Grammar-Aware Coverage-Guided Fuzzing with Grammarinator and AFL++

Renáta Hodován
Department of Software Engineering
University of Szeged
Szeged, Hungary
hodovan@inf.u-szeged.hu

Ákos Kiss
Department of Software Engineering
University of Szeged
Szeged, Hungary
akiss@inf.u-szeged.hu

***Abstract*—In recent years, program code has been produced at an unprecedented pace, particularly following the rapid adoption of generative AI technologies. Despite this increase in volume, software systems must still be tested with the same level of rigor as before. Consequently, software testing requires an increasing degree of automation. Among automated approaches, randomized test generation—commonly referred to as fuzzing—has demonstrated strong bug-finding capabilities for many years. Nowadays, the majority of fuzzing approaches are based either on coverage-guided techniques that mutate existing test cases or on grammar-driven generation methods. While early coverage-guided methods relied on relatively simple metrics and brute-force mutation strategies to uncover numerous defects, grammar-based generators have gained significant popularity due to their ability to produce syntactically valid and complex inputs. However, these grammar-driven approaches typically cannot reuse existing test cases. Grammarinator is a grammar-based test generator capable of both generating and parsing, and thus also mutating, test cases using ANTLR grammars. This work integrates Grammarinator into one of the most widely used fuzzing frameworks, AFL++, and extends it with several additional features to further improve effectiveness. Experimental results show that the integrated approach achieves significantly higher coverage and discovers more bugs than state-of-the-art fuzzers.**

***Index Terms*—fuzzing, grammar-aware, coverage-guided**

I. Introduction

Nowadays, an increasing number of tasks are being automated, which leads to a continuously growing demand for software. With the rapid adoption of generative AI tools, this growth rate has accelerated even further. However, the increase in code volume must not come at the expense of software quality. As a consequence, the need for thorough and scalable testing techniques is also growing. Similarly to code development, software testing increasingly relies on automated solutions. Fuzzing has long been an established and effective technique for complementing traditional positive testing approaches [1]. While traditional positive testing focuses on verifying whether the software produces the expected output for a given input, fuzzing aims to assess whether the software can handle arbitrary and unexpected inputs. The core idea of fuzzing is to automatically generate a large number of test cases, execute them against the target application, and monitor the execution for anomalous behavior. Such anomalous behavior may include crashes, hangs, unhandled exceptions, or even incorrect outputs, provided that an oracle is available to determine the expected result.

There are multiple ways to categorize fuzzers. One of the most common classification criteria is how the fuzzer generates test inputs. If new test cases are created by mutating elements of an existing seed corpus, the fuzzer is referred to as a *mutation-based* fuzzer [2], [3]. If test cases are generated from a model, grammar, or other formal specification of the input format, the approach is called *generative* fuzzing [4]. These two techniques can also be combined, in which case the fuzzer is referred to as a *hybrid generative–mutational* fuzzer. Another widely used classification criterion is based on the amount of information available about the target application. Accordingly, fuzzers are commonly divided into *black-box*, *gray-box*, and *white-box* fuzzers. In black-box fuzzing, only the input of the target and its observable outputs are available to the fuzzer [5]–[7]. In contrast, white-box fuzzing assumes full access to the target's internal structure, including its source code [8], [9]. Currently, the most widely adopted category is gray-box fuzzing, which lies between these two extremes. In gray-box fuzzing, the fuzzer has partial visibility into the target, typically through lightweight code instrumentation that provides additional runtime feedback beyond the final output (often, but not exclusively, code coverage information) [10], [11]. This feedback is used to guide test generation, primarily mutation, by identifying which test cases were useful, which inputs should be further mutated, and which paths do not warrant additional resources.

Traditionally, *mutation-based gray-box* fuzzers were motivated by the idea of starting from a carefully selected test corpus that demonstrates as many relevant target features as possible while avoiding redundancy. From this corpus, new test cases are generated using very fast, format-unaware mutation operators. A typical characteristic of these mutators is that they treat any input as a raw byte array and apply random byte-level mutations at random positions. Such mutations typically include deletion, insertion, replacement, modification, or repetition of bytes. The mutated inputs are then executed on the target application, and based on the collected feedback (e.g., coverage), the fuzzing harness can determine whether the exploration proceeds in a promising direction. In essence, this approach performs an efficient brute-force search. This

979-8-3315-8589-1/26/$31.00 ©2026 IEEE
DOI 10.1109/SANER-C67878.2026.00055

strategy has proven to be highly effective for simple textual or binary input formats. However, for highly structured formats with complex syntactic and semantic constraints, it mostly produces invalid inputs. Such inputs are typically rejected by the parser at a very early stage. As a consequence, mutation-based gray-box fuzzers have only limited applicability when targeting applications with highly structured input formats, like compilers or interpreters of programming languages. For such targets, approaches that generate or mutate inputs according to an explicit input format description are generally more suitable. Initially, these techniques were limited to *black-box generation* [6]. More recently, however, an increasing number of approaches have begun to exploit the potential of coverage-guided fuzzing. Representative *generative gray-box* examples include Nautilus [12] and Superion [13]. While these techniques are capable of generating format-compliant test cases and guiding generation based on coverage feedback, they are unable to start from existing, human-written seed inputs. This limitation is significant, as seed corpora are of substantial value in fuzzing. All information encoded in the seed corpus does not need to be rediscovered by the fuzzer, which has a direct impact on efficiency.

A recent exception to this limitation combines Grammarinator (an ANTLR grammar-based test generator) with libFuzzer (a coverage-guided fuzzer) through libFuzzer's custom mutator interface [14], resulting in a *hybrid generative-mutational gray-box* fuzzer. However, this approach also has some drawbacks. In particular, libFuzzer is not under active development, has several known issues, and its custom mutator interface provides only limited information about the state of the fuzzing session to the mutators. The present work extends this line of research by integrating Grammarinator with a significantly more modern fuzzer, AFL++, which is under continuous development and provides substantially richer feedback information.

The rest of this paper is organized as follows: In Section II, we give an overview of the tools our work is based on. In Section III, we discuss the key aspects of the Grammarinator-AFL++ integration. In Section IV, we present the results of the evaluation of the integration. In Section V, we overview related works from the literature, and finally, in Section VI, we conclude our paper.

II. Background

A. ANTLR

The ANTLR project [15] was created more than two decades ago with the goal of eliminating the need to manually implement lexical analyzers and parsers required for parse tree construction. Instead, it defined an extended context-free grammar (eCFG) formalism that can be interpreted by the ANTLR toolset. Based on this grammar representation, ANTLR automatically generates lexers and parsers for a wide range of target languages, which can then tokenize input streams and construct parse trees. The project has proven to be highly successful and is widely used in both industrial and academic settings. Over the years, the tooling around the supported grammar formalism has been extended with numerous additional features. ANTLR supports inline code predicates and semantic actions embedded directly into grammars, enabling the expression of conditions and behaviors that cannot be captured using eCFGs alone. It also supports grammar inheritance, allowing base grammars to be specialized and reused. Furthermore, ANTLR provides a listener mechanism that enables user-defined callbacks to be invoked at the beginning or end of successful rule matches during parsing. Beyond extending its feature set, the project has also accumulated several hundred maintained grammar specifications in ANTLR format within the grammars-v4 repository [16]. These grammars are publicly available and can be reused or extended by the community.

B. Grammarinator

Very similar ideas also inspired the Grammarinator project [4], which aimed to reduce the cost of writing test generators for different input formats by relying on grammar-based format descriptions. To this end, the project adopted the widely used grammar specification developed by ANTLR and introduced a grammar processor conceptually similar to ANTLR's tooling, yet independent from it. This processor reads the grammar and writes a test generator implementation, which contains a Python class that provides one method per grammar rule. Each method produces a parse-tree-like representation conforming to the corresponding rule. To prevent unbounded growth of generated trees, the generation process enforces externally configurable size constraints, including a maximum derivation depth (corresponding to the depth of the rule invocation stack) and an upper bound on the number of generated leaf nodes. These leaf nodes of the tree representation contain the output test tokens. In the simplest case, a concrete test case can be obtained by reading these leaves from left to right. However, since ANTLR grammars typically do not encode whitespace information, such nodes are absent from the generated tree. As a result, test generation usually involves an additional serialization step, during which a dedicated serializer inserts the required whitespace at appropriate positions.

Originally, Grammarinator was limited to generating grammar-conforming test cases from scratch, to mutating existing tests by regenerating subtrees, and to crossing over existing tests by swapping compatible subtrees. It was later extended with a diverse set of structure-aware mutators that exploit grammatical information encoded in the trees to perform grammar-conforming transformations. These include operations on quantified or repetitive constructs (e.g., deletion, duplication, and reordering), as well as the replacement of compatible subtrees.

A major advantage of Grammarinator over other structure-aware fuzzers was that it was built on top of a widely adopted parser grammar format. Firstly, this allowed it to avoid the cost of writing grammars in many cases. Secondly, since its grammars were fully compatible with ANTLR, it was able to construct seed corpora (i.e., trees that could be mutated

further) from human-written test inputs. A clear limitation, however, was that Grammarinator operated exclusively in a black-box manner, as it did not provide any form of guiding harness.

C. Grammarinator + LibFuzzer

Recognizing the lack of a guiding harness, Grammarinator was integrated with libFuzzer in a previous work [14]. LibFuzzer [10] was one of the first widely adopted coverage-guided in-process fuzzers and was developed as part of the LLVM project. According to its core design, users are required to implement the target API invocations in a predefined function (named `LLVMFuzzerTestOneInput`). This function must then be linked together with the library implementing the actual functionality (compiled with clang using the `-fsanitize=fuzzer` option). This setup provides the instrumentation and runtime feedback required by the fuzzing harness. In this configuration, `LLVMFuzzerTestOneInput` is repeatedly invoked by a special main function in an infinite loop, where each iteration supplies new test inputs. These test inputs are generated by libFuzzer from a seed corpus using structure-unaware mutation operators.

Beyond `LLVMFuzzerTestOneInput`, libFuzzer provides two additional optional user-defined hook functions (`LLVMFuzzerCustomMutate` and `LLVMFuzzerCustomCrossOver`). By defining these functions, users can implement specialized, structure-aware mutation strategies. The Grammarinator–libFuzzer integration was realized through these interfaces after C++ support was added to Grammarinator alongside its original Python implementation. These optional hooks expose a very simple API. Depending on whether mutation or crossover is performed, they provide access to one or two input byte arrays and a pointer to an output byte array where the result must be written.

To avoid the overhead caused by repeated parsing, the corpus was parsed in advance into the required tree-based representation. However, since all the hook functions must accept and produce byte arrays, working with trees means that these functions need to accept and produce trees encoded as byte arrays. Hence, during each mutation or crossover step, this encoded tree representation had to be decoded, transformed, re-encoded, and then decoded again within `LLVMFuzzerTestOneInput` to serialize the tree back into a concrete test input. Although the authors attempted to mitigate the overhead of repeated encoding and decoding through caching, this optimization was only effective up to a certain point. Due to the limited information provided by libFuzzer's custom mutator API, it was not possible to reduce the full encode–decode cycle to only saving and restoring test cases deemed useful. While the integration itself was successful, the Grammarinator-based version also inherited known libFuzzer issues, such as unexpected hangs when running in parallel fork mode.

D. AFL++

American Fuzzy Lop (AFL), originally written and released by Michal Zalewski, was the first widely adopted coverage-guided fuzzer, whose exceptional effectiveness paved the way for guided fuzzing approaches [11]. Unlike library-focused fuzzers, AFL is capable of testing entire applications by compiling them using its own compiler wrappers. These wrappers can transparently produce instrumented binaries suitable for fuzzing without requiring changes to the existing build setup. The resulting instrumented binary is then able to communicate with AFL's fork server to provide feedback information about test executions. The original AFL implementation led to the creation of numerous independent forks, each aiming to improve or extend specific aspects of the fuzzer. However, these efforts often resulted in fragmented and experimental variants that were difficult to track and maintain. To address this fragmentation, the AFL++ project was created with the goal of consolidating the most effective features across these efforts [17]. AFL++ combines advances in instrumentation, guidance strategies, debugging support, and custom mutator interfaces into a single, actively maintained framework. Beyond core fuzzing functionality, the project provides a comprehensive fuzzing ecosystem. This ecosystem includes support for corpus and test case minimization, parallel and distributed fuzzing, result visualization, and input format analysis, among other features. The custom mutator API of AFL++, together with its active and continuous development, motivated us to design the Grammarinator integration for AFL++ as well.

III. Grammarinator with AFL++

AFL++ provides a custom mutator interface that is conceptually similar to the extension mechanisms of libFuzzer, yet exposes substantially richer internal state and lifecycle hooks. Instead of linking the mutator directly into the target binary, AFL++ loads the mutator from a shared library specified via an environment variable (`AFL_CUSTOM_MUTATOR_LIBRARY`). The library should implement a predefined set of mandatory and optional callback functions that are invoked at well-defined points of the fuzzing loop.

A. Custom Mutator Architecture

The Grammarinator-AFL++ integration is realized by implementing AFL++'s custom mutator API, including hooks for queue management, mutation, trimming (a.k.a. test case reduction or minimization), and post-processing. In contrast to libFuzzer, which operates in a largely stateless manner across inputs, AFL++ exposes its internal fuzzing state, which is provided to the custom mutator during initialization and remains accessible throughout the fuzzing session. We extend this data by maintaining a mutator-specific state that stores grammar-aware data structures, including the currently selected input tree, the most recent mutated tree, and auxiliary buffers.

B. In-Memory Tree Representation

A key advantage of this integration is the ability to keep the current tree in memory. Unlike libFuzzer-based grammar

fuzzing, where each mutation typically requires re-decoding and re-encoding the test case, the AFL++ integration allows the mutator to directly operate on an in-memory tree representation.

When AFL++ selects a queue entry, a custom mutator hook (`afl_custom_queue_get`) signals that a new test case is being processed, upon which the encoded tree is decoded only once and stored as the current in-memory tree. Subsequent mutation steps operate on clones of this tree, avoiding repeated decoding overhead. Both the custom mutator and auxiliary hooks (e.g., trimming and queue insertion) access the same in-memory representation, reducing computational cost.

C. Grammar-Aware Crossover with a Subtree Population

a) Motivation and design rationale: Guided fuzzers that employ evolutionary mutation strategies typically combine mutation with crossover (a.k.a. recombination or splicing). In this process, a fragment of an already accepted test case (the donor) is inserted into another test case currently under mutation (the recipient), with the goal of transferring previously useful structure into a new syntactic context and thereby triggering novel behavior.

AFL++ natively supports this paradigm by providing a second input buffer to the custom fuzzing hook. The mutator may interpret this buffer as a donor and decide dynamically whether to perform mutation or crossover. However, this design introduces a constant overhead, as the donor input is always loaded, regardless of whether it is ultimately used. More importantly, in a structure-aware setting, crossover requires the donor to contain a subtree that is compatible with both the recipient context and the selected crossover operator; otherwise, the operation fails and the mutation step is discarded, resulting in an unproductive iteration.

To address these limitations, we disable AFL++'s native donor-based crossover mechanism (by defining the `afl_custom_splice_optout` hook), ensuring that the custom mutator never receives a second external input. Instead, we replace traditional crossover with a grammar-aware subtree population maintained entirely in global mutator state. This design decouples recombination from individual donor inputs and allows crossover-like transformations to select from the accumulated set of subtrees.

b) Subtree population and collection: The subtree population serves as a global in-memory cache of subtrees extracted from test cases that AFL++ deems interesting. Whenever AFL++ reports a new queue entry (via the `afl_custom_queue_new_entry` hook), or after a successful grammar-aware trimming step, the current in-memory tree is decomposed into its constituent subtrees. These subtrees are then added to the population and become available as potential donors for future recombination steps.

Over time, the population becomes a diverse set of subtrees that have already contributed to coverage growth or other interesting behavior. As a result, the probability of finding a structurally compatible donor for a given syntactic context increases rapidly during the fuzzing campaign.

c) Grammar-aware crossover operators: Recombination is realized through two dedicated grammar-aware operators that draw donors exclusively from the subtree population. The first operator ("replace from subtree population") performs subtree replacement by selecting a node in the recipient tree and replacing it with a compatible subtree from the population. The second operator ("insert quantified from subtree population") targets nodes that correspond to quantified elements of the CFG and inserts an additional compatible subtree, provided that the quantifier bounds are respected. This enables controlled growth of repeated structures while preserving grammatical correctness. Both operators rely on the subtree population for donor selection and therefore avoid the need to decode or inspect an external donor input.

d) Compatible and frequency-aware selection: When selecting a donor from the subtree population, candidates are first filtered for compatibility and size constraints derived from the mutation context. Among the remaining candidates, selection is weighted inversely by the frequency with which each subtree has appeared so far. This biases recombination toward rarer substructures, promoting diversity while still exploiting fragments that have previously led to interesting behavior.

e) Comparison to donor-based crossover: In contrast to traditional donor-based crossover, where recombination relies on the incidental presence of a compatible subtree in a single donor input, the subtree population aggregates suitable fragments across the entire fuzzing history, substantially reducing failed crossover attempts and eliminating the overhead of handling an unused second input.

This stands in contrast to Grammarinator's libFuzzer integration, where the stateless custom mutator interface provides no feedback about mutation outcomes. Consequently, maintaining a selectively populated subtree pool is infeasible in that setting, as successful fragments cannot be distinguished from ineffective ones.

D. Memoization of Generated Test Cases

To reduce redundant executions, we employ a memoization mechanism that filters out duplicate test cases before they are passed to the target. Each generated test case is checked against a bounded cache of recently seen tests. (More precisely, it is not the actual test cases that are stored in the cache, but a hash is computed for them and comparison is performed on these hashes.) If a duplicate is detected, the test case is discarded, thereby avoiding a potentially expensive target execution with no expected coverage gain.

The cache retains only the most recent test cases, where the exact number is externally configurable, providing a simple and effective trade-off between memory usage and duplicate suppression. Since this mechanism operates solely on the generated inputs and does not depend on fuzzer-specific feedback, it is not tied to the AFL++ integration and has therefore been backported to the Grammarinator-libFuzzer integration as well.

E. Grammar-Aware Trimming

After AFL++ identifies an input as interesting (e.g., because it exercises new coverage), it attempts to trim the test case in order to remove irrelevant parts that do not change the overall coverage of the test case. The goal of trimming is to minimize the input while preserving the observed interesting behavior, in our case maintaining the same coverage. Smaller test cases improve subsequent mutation effectiveness by focusing mutations on structurally relevant regions.

AFL++'s default trimming strategy operates at the byte level by randomly removing ranges of bytes. While effective for unstructured inputs, this approach is incompatible with our setting: since inputs are encoded representations of trees, arbitrary byte removal would corrupt the encoding and render the test case undecodable, effectively halting further grammar-based fuzzing. Although AFL++ allows trimming to be disabled entirely, doing so would discard a potentially powerful optimization.

Instead, we implement a grammar-aware trimming strategy using AFL++'s custom trimming hooks. Our approach applies the minimizing delta-debugging (ddmin) algorithm [18] to those nodes of the tree that represent optional constructs, i.e., which correspond to quantified elements of the eCFG. The set of nodes is iteratively partitioned into smaller subsets following the ddmin algorithm. In each iteration, a subset of nodes is tentatively removed from the tree and the resulting test case is evaluated by AFL++ to determine whether it still exhibits the same coverage. If the coverage is preserved, the reduction is accepted and further refined; otherwise, the change is reverted and alternative subsets are explored.

Since the trees can be large, exhaustively exploring all trimming configurations does not justify the required computational effort. Therefore, the maximum number of trimming steps is externally parameterized, allowing the trade-off between trimming aggressiveness and computational overhead to be tuned.

This design yields a fully grammar-aware trimming mechanism that operates directly on tree structures. Beyond improving the effectiveness of ongoing fuzzing campaigns, the same implementation can be reused by other components of the AFL++ ecosystem. In particular, `afl-tmin` supports loading the same custom mutator shared library and invokes only the trimming-related hooks. As a result, our approach also enables structure-aware reduction of initial seed corpora, improving their quality before fuzzing begins.

IV. Evaluation

In this section we detail the evaluation of the Grammarinator-AFL++ integration discussed above, which has been made publicly available in Grammarinator 26.1.

A. Target Selection

Grammar-based fuzzers are particularly well suited for targets that expect highly structured inputs, as the additional cost of structure-aware generation and mutation can be compensated by the higher proportion of syntactically valid and semantically meaningful test cases. Consequently, we selected JavaScript as the target input format and evaluated our approach on the JerryScript JavaScript engine (version 3.0.0), which processes complex, deeply structured inputs.

JerryScript was also used in prior work on integrating Grammarinator with libFuzzer, enabling a direct and meaningful comparison with previously published results under similar conditions.

B. Baseline Selection

As baselines for Grammarinator-AFL++, we selected four fuzzers. First, AFL++ [17], which naturally serves as a baseline since our approach directly extends it with custom mutators. Second, the Grammarinator-libFuzzer integration [14], and third, libFuzzer [10] as its underlying baseline. As our fourth baseline, we chose GrammarMutator [19], another custom mutator approach integrated into AFL++ which pursues similar goals by enabling structure-aware fuzzing through grammar-based mutation. GrammarMutator relies on a custom grammar format and does not support direct parsing of existing input corpora. While its grammar format could theoretically be converted into an ANTLR-compatible subset to allow partial parsing, this functionality has been broken for several years[1]. (We excluded standalone Grammarinator from the baselines because prior work showed that the Grammarinator-libFuzzer integration outperforms it.)

C. Input Preparation

a) Input corpus: The preparation of the seed corpus was performed in multiple stages. We began by collecting the full test suite of the JerryScript project, consisting of 1,167 JavaScript test cases. Using `afl-cmin` together with an AFL-instrumented JerryScript binary, we minimized this corpus to a subset of 256 inputs that collectively preserved the original coverage.

From this minimized source-level corpus, we derived a corresponding tree corpus using Grammarinator's parsing utilities and the official JavaScript parser grammar from grammars-v4[2]. Finally, we applied trimming to both the source-based and tree-based corpora. For the source corpus, which is used by the baseline AFL++ and libFuzzer configurations, we employed the standard `afl-tmin` tool. For the tree corpus, used by both the AFL++ and libFuzzer Grammarinator integrations, we ran `afl-tmin` with the custom shared library loaded, enabling grammar-aware trimming through the custom trimming hooks described earlier.

While this multi-stage setup provides a fair and comparable starting point for the AFL++, Grammarinator-AFL++, libFuzzer, and Grammarinator-libFuzzer configurations, GrammarMutator cannot directly operate on existing input corpora. Therefore, for this fuzzer, we generated a seed corpus using its own generator, consisting of 256 inputs, matching the size of the minimized seed corpora used by the other configurations.

[1] https://github.com/AFLplusplus/Grammar-Mutator/issues/35

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

b) Input grammar: In addition to the input corpus, grammar-based fuzzing with Grammarinator requires a grammar definition from which the test generator is derived. By default, this grammar is identical to the parser grammar used to construct the seed tree corpus. However, the grammar can be extended to better support generation-oriented use cases.

In the JavaScript grammar, the `identifier` rule is defined as a regular expression describing the syntactic form of valid identifiers. While this definition is sufficient for parsing, it is poorly suited for generation, as it rarely produces identifiers corresponding to built-in JavaScript entities such as global functions, classes, or well-known member names recognized by the engine. To address this limitation, we introduced a derived grammar in our experiments that overrides and augments the `identifier` rule with an explicit enumeration of such built-in identifiers. This approach is conceptually analogous to the use of user-provided dictionaries in traditional AFL++ or libFuzzer setups, where mutation is biased toward semantically meaningful string constants. To ensure a fair comparison, the same set of identifier strings was also included in the dictionaries supplied to the baseline fuzzers, allowing them to benefit from the same domain knowledge.

D. Setup

We ran fuzzing sessions for 48 hours with each fuzzer, repeating every configuration five times to account for variability introduced by randomization.

All Grammarinator-based fuzzers were configured with a maximum derivation depth of 25, an upper bound of 500 tokens, a memoization cache size of 1,000 entries, and a limit of 200 trimming steps per test case. The JerryScript fuzz target was executed with a per-input timeout of 1 second.

The libFuzzer-based configurations were executed in fork mode with a single worker process. This setup ensures that, in the event of a crash or timeout, the target process can be restarted by the supervising process without requiring the entire fuzzing session to be restarted.

E. Results

One of the most expressive metrics for evaluating the effectiveness of a fuzzing approach is the amount of code coverage achieved by the generated test cases. Since a fuzzer can only expose faults along execution paths that it is able to reach, higher coverage generally correlates with a greater potential for bug discovery. Consequently, we measured and compared the code coverage achieved by each evaluated approach.

Because libFuzzer- and AFL++-based fuzzers rely on different coverage representations, direct comparison is not possible. To obtain comparable measurements, we re-executed the corpora generated by all fuzzers using AFL++'s coverage measurement tool, `afl-showmap`. This allowed us to compute coverage metrics in a uniform manner across all evaluated fuzzers.

Figure 1 presents the aggregated edge coverage progression over time for all five fuzzing approaches, based on five

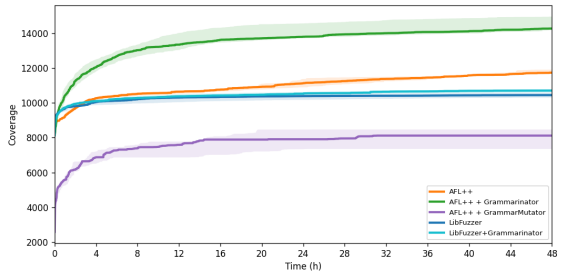


Fig. 1: Change of coverage over time while fuzzing JerryScript for 48 hours.

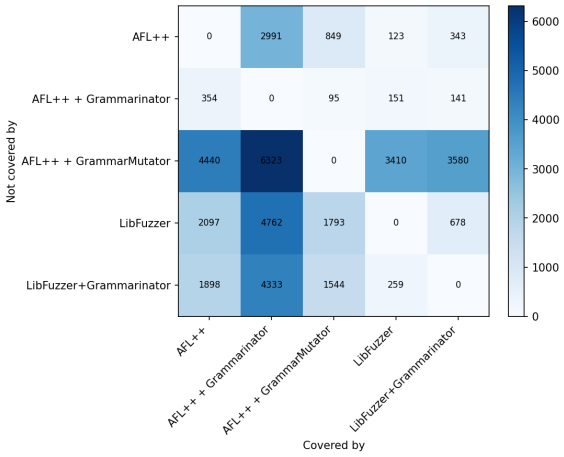
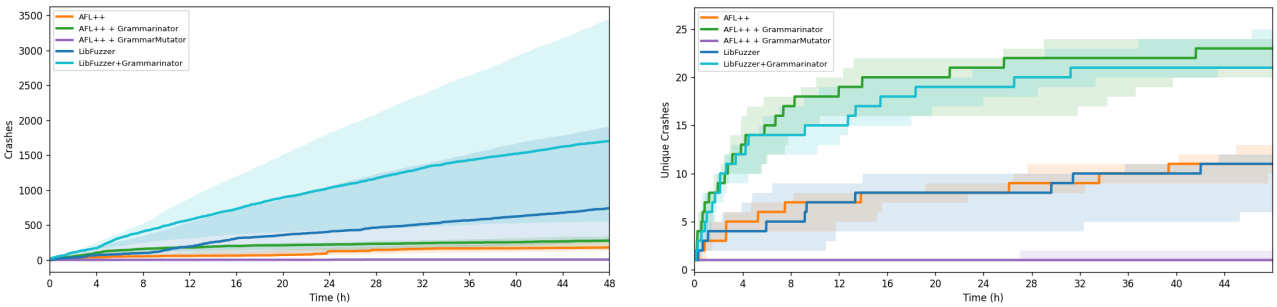


Fig. 2: Pairwise comparison of uniquely covered code regions across five runs per fuzzer after 48 hours of fuzzing JerryScript.

independent runs each. Solid lines indicate the median coverage across runs, while the shaded regions above and below represent the minimum and maximum values observed. The results show a clear ordering among the evaluated approaches: AFL++ integrated with Grammarinator achieves the highest coverage, followed by baseline AFL++, then the libFuzzer-based configurations, and finally AFL++ using GrammarMutator.

Since `afl-showmap` reports coverage using unique edge identifiers, the collected data also enables a more fine-grained comparison beyond absolute coverage numbers. Specifically, it allows us to analyze the uniqueness of coverage between different fuzzers.

The matrix shown in Figure 2 compares each pair of fuzzers in terms of uniquely discovered coverage. Cells indicate how many coverage elements are found exclusively by the column fuzzer compared to the row fuzzer. Under this interpretation, the most effective fuzzer is characterized by darker columns (finding more unique coverage) and lighter rows (missing less coverage relative to others). This analysis further confirms that AFL++ with Grammarinator discovers the largest amount of unique coverage. It is followed by baseline AFL++, then the



(a) (b)

Fig. 3: Number of discovered crashes and unique crashes after fuzzing JerryScript for 48 hours. (Results combined from five runs.)

libFuzzer variants, and finally AFL++ with GrammarMutator. Notably, while AFL++ with GrammarMutator does cover some code not reached by baseline AFL++, it is unable to compensate for its inability to leverage an existing input corpus. As a result, the benefits of structure-aware mutation alone are insufficient to close the gap to the other approaches.

The second key metric, and the ultimate goal of fuzzing, is bug-finding effectiveness. Figure 3(a) reports the total number of crashes detected by each fuzzer during the 48-hour fuzzing campaign, while Figure 3(b) illustrates the number of discovered unique crashes over time. The results show that both baseline libFuzzer and the Grammarinator-libFuzzer integration triggered the highest number of crashes overall. However, when considering unique crashes, AFL++ with Grammarinator clearly outperformed the other approaches, followed by the Grammarinator-libFuzzer integration. This indicates that, although the libFuzzer-based approaches encountered crashes frequently, they often rediscovered the same underlying faults, resulting in a high degree of redundancy.

Finally, by leveraging AFL++'s introspection capabilities, we recorded how often each Grammarinator-specific custom mutator contributed to test cases that discovered new coverage. The results from five independent runs were aggregated and plotted over normalized time. Figure 4 shows that the subtree pool–based *replace* mutator was by far the most effective contributor to coverage growth. The subtree insertion mutator, which also draws from the subtree pool, was likewise dominant compared to subtree regeneration and local replacement operators. These results empirically validate the design choice of introducing a subtree population as a primary recombination mechanism.

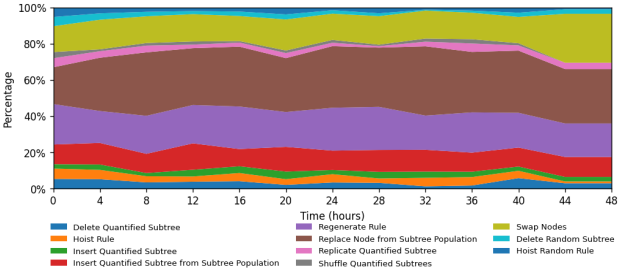


Fig. 4: Percentage of Grammarinator-based mutators discovering new coverage during 48 hours of fuzzing JerryScript. The results of 5 runs are aggregated.

V. Related Work

The present work is most closely related to the libFuzzer-based integration of Grammarinator [14]. While the libFuzzer-based integration supports grammar-based generation and mutation with coverage guidance, its stateless execution model results in repeated encode–decode overhead, provides no explicit feedback on interesting inputs, and cannot support grammar-aware trimming. Furthermore, it does not provide crossover operations based on an in-memory subtree pool.

Nautilus [12] is one of the most prominent grammar-based, coverage-guided fuzzers. It is a standalone fuzzer that employs AFL-based instrumentation together with a custom guidance mechanism. Nautilus relies on a proprietary grammar format and does not support starting from an existing seed corpus. Although earlier versions provided a converter from ANTLR grammars, this feature was removed in later releases. Its mutation strategies include subtree regeneration, crossover between randomly selected seeds, random recursion repetition, and AFL-style byte-level mutations applied to leaf nodes.

The design of Nautilus subsequently influenced AFL++'s GrammarMutator custom mutator [19]. GrammarMutator reimplements Nautilus's mutation strategies but integrates them directly into the AFL++ harness [17]. Unlike Nautilus, it selects splice donors from a dedicated chunk store rather than from random seed inputs. It also implements a simple top-down trimming strategy, but does not perform ddmin-style minimization.

Superion [13] represents an earlier AFL-based, structure-aware approach that also relied on ANTLR to generate syntactically valid inputs. It combined AFL's default mutator with dictionary-based token replacement and a tree-level crossover operation based on successful parsing of multiple

inputs. While Superion supported seed corpora, its reliance on repeated parsing during fuzzing time introduced significant performance overhead, and the project has not been maintained for several years.

Several existing fuzzers aim to achieve structure awareness without relying on explicit grammars, instead leveraging input-to-state correspondence, comparison solving, taint analysis, or lightweight symbolic execution, primarily targeting binary or protocol-level input formats [20]–[24].

VI. Conclusion and Future Work

In this work, we designed and implemented the integration of the grammar-based test generator Grammarinator into AFL++, and further extended it by exploiting the additional feedback provided by the AFL++ framework. First, we leveraged AFL++'s notifications about interesting test cases to construct a subtree population, which enabled the definition of two novel grammar-aware crossover operators. These operators proved to be the dominant contributors to fuzzing effectiveness in our evaluation. Second, by utilizing AFL++'s custom trimming hooks, we developed a ddmin-based, structure-aware trimming strategy that can be applied both during fuzzing and in the seed corpus preparation phase. In addition, we introduced a memoization mechanism that reduces the generation and execution of duplicate test cases, thereby avoiding unnecessary target executions. Our evaluation, conducted against four state-of-the-art fuzzers, demonstrates that the proposed approach achieves superior results in both code coverage and bug-finding capability.

In future work, we plan to conduct a detailed ablation study to isolate the individual impact of each extension, which we could not include in the present paper due to space constraints. Specifically, we intend to quantify the effect of grammar-aware trimming both during corpus preprocessing and throughout the fuzzing campaign. We also aim to measure the added value of the subtree population and the two subtree-based crossover operators it enables. Finally, we will evaluate the impact of memoization by reporting how many target executions were avoided and whether the saved time translates into improved coverage and bug discovery. Beyond the ablation study, we plan to assess the effectiveness of the extended AFL++ approach on additional fuzz targets.

Acknowledgment

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

[1] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990, doi:10.1145/96267.96279.

[2] A. Helin, "Radamsa." [Online]. Available: https://gitlab.com/akihe/radamsa

[3] S. Hocevar, "zzuf: Multi-purpose fuzzer." [Online]. Available: https://github.com/samhocevar/zzuf

[4] R. Hodován, Á. Kiss, and T. Gyimóthy, "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)*. ACM, Nov. 2018, pp. 45–48, doi:10.1145/3278186.3278193. [Online]. Available: https://github.com/renatahodovan/grammarinator

[5] GitLab B.V., "GitLab protocol fuzzer community edition." [Online]. Available: https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce

[6] Mozilla Corporation, "funfuzz." [Online]. Available: https://github.com/MozillaSecurity/funfuzz/

[7] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *21st USENIX Security Symposium (USENIX Security 12)*. USENIX Association, Aug. 2012, pp. 445–458.

[8] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, pp. 20–27, Jan. 2012, doi:10.1145/2090147.2094081.

[9] C. Cadar, D. Dunbar, and D. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, 2008, pp. 209–224.

[10] LLVM Project, "libFuzzer – a library for coverage-guided fuzz testing." [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[11] M. Zalewski, "American fuzzy lop." [Online]. Available: https://lcamtuf.coredump.cx/afl/

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

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

[14] R. Hodován and Á. Kiss, "Grammarinator meets LibFuzzer: A structure-aware in-process approach," in *Proceedings of the 20th International Conference on Software Technologies (ICSOFT 2025)*. SciTePress, Jun. 2025, pp. 178–189, doi:10.5220/0013571500003964.

[15] T. Parr, *The Definitive ANTLR 4 Reference*. The Pragmatic Programmers, 2013. [Online]. Available: https://www.antlr.org

[16] ANTLR Community, "grammars-v4: Antlr v4 grammars repository," https://github.com/antlr/grammars-v4.

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

[18] R. Hildebrandt and A. Zeller, "Simplifying failure-inducing input," in *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '00)*. ACM, Aug. 2000, pp. 135–145, doi:10.1145/347324.348938.

[19] "Grammar-Mutator." [Online]. Available: https://github.com/AFLplusplus/Grammar-Mutator

[20] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS '16)*. Internet Society, Feb. 2016, doi:10.14722/ndss.2016.23368.

[21] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2018, pp. 711–725, doi:10.1109/SP.2018.00046.

[22] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1980–1997, 2021, doi:10.1109/TSE.2019.2941681.

[23] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence," in *Network and Distributed Systems Security (NDSS) Symposium 2019*. Internet Society, Feb. 2019, doi:10.14722/ndss.2019.23371.

[24] R. Dutra, R. Gopinath, and A. Zeller, "FormatFuzzer: Effective fuzzing of binary file formats," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, Feb. 2024, doi:10.1145/3628157.