as the target languages.) As executing the fuzzer in a Python interpreter within the custom operators did not seem to be optimal performance-wise, we extended Grammarinator to generate C++ fuzzers. This not only involved generating code from the grammar in the statically typed C++ instead of the dynamically typed Python language but also porting the runtime library necessary for the execution of the generated fuzzers.

With the C++ port in place, we could call Grammarinator's subtree-regenerating or -replacing functionalities from the custom mutator and cross-over implementations. However, Grammarintor works with trees, while LibFuzzer, being of general purpose, treats all test cases as simple arrays of bytes. In a naïve integration, this would require the test-to-tree conversion (i.e., parsing) of inputs every time the generation of a new test case is requested. (And, once Grammarinator mutated the tree, the serialization of the tree to the original format will also be necessary: both because the fuzz target expects it that way, and because LibFuzzer needs to store the test cases it deems interesting.) Unfortunately, continuously parsing the test cases over and over again would slow down fuzzing unacceptably, making such a naïve approach useless in practice.

As an answer to this second challenge, we have moved parsing from fuzzing-time to preprocessing-time. I.e., we required the test corpus to be in tree form by the time fuzzing starts. Fortunately, the tree-to-test conversion utility of the Grammarinator toolchain covered this requirement. However, this change moves the task of tree serialization from the mutator to the fuzz target. This means that there is a small development/maintenance price to pay for better performance as the fuzz target needs to be extended with the tree-to-test conversion logic. However, in the best case, this can be as simple as adding an extra compile flag to the fuzz target's compilation command to inject (include) the necessary code.

It must be noted that the trees that Grammarinator works with are data structures in memory, while the trees stored in files in the corpus and handled by LibFuzzer are some encoded forms of those data structures. Thus, there is a need for some decoding and encoding of trees when working with a tree corpus, but—if an appropriate codec is used—its overhead is by orders of magnitude smaller than that of complete parsing. The Python runtime library of Grammarinator already had a pickle-based tree codec, but that approach was not portable to C++. Therefore, we have implemented a FlatBuffers[3]-based codec for C++ (and also for Python, which had the fortunate side effect that the same tree corpus could be used by both the Python and C++-based Grammarinator).

Although tree decoding is much faster than parsing, the construction of the data structures in memory still costs time and should be avoided if possible, which posed our third challenge. To address this, we have introduced a single-element cache where the encoded tree representation is the key and its in-memory data structure is the associated value. Whenever a mutation or a cross-over generates a new test case—i.e., a tree—its encoded form along with the tree itself is put into the cache. Then, if LibFuzzer passes the encoded tree to the fuzz target for testing, the corresponding tree is retrieved from the cache. Moreover, since LibFuzzer typically applies multiple mutations to the same test case in a sequence, querying the cache as the first step of mutation is also beneficial.

An overview of the above-described approach is shown in Figure 1. The diagram displays some com-

[2]All source code is available at https://github.com/renatahodovan/grammarinator.

[3]https://github.com/google/flatbuffers

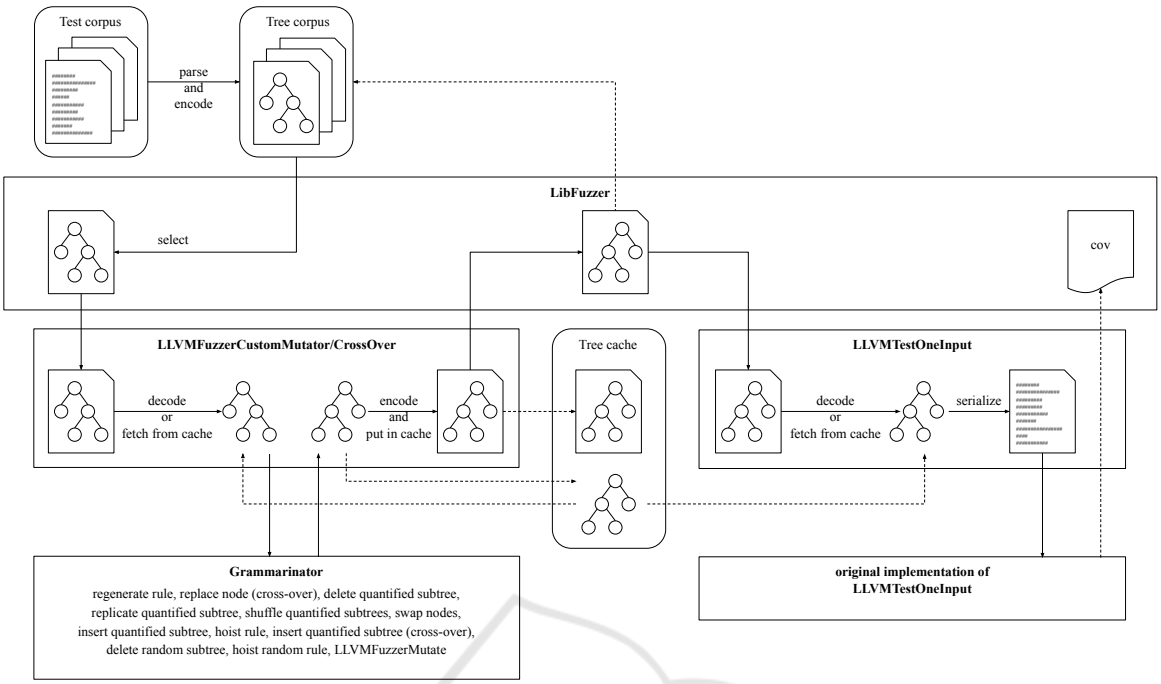

Figure 1: An overview of the integration of Grammarinator into LibFuzzer.

ponents using the API function names as defined by LibFuzzer. I.e., LLVMFuzzerCustomMutator and LLVMFuzzerCustomCrossOver are the functions that implement the custom mutator and cross-over extension points, respectively, while LLVMFuzzerTestOneInput is the mandatory fuzz target that calls the code being tested.

3.2 Structure-aware (and Structure-unaware) Mutators

The main motivation for the integration of Grammarinator and LibFuzzer was to create a technique that has the best features of both tools. We have realized that although the built-in mutators of LibFuzzer cannot comply with input format restrictions, the ideas behind them can be useful for structure-aware mutations as well. It turned out that the two existing mutation operators of Grammarinator (i.e., the regeneration of a subtree and the swapping of two compatible subtrees between trees) covered only a subset of these ideas. For example, while LibFuzzer's Erase Bytes mutator deletes a randomly selected byte sequence from a random position of the input, there could be a structure-aware mutator that deletes subtrees—but only if the grammar defined them as optional. Similarly, while the Insert Byte and Copy Part mutators insert and replace randomly selected input segments at random positions, respectively, their structure-aware counterparts could perform those operations on subtrees—again, according to the rules of the grammar. Therefore, we have extended Grammarinator with several new mutators inspired by the operators of LibFuzzer.

To know which subtree can be deleted or where can a further subtree be inserted, it is necessary to have information about which parts of the tree originate from quantified expressions (?, *, or +) of the grammar and what the lower and upper bounds of the quantifiers are. Similarly, to change a decision made at an alternation (|), it is necessary to have information about which parts of the tree originate from which alternative of which alternation. When generating a completely new test case with Grammarinator, this information can be added to the tree relatively easily. However, when building the trees from existing test cases using ANTLR4, the parse trees lack this information. Thus, in addition to extending the internal representation of Grammarinator to store and improving the fuzzer to generate this information, we have also enhanced the test-to-tree conversion utility to recreate this information when parsing is performed. This way, all structure-aware mutators could be applied both to trees newly generated and to those converted from existing tests.

The implemented structure-aware mutators are listed in Table 2 along with the LibFuzzer mutators

Table 2: Structure-aware Grammarinator Mutators and Their LibFuzzer Counterparts.

Grammarinator Mutator	Description	LibFuzzer Counterpart(s)
Regenerate Rule	Regenerate a subtree based on the grammar rule associated with the root of the subtree.	*Change ASCII Int, Change Bin Int, Manual Dict*
Replace Node (Cross-Over)	Replace a subtree with another subtree that has a root of the same type and is from a different tree.	*Cross-Over*
Delete Quantified Subtree	Delete a subtree that corresponds to a quantified expression (?, *, or +) in the grammar.	*Erase Bytes*
Replicate Quantified Subtree	Duplicate a subtree that corresponds to a quantified expression (* or +) in the grammar.	*Insert Repeated Bytes*
Shuffle Quantified Subtrees	Randomly shuffle the subtrees that correspond to a quantified expression (* or +) in the grammar.	*Shuffle Bytes*
Swap Nodes	Swap two subtrees that have roots of the same type.	*no direct counterpart*
Insert Quantified Subtree	Insert a copy of a subtree that corresponds to a quantified expression (?, *, or +) between other subtrees that correspond to the same quantified expression.	*Insert Byte, Copy Part*
Hoist Rule	Lift a subtree up in the tree hierarchy to replace an ancestor that corresponds to the same grammar rule.	*no direct counterpart*
Insert Quantified Subtree (Cross-Over)	Insert a subtree that corresponds to a quantified expression (?, *, or +) between other subtrees of another tree that correspond to the same quantified expression.	*Cross-Over*

Table 3: Structure-unaware Grammarinator Mutators and Their Structure-aware Counterparts.

Structure-unaware Mutator	Description	Structure-aware Counterpart
Delete Random Subtree	Delete a random subtree.	*Delete Quantified Subtree*
Hoist Random Rule	Lift a subtree up in the tree hierarchy to replace any of its ancestors.	*Hoist Rule*
LLVMFuzzer Mutate	Call the original mutator implemention of LibFuzzer on a token node.	*no direct counterpart*

that are most similar to their logic. The first two rows in the table are the original two mutators of Grammarinator, while the rest are those inspired by LibFuzzer.

While we expect the new structure-aware mutators to reach beyond the parser logic of the tested code more easily, we do not want to lose the ability to test the parser either. Therefore, we have created additional tree-based but structure-unaware mutators. These operators are similar to some structure-aware mutators with the important difference that when selecting which part of the tree to apply them on, fewer or no constraints are adhered to. (Moreover, since the built-in mutators of LibFuzzer are also accessible, one of the structure-unaware mutators is to invoke them on leaf nodes, i.e., tokens.) The list of implemented structure-unaware mutators is given in Table 3.

4 EVALUATION

4.1 Setup

In our evaluation, we wanted to compare the integration of Grammarinator and LibFuzzer to its two main building blocks, i.e., to the unguided Grammarinator and the structure-unaware LibFuzzer. For LibFuzzer, we used a custom build of LLVM (and clang) version c4a00be08aa1, extended with a PC dump functionality at exit. This extension enabled the comparison of the coverage metrics reported by LibFuzzer across different fuzzers. For Grammarinator, our work was based on version 23.7.post140 (git revision 68b0350). To get comparable statistics, we have invoked even the unguided Grammarinator from the LibFuzzer framework by implementing the custom mutator extension point of LibFuzzer in such a way that no matter what input was selected by LibFuzzer for mutation, it was discarded and Grammarinator always generated a completely new test case. (The initial corpus was still loaded, though, to ensure

that all fuzzing sessions start with the same baseline statistics.)

In the evaluation, we used two common statistics as the basis of the comparison: coverage and feature count metrics as reported by LibFuzzer. In this context, coverage refers to edge coverage, while feature count is a combined metric computed by LibFuzzer, which includes edge counters, value profiles, indirect caller/callee pairs, and more (LLVM Project, nd). In addition, we disclose the number and type of bugs found. Finally, we also report the speed of fuzzing, more precisely, the number of inputs generated during a fuzzing session.

For the evaluation, we looked for a JavaScript engine as our target because the language makes for a complex and structured-enough input format, and a full execution engine ensures that there are deeper parts for the fuzzer to reach than the parser logic. We expect such targets to be best fit for a Grammarinator-LibFuzzer integration. (For non-structured input formats or targets consisting of a parser only, LibFuzzer's simple and fast mutators are likely to be more effective. Moreover, the time invested in sophisticated computationally intensive tree mutations is more likely to pay off on a computationally intensive target.) Thus, we have selected JerryScript (JS Foundation et al., nd) as the target (at v3.0.0 release, git revision 50200152), a JavaScript engine written in C of cca. 166k LOC. The project already includes a LibFuzzer target (i.e., LLVMFuzzerTestOneInput implementation for JavaScript execution) and has been fuzz-tested for several years.

Since Grammarinator needs a grammar for fuzzing, we have downloaded the JavaScript grammar available from the ANTLRv4 grammar repository[4]. To enhance its usefulness for fuzzing, we have made three small modifications to the grammar. First, since every test case should yield consistent results even if the corpus changes, there should be no dependencies between the elements of the corpus. The import statements of the JavaScript language go against this directive, therefore we have disabled their generation. Second, we have also disabled the generation of hashtag lines as they are but unstructured comment lines from the perspective of the grammar (and useless for the target, too). Third, we have also extended the grammar, namely by explicitly listing the names of built-in classes and functions supported by JerryScript as alternatives to the identifier tokens. (To ensure a fair comparison, we also created a so-called dictionary of these identifiers and the literals of the lexer for LibFuzzer to use during fuzzing (LLVM Project, nd).)

[4]https://github.com/antlr/grammars-v4/tree/master/javascript/javascript

A grammar is mandatory for Grammarinator to work, but it can take optional parameters as well to influence its behavior (Hodován et al., 2018). The JavaScript grammar is highly recursive, therefore we have limited the depth of the trees to 25 to avoid generating overly deep trees or running into infinite recursion. We have also limited the maximum number of generated tokens to 500 to restrict not only the depth but also the width of the generated trees.

When using Grammarinator-based approaches, we observed that some generated JavaScript inputs contained infinite loops. Since these constructs are syntactically and semantically valid, their execution correctly continues indefinitely. While this behavior is expected, it causes the fuzzing process to stall whenever such inputs are executed. In contrast, LibFuzzer's native byte-level mutations rarely resulted in infinite loops, as a mutation would need to modify loop-related bytes in a way that remains syntactically valid while also causing unbounded execution—an unlikely combination. To address this, we run LibFuzzer with the fork mode enabled (-fork=1) and limit the test execution time to one second (-timeout=1). Moreover, we also instruct LibFuzzer to not halt fuzzing if any issue is found (-ignore_timeouts=1 -ignore_crashes=1 -ignore_ooms=1). This configuration ensures that the fuzzing target runs in a separate child process, allowing it to crash, timeout, or exhaust memory without affecting the main LibFuzzer process. By preventing repeated restarts and redundant corpus reloading, fork mode enables a more stable and efficient fuzzing session.

4.2 Results

In the evaluation, we ran all three fuzzers (the unguided Grammarinator, the structure-unaware LibFuzzer, and the Grammarinator-LibFuzzer integration) for one day, five runs each, all using fork mode 1. The fork mode 1 simulates a single process environment since the tests run in a single child process without parallelization, but it helps avoid the problem with timeout-inducing inputs discussed in the previous subsection. We used the same initial corpus for all three fuzzers, the only difference being the format of the elements in the corpus: the Grammarinator-LibFuzzer integration used a corpus of trees converted from the original tests in a preprocessing step as discussed in Section 3. During fuzzing, we recorded the change in the coverage and feature count metrics for each run of each fuzzer and plotted them, as shown in Figure 2. As visible from the charts, the unguided Grammarinator performed the worst and showed close to no improvement in the second half

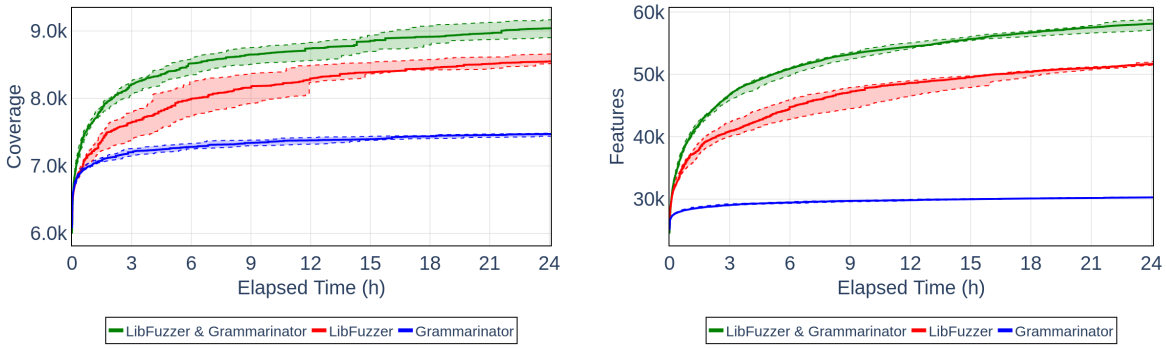

Figure 2: Change of coverage and features over time while fuzzing JerryScript for 24 hours. (Dashed lines show minimum and maximum values, while solid lines show the median of five runs.)

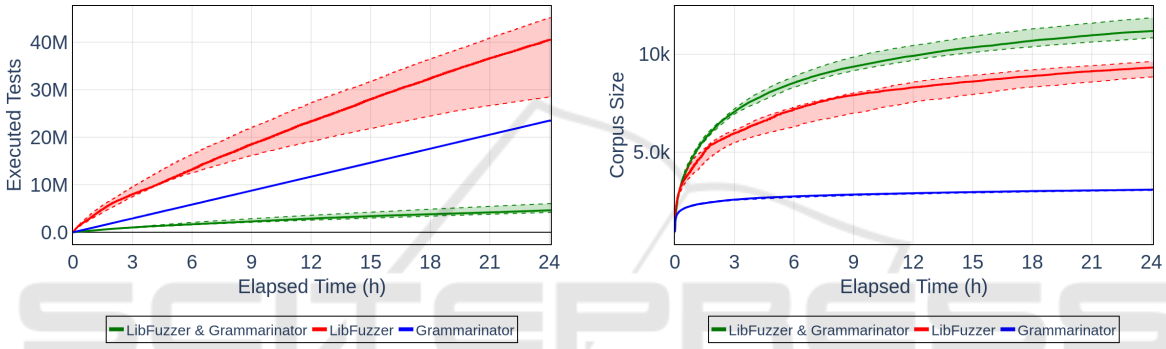

Figure 3: Number of executed tests and change of corpus size over time while fuzzing JerryScript for 24 hours. (Dashed lines show minimum and maximum values, while solid lines show the median of five runs.)

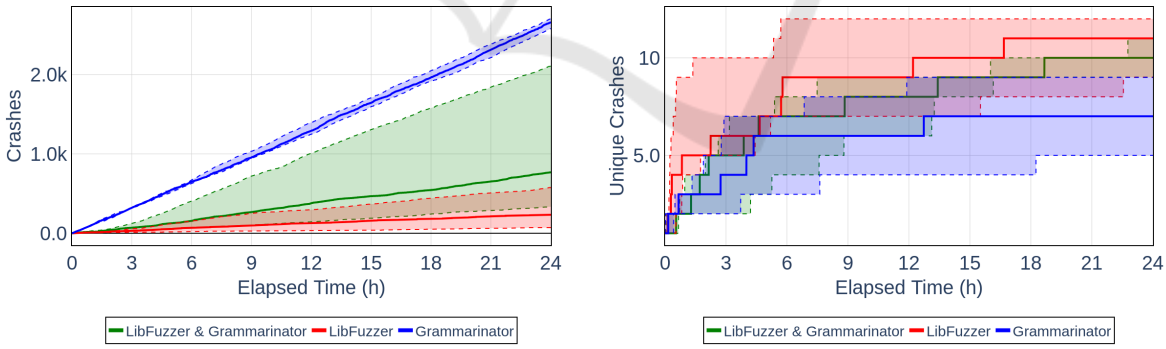

Figure 4: Number of all triggered and unique crashes over time while fuzzing JerryScript for 24 hours. (Dashed lines show minimum and maximum values, while solid lines show the median of five runs.)

of the evaluation, failing to discover new edges or features. In contrast, both LibFuzzer and Grammarinator guided by LibFuzzer continued to show steady increases in both coverage and feature count, with the guided Grammarinator achieving a faster increase than the structure-unaware LibFuzzer.

Figure 3 shows that LibFuzzer tested the most inputs in 24 hours, up to 11 times more than the guided Grammarinator and almost 2 times more than the unguided Grammarinator. The speed of LibFuzzer is not surprising, as it does not interpret the input byte array, and its mutations are very simple and fast operations. Additionally, it is further accelerated by the fact that most of the test cases it generates are quickly

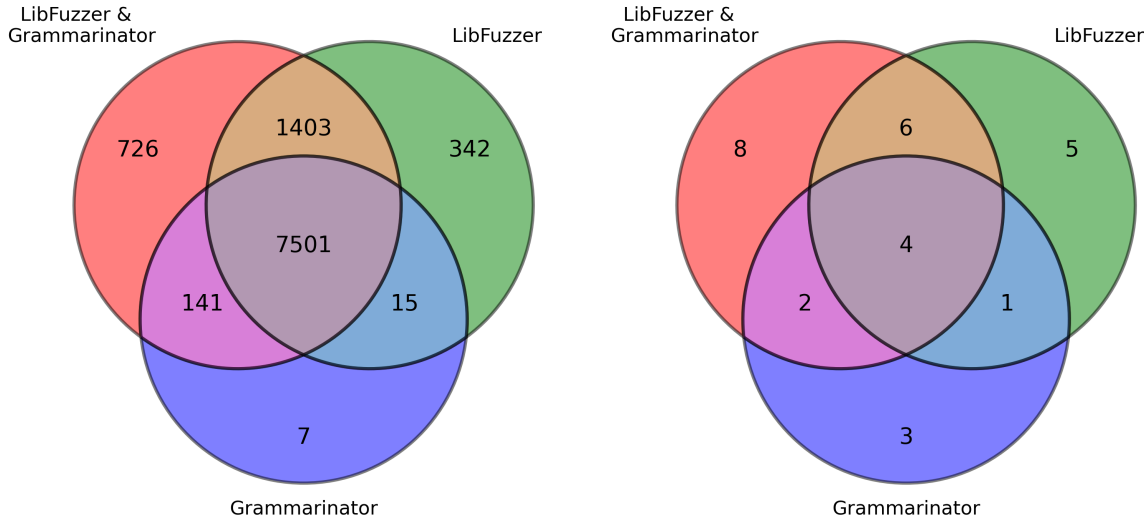

Figure 5: Number of covered edges and unique crashes after fuzzing JerryScript for 24 hours. (Results combined from five runs.).

Table 4: Statistics of Fuzzing JerryScript for 24 Hours (Minimum and Maximum Values of Five Runs).

Metric	**Grammarinator**	**LibFuzzer**	**LibFuzzer & Grammarinator**
Input Corpus	1,142	1,142	1,142
Output Corpus	3,269–3,312	9,067–9,853	11,065–12,085
Executed Tests	23.4M–23.5M	28.5M–45.1M	4.1M–6.0M
Crashes	2,578–2,703	78–588	336–2,108
Unique Crashes	5–9	9–12	9–11
all runs combined	*10*	*16*	*20*
Timeouts	1,680–1,781	44–179	5,805–9,574
Coverage	7,441–7,484	8,516–8,658	8,901–9,166
Features	30,250–30,343	51,486–52,083	57,102–58,763

discarded due to parser syntax errors, meaning that actual JavaScript execution occurs only rarely. The slowness of the Grammarinator-based fuzzers, especially that of the guided Grammarinator, might seem surprising but is explained by both the more sophisticated mutation strategies and the higher number of timeout-inducing inputs. The Grammarinator-LibFuzzer integration generated 32–217 times more test cases containing infinite loops than LibFuzzer, causing significantly more executions to be terminated due to timeouts. While non-timeout test cases can be executed at a rate of hundreds or thousands per second, a single timeout consumed an entire second, significantly reducing the overall fuzzing throughput. On the other hand, Figure 3 also shows that, although slower, the guided Grammarinator generated the most useful or interesting tests, both in absolute and relative terms, since the size of the corpus (containing the generated tests considered worthwhile keeping) surpasses the other two fuzzers during the entire evaluation period.

When evaluating fuzzers, the number of found bugs is an important aspect. Figure 4 shows the number of crashes triggered during the 24-hour measurement period. The figure shows that the unguided Grammarinator crashed JerryScript more than 2,500 times, but it is also visible that a large number of the crashes were duplicates. The guided fuzzers, although triggering crashes less often, found more unique bugs.

Figure 5 presents the results from a slightly different perspective. We have combined and visualized both the edges covered and unique bugs found during all five runs for each fuzzer. The diagrams show that all fuzzers could cover edges and find bugs that none of the others could, but it was the Grammarinator-LibFuzzer integration that found the highest number of edges and unique bugs exclusively. It was also the Grammarinator-LibFuzzer integration that covered the most edges and found the most unique bugs in total during the five runs.

In Table 4, we summarize the results shown in the

aforementioned figures. In the most important metrics, Grammarinator guided by LibFuzzer performed very well: it generated the most of the useful test cases, achieved the highest coverage, and found most of the unique bugs (when combining all five runs of the evaluation).

5 RELATED WORK

The field of fuzzing has undergone tremendous development in its over 30-year history, and this pace of progress is accelerating. Its literature became huge. While we mentioned some of the relevant papers in Section 1, due to page limits, we will now focus specifically on the three approaches most closely related to our work: Superion (Wang et al., 2019), Nautilus (Aschermann et al., 2019) and libprotobuf-mutator (Google Inc., nd). All approaches select elements from a corpus, run them through an instrumented target to collect coverage feedback, and then either add the test to the corpus or discard it based on the result. However, upon closer inspection, despite the similarities, they differ in several key points. In the following, these differences will be enumerated.

Superion is a self-contained, ANTLR grammar-based coverage-guided fuzzer framework built on AFL, which replaces AFL's mutators with its own solutions. It defines three new mutators: two of them are grammar-based, and one operates on byte sequences. The byte sequence-based mutator uses a simple heuristic to find token boundaries in the test, then creates new tests by replacing these with elements from the dictionary. The grammar-based mutators start by parsing, then randomly delete or swap parts of the inputs that belong to contiguos parts of the parse trees. These swaps and deletions, however, occur at random tree nodes without checking whether the grammar allows these operations. Grammar-based input generation is also not an option.

Nautilus is also a self-contained grammar-based coverage-guided fuzzer framework that is based on AFL and uses its own grammar format. Although the original publication had a transformer to convert any ANTLR grammar into the expected format, this support was discontinued shortly thereafter[5]. This forces users to manually write the necessary grammars, in contrast to Grammarinator, which utilizes the ANTLR4 grammar format and thus has out-of-the-box access to hundreds of grammars. Because of its own grammar format, Nautilus cannot build a seed corpus from existing tests, relying solely on tests generated by itself, which loses valuable inputs. In contrast, our approach can both start with an empty corpus and convert existing tests to a seed tree corpus. Regarding mutators, there is some overlap between Nautilus and our tool. Both can regenerate subtrees according to the grammar, swap subtrees of the same type between corpus elements, and include a transformation similar to hosting. Both approaches also call back to the base system's structure-unaware mutations: Nautilus applies AFL mutations to serialized subtrees, while Grammarinator applies LibFuzzer mutations to leaf nodes. But, in addition to the common mutations, Nautilus has three unique mutators, while Grammarinator has nine more.

The third related solution is the structure-aware fuzzing approach based on libprotobuf-mutator (Google Inc., nd). Libprotobuf-mutator was originally developed for targets that expect input in the Protocol Buffer format. Protocol Buffers is a language- and platform-neutral mechanism for serializing structured data, primarily used as a communication protocol. It has an interface definition language (IDL) that defines the expected input structure, which the target application can use for encoding or decoding data. Libprotobuf-mutator performs mutations on structured data described by such interfaces and also has integration with LibFuzzer. The main similarity with our work is its ability to fuzz even such structured data formats that do not directly use Protocol Buffers. This can be achieved by defining the logical structure of the input format using Protocol Buffers' IDL. Based on this, libprotobuf-mutator can perform the generation or mutation of inputs in Protocol Buffer format, and then a hand-written converter can convert those inputs to the target format. While both Grammarinator and libprotobuf-mutator work with their respective internal representations encoded in trees and require a decoding step (the serializer for Grammarinator, the converter for libprotobuf-mutator) before testing the input, the main difference is that Grammarinator works with a widely used grammar format that has descriptions for numerous input formats, but libprotobuf-mutator often requires the writing of an IDL. Additionally, writing the converter needs further manual work. Furthermore, libprotobuf-mutator has the same drawback as Nautilus, i.e., it cannot parse existing tests to build a seed corpus.

6 CONCLUSIONS

In this paper, we introduced the integration of the originally unguided Grammarinator with the

[5]https://github.com/nautilus-fuzz/nautilus/issues/1

structure-unaware LibFuzzer, resulting in a guided, structure-aware in-process fuzzer. We outlined the key challenges encountered during this integration and detailed the most effective solutions we developed. Beyond the integration, we proposed and implemented a set of structure-aware and structure-unaware mutators to enhance the new fuzzer's capabilities.

To evaluate the effectiveness of our approach, we conducted experiments using the JerryScript JavaScript engine as the target. Our results show that the Grammarinator-LibFuzzer integration achieved higher coverage and feature count than either of its predecessors. During the experiments, 29 bugs were discovered in total, with 20 detected by the Grammarinator-LibFuzzer integration and 8 exclusively found by this new fuzzer.

Encouraged by these promising results, we plan to continue this line of research by seeking answers to several open questions: How does input format complexity impact fuzzing efficiency? Which mutators contribute most to new coverage and feature discoveries, and how does their effectiveness change over time? How does the guided Grammarinator compare to other guided structure-aware fuzzers in terms of performance and capabilities? How would integrating Grammarinator with other guided fuzzing frameworks impact its efficiency? To explore these questions, we aim to conduct a larger-scale evaluation across a wider range of input formats, fuzz targets, baseline fuzzers, and guided fuzzing harnesses.

ACKNOWLEDGEMENTS

This research was supported by project no. TKP2021-NVA-09. Project no. TKP2021-NVA-09 has been implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

REFERENCES

Aschermann, C., Frassetto, T., Holz, T., Jauernig, P., Sadeghi, A.-R., and Teuchert, D. (2019). Nautilus: Fishing for deep bugs with grammars. In *Proceedings of the Network and Distributed Systems Security (NDSS) Symposium 2019*. Internet Society. doi:10.14722/ndss.2019.23412.

Atlidakis, V., Geambasu, R., Godefroid, P., Polishchuk, M., and Ray, B. (2020). Pythia: Grammar-based fuzzing of REST APIs with coverage-guided feedback and learning-based mutations. *arXiv preprint*. arXiv:2005.11498 doi:10.48550/arXiv.2005.11498.

Böhme, M., Pham, V.-T., and Roychoudhury, A. (2016). Coverage-based greybox fuzzing as Markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*, pages 1032–1043. ACM. doi:10.1145/2976749.2978428.

Diehl, C. (2015). Dharma. https://blog.mozilla.org/security/2015/06/29/dharma/.

Even-Mendoza, K., Sharma, A., Donaldson, A. F., and Cadar, C. (2023). GrayC: Greybox fuzzing of compilers and analysers for C. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*, pages 1219–1231. ACM. doi:10.1145/3597926.3598130.

Fioraldi, A., Maier, D., Eißfeldt, H., and Heuse, M. (2020). AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

Fratric, I. (n.d.). Domato, a DOM fuzzer. https://github.com/googleprojectzero/domato.

GitLab B.V. (n.d.). GitLab protocol fuzzer community edition. https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce.

Google Inc. (n.d.). libprotobuf-mutator. https://github.com/google/libprotobuf-mutator.

Grieco, G., Song, W., Cygan, A., Feist, J., and Groce, A. (2020). Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020)*, pages 557–560. ACM. doi:10.1145/3395363.3404366.

Hodován, R., Kiss, Á., and Gyimóthy, T. (2018). Grammarinator: A grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating Test Case Design, Selection, and Evaluation (A-TEST 2018)*, pages 45–48. ACM. doi:10.1145/3278186.3278193 https://github.com/renatahodovan/grammarinator.

Holler, C., Herzig, K., and Zeller, A. (2012). Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458. USENIX Association.

JS Foundation et al. (n.d.). JerryScript. https://www.jerryscript.net.

Lemieux, C., Padhye, R., Sen, K., and Song, D. (2018). PerfFuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*, pages 254–265. ACM. doi:10.1145/3213846.3213874.

Lemieux, C. and Sen, K. (2018). FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*, pages 475–485. ACM. doi:10.1145/3238147.3238176.

LLVM Project (n.d.). libFuzzer – a library for coverage-

guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

Miller, B. (2008). Foreword for fuzz testing book. https://pages.cs.wisc.edu/~bart/fuzz/Foreword1.html.

Miller, B. P., Fredriksen, L., and So, B. (1990). An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44. doi:10.1145/96267.96279.

Mozilla Corporation (n.d.). funfuzz. https://github.com/MozillaSecurity/funfuzz/.

Parr, T. (2013). *The Definitive ANTLR 4 Reference*. The Pragmatic Programmers. https://www.antlr.org.

Pham, V.-T., Böhme, M., Santosa, A. E., Căciulescu, A. R., and Roychoudhury, A. (2021). Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 47(9):1980–1997. doi:10.1109/TSE.2019.2941681.

Veggalam, S., Rawat, S., Haller, I., and Bos, H. (2016). IFuzzer: An evolutionary interpreter fuzzer using genetic programming. In *Computer Security – ESORICS 2016 – 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I*, volume 9878 of *Lecture Notes in Computer Science (LNCS)*, pages 581–601. Springer. doi:10.1007/978-3-319-45744-4_29.

Wang, J., Chen, B., Wei, L., and Liu, Y. (2017). Skyfire: Data-driven seed generation for fuzzing. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594. IEEE. doi:10.1109/SP.2017.23.

Wang, J., Chen, B., Wei, L., and Liu, Y. (2019). Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735. IEEE. doi:10.1109/ICSE.2019.00081.

Yang, X., Chen, Y., Eide, E., and Regehr, J. (2011). Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*, pages 283–294. ACM. doi:10.1145/1993498.1993532.

Zalewski, M. (n.d.). American fuzzy lop. https://lcamtuf.coredump.cx/afl/.