Repair-Driven Greybox Fuzzing

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

We present *repair-driven greybox fuzzing*, a new approach to greybox fuzzing that combines the strengths of unstructured, byte-level input mutation and grammar-guided input generation. By increasing input diversity while preserving input validity, repair-driven fuzzing promises to improve bug-finding ability for systems under test such as programming language interpreters that consume highly structured inputs. Our idea is to first mutate an input using a standard byte-level mutator, typically leading to an invalid input, and then *repair* the input using a grammar. Aggressively breaking and then repairing an input provides an effective way to reach parts of the input space that would be left unexplored by both byte-level and idiomatic grammar-based mutations. We put this idea into practice via RepairFuzz, a new greybox fuzzer based on AFL++, leveraging the byte level mutations of AFL++ and using the CPCT$^+$ error recovery algorithm for input repair. We present experiments applying RepairFuzz to six SUTs covering four programming language input formats (Lua, PHP, JavaScript and Ruby), and present an experimental comparison with AFL++, Grammarinator and Nautilus, the state-of-the-art in standard greybox fuzzing and grammar-guided fuzzing. Our evaluation shows that RepairFuzz was able to find 13 confirmed bugs that were previously unknown, including 9 that were not found by AFL++, Grammarinator or Nautilus. Further, RepairFuzz yields absolute increases in code coverage for several SUTs and substantial complementary code coverage across all.

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

Additional Key Words and Phrases: Grammar-based fuzzing, mutations, input repair

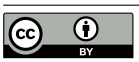
ACM Reference Format:
Bachir Bendrissou, Alastair F. Donaldson, and Cristian Cadar. 2026. Repair-Driven Greybox Fuzzing. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA138 (October 2026), 23 pages. https://doi.org/10.1145/3832229

1 Introduction

Greybox fuzzing [49, 78] is a popular technique for automatically finding bugs in software and has achieved significant impact in practice, finding a large number of bugs and security vulnerabilities in widely-used software [36]. Its most popular variant, *coverage-guided mutation-based fuzzing*,[1] operates on a queue of inputs that are processed in a mutate-execute-enqueue loop: (1) An input is extracted from the input queue and various byte-level mutation operators are applied to generate a set of *mutated inputs*; (2) The mutated inputs are executed on the system under test (SUT); (3) Those mutated inputs that trigger bugs in the SUT, causing it to fail, are saved for further investigation; and (4) Those mutated inputs that achieve new coverage in the SUT are added to the input queue. Unlike greybox fuzzing, *blackbox fuzzing* does not employ any form of execution feedback. This

[1] In this paper we use *greybox fuzzing* to mean *coverage-guided mutation-based fuzzing*, while acknowledging that there are other forms of greybox fuzzing.

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



ACM 2994-970X/2026/10-ARTISSTA138
https://doi.org/10.1145/3832229

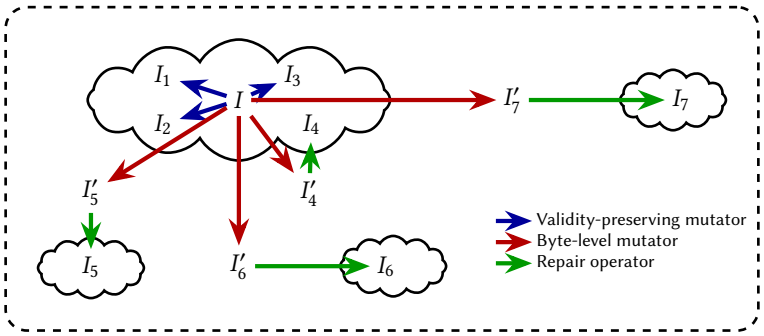


Fig. 1. Validity-preserving mutation vs. byte-level mutation with repair. Bubbles represent valid input regions.

makes blackbox fuzzing applicable to a wider range of SUTs, but the absence of feedback (such as code coverage) limits its ability to explore interesting regions of the SUT to discover deeper bugs.

Despite its success, greybox fuzzing struggles with SUTs that take highly-structured inputs, such as programming language interpreters that need well-formed programs to exercise the core of their implementation: the byte-level mutation operators employed by popular fuzzers such as AFL [78] and libFuzzer [49] often fail to yield valid inputs [37]. To mitigate this, prior work has proposed using *validity-preserving mutation operators* that exploit existing knowledge of syntactic and semantic input constraints [2, 32, 62]. To our knowledge, Nautilus [2] was the first system to combine greybox and grammar-based fuzzing, by replacing the mutation operators of greybox fuzzers with mutation operators that walk a grammar to construct valid mutated inputs.

While promising, grammar-based mutation operators are biased by the way grammars are written, thus failing to guarantee a uniform exploration of the space of valid inputs. Consequently, certain classes of valid inputs may be over-represented and others under-represented in the generated test cases, leading to uneven exploration that can leave some SUT functionality under-tested.

In this work, we propose a new greybox fuzzing approach, *repair-driven* greybox fuzzing, where we combine input grammars with byte-level mutations to achieve a more thorough exploration of the input space. Instead of using constrained validity-preserving mutators, we employ byte-level mutators, as used in traditional greybox fuzzing implementations. However, after a byte-level mutation we *repair* the input using a grammar. The basic workflow of our approach is as follows:

(1) Start with an input that conforms to a grammar—an existing seed input or one generated by a grammar-based generator.
(2) Perform a byte-level mutation that (with a high probability) makes the input invalid according to the grammar.
(3) If the input is no longer valid, repair it so that it conforms to the grammar.

Our hypothesis is that aggressively breaking and then repairing inputs provides an effective way to reach parts of the input space that would be left unexplored by both byte-level and idiomatic grammar-based mutations when applied in isolation.

Figure 1 illustrates our idea and contrasts it to techniques based on validity-preserving mutation, with bubbles representing valid input regions. Generic validity-preserving mutations, such as those based on a grammar, may be effective for generating distinct inputs in the same region, indicated by the blue arrows. However, without incredibly specialised mutations, it is unlikely that radically different inputs will be generated, indicated by the lack of blue arrows between distinct regions.

In contrast, byte-level mutations can generate widely diverse but usually *invalid* inputs, indicated by the red arrows. Our insight is that *after repair*, indicated by green arrows, such mutations can yield valid inputs in diverse parts of the input space. Thus the combination of byte-level mutations

and grammar-based repair has the potential to achieve better coverage of valid input regions compared with validity-preserving mutations alone.

Based on this idea we present the design and implementation of RepairFuzz, a new greybox fuzzer that combines the byte-level mutations of AFL++ [33] with the error recovery algorithm of CPCT$^+$ [28]. To fuzz-test an SUT, RepairFuzz requires a grammar for the SUT's input format and (as standard in greybox fuzzing) a source of example inputs for mutation.

We present experiments applying RepairFuzz to six SUTs covering four programming language input formats (Lua, PHP, JavaScript and Ruby); this captures all SUTs and input formats considered in prior state-of-the-art work on grammar-based greybox fuzzing [2], plus two additional SUTs. We consider two configurations of RepairFuzz—one where we attempt to repair *every* invalid mutated input, and one where we do so for only 50% of such inputs (selected at random). The latter configuration is particularly interesting because it allows for the application of multiple successive validity-breaking byte-level mutations, whose subsequent repair may further diversify the kinds of valid inputs that are eventually obtained. We compare RepairFuzz with AFL++ and the grammar-based fuzzer Grammarinator [40] via two sets of experiments: experiments where the grammar-based greybox fuzzer Nautilus is used as a source of inputs for mutation, and experiments where a fixed corpus of seed inputs derived from the test suites of SUTs is used. Our evaluation shows that RepairFuzz was able to find 13 confirmed bugs that were previously unknown, including 9 that were not found by AFL++ or Nautilus (state-of-the-art greybox fuzzers), or Grammarinator (a state-of-the-art blackbox fuzzer). All of the 9 bugs that are unique to RepairFuzz have already been fixed in response to our reports. Furthermore, our results show that for several SUTs, RepairFuzz yields increases in percentage code coverage compared with fuzzing using AFL++ or Grammarinator. While coverage gains over Nautilus are expected in experiments where both RepairFuzz and AFL++ operate on Nautilus-generated inputs, our results also show that RepairFuzz yields substantial *complementary* coverage compared with AFL++ and Grammarinator across *all* SUTs; i.e., RepairFuzz can cover non-trivial portions of the SUT codebases that are not exercised by AFL++ and Grammarinator. Additionally, our repair strategy was shown to be efficient, repairing the majority of invalid inputs generated by byte-level mutation within a 10 ms timeout. We also observed performance improvements in RepairFuzz when reducing the repair timeout parameter, both with respect to code coverage and bug finding.

In summary, our main contributions are:

(1) The idea of repair-driven greybox fuzzing, where byte-level mutation and grammar-based repair are combined to drive greybox fuzzing with more diverse valid inputs.
(2) The design and implementation of this idea in a new fuzzing tool, RepairFuzz, which enhances the baseline AFL++ fuzzer with repair facilities based on the CPCT$^+$ algorithm.
(3) An evaluation comparing RepairFuzz with state-of-the-art greybox and blackbox fuzzers across six SUTs covering four complex input formats, demonstrating its effectiveness for bug finding and coverage, with 13 previously unknown bugs discovered, 9 of which were not found by competing tools.

2 Background

We present background on byte-level mutations (§2.1), structure-aware mutations (§2.2) and the CPCT$^+$ grammar-based repair algorithm (§2.3).

2.1 Byte-Level Mutations

Traditional mutation-based fuzzers rely on byte-level mutations to generate new inputs. The key mutation stages used by AFL++ [33]—the state-of-the-art tool that we use in our experiments—are:

(1) **Havoc:** This stage applies a wide range of random mutations at random positions in the input string, e.g. bit flips, deletion/duplication of byte blocks, and insertion of random data.
(2) **Splice:** This operator combines two randomly selected inputs. A splice point is chosen arbitrarily, and the two inputs are merged at that position to form a new input.
(3) **Trim:** This pre-processing step aims to shrink an input while preserving its execution profile. The goal is to produce a minimal input that still triggers the same code coverage as the original.

AFL++ also supports an API for integrating custom mutation logic, allowing users to plug in new mutators without modifying or forking the core fuzzer. We leverage this API to implement our grammar-aware repair process as a post-mutation step.

2.2 Structure-Aware Mutations

Structure-aware fuzzers use an input specification (typically a context-free grammar) to mutate test inputs in a manner that preserves syntactic validity, so that mutated inputs pass the initial parsing or validation stage of an SUT and exercise deeper application logic [2, 38, 41, 62]. Structure-aware fuzzers typically operate at the level of abstract syntax trees (ASTs). Common mutations include subtree deletion, duplication, and replacing a subtree with another from the same AST or from the AST associated with another corpus input. AST-based mutation is particularly effective for complex formats, where random byte-level edits are unlikely to preserve grammatical correctness.

A structure-aware fuzzer requires a source of inputs to mutate. This can be a corpus of existing inputs, or alternatively the fuzzer may construct new inputs from scratch by randomly walking the grammar, starting from the start symbol and expanding non-terminals until a complete input is generated. State-of-the-art structure-aware fuzzers include Nautilus [2], Grammarinator [40], and Kitten [76], which have proven effective in detecting bugs and vulnerabilities in SUTs that consume a range of programming language input formats. Nautilus is a greybox fuzzer, while Grammarinator and Kitten are blackbox. Nautilus generates inputs from scratch from a given grammar and subsequently mutates them, but it does not support using a corpus of existing inputs. In contrast, Grammarinator and Kitten can operate with or without a corpus.

2.3 The CPCT$^+$ Grammar-Based Repair Algorithm

Modern compilers and IDEs help developers by providing mechanisms that detect and correct syntax errors. This has led to research on automated *error recovery* [21, 42, 45]. We leverage error recovery techniques as a means for repairing mutated inputs during greybox fuzzing.

Error recovery techniques can be broadly categorised as *language-specific* or *language-neutral*. Language-specific approaches are hand-crafted for individual languages and thus offer higher accuracy but are costly to develop and maintain. In contrast, language-neutral techniques (e.g. panic mode recovery [42, p. 348]) can be applied to any language given a suitable grammar, but are often less precise and efficient. Their reliance on token skipping or heuristic insertion to produce a single minimal-cost repair sequence can lead to cascading errors that obscure the root cause.

CPCT$^+$ [28] is a state-of-the-art error recovery algorithm designed for LR parsers. Unlike traditional repair techniques [21, 42, 45], CPCT$^+$ computes the complete set of *minimum-cost repair sequences*—comprising insertions, deletions, and shifts—which, when applied, transform an invalid input into a syntactically valid one. CPCT$^+$ offers several key advantages:

(1) **Minimal-cost repair:** The algorithm models recovery as a shortest-path problem over the LR parser's state machine and uses a variant of Dijkstra's algorithm to compute all repair sequences with minimal cost. When multiple such repairs exist, it ranks them based on how much they reduce the likelihood of cascading errors. This helps to minimise the total cost of recovery while maximising the extent to which the original input's structure is preserved.

(2) **Generality:** CPCT+ is language-neutral and can be applied to any language for which an LR-compatible grammar is available. Since LR parsing is widely supported by parser generators such as Yacc [43], Bison [35], and Menhir [63], CPCT+ can be readily integrated into a broad range of compiler infrastructures without requiring language-specific adaptation.
(3) **Scalability:** In a large-scale evaluation on a real-world corpus of 200,000 invalid Java programs, CPCT+ successfully repaired over 98% of inputs within a 0.5-second timeout [28]. This timeout is configurable, enabling users to limit repair time per input. By skipping particularly costly cases, the algorithm maintains high throughput and remains practical at scale.

CPCT+ is designed around LR parsers [28]. This is a natural design choice, since CPCT+ implements a search-based repair algorithm that maps well onto the deterministic state machines exposed by LR parsers [45]. In particular, LR parsers provide explicit parser states, parse stacks, and shift/reduce transitions, allowing CPCT+ to identify minimum-cost repairs at a global level. This reduces cascading errors and often results in higher quality repairs. In contrast, error recovery approaches for LL parsers, such as those employed by ANTLR, typically rely on local heuristics, including token insertion/deletion and synchronisation-based recovery [42]. While these approaches are generally faster and easier to integrate into interactive tooling, they do not globally optimise repairs in the same way as CPCT+ does. Consequently, they may skip larger portions of the input [74], increasing the likelihood of cascading errors and decreasing recovery accuracy.

A key limitation of LR-based error recovery is that repairs are generated at the input position where parsing first detects an error, which is not necessarily the location of the actual syntactic issue [43]. As a result, the reported repair may sometimes appear unintuitive or differ from what a human programmer would consider the intended fix. However, this limitation is less significant in the context of fuzzing, where the primary objective is robust recovery from malformed inputs rather than producing human-intuitive repairs.

Another practical limitation concerns grammar engineering. Writing and maintaining LR-compatible grammars can be challenging, particularly because developers must manage shift/reduce and reduce/reduce conflicts [43]. Although modern LR parser generators can automatically resolve certain ambiguities through precedence and associativity declarations, grammar development remains more involved than in modern LL-based systems such as ANTLR. Furthermore, while many production compilers already provide Yacc-compatible grammar specifications [39, 65], these grammars are not always directly reusable in practice because grammar rules are often intertwined with semantic analysis logic. Consequently, parts of the grammar frequently need to be rewritten or refactored before they can be reused within a CPCT+-based pipeline.

3 Repair-driven greybox fuzzing

We now describe our repair-driven greybox fuzzing in detail (§3.1) and present an end-to-end illustrative example (§3.2).

3.1 Our Approach

Recall that our new idea, *repair-driven greybox fuzzing*, aims to address the limitations of both traditional byte-level fuzzers and grammar-based fuzzers by preserving the input diversity and scalability of byte-level fuzzers, while using repair to mitigate their tendency to generate syntactically invalid inputs that are often quickly rejected by an SUT.

Figure 2 presents an overview of repair-driven greybox fuzzing. The left-hand side shows the standard greybox fuzzing stage of selecting an input from the queue and performing byte-level mutations on it. If the input does not conform to the grammar, the error-recovery stage repairs it.

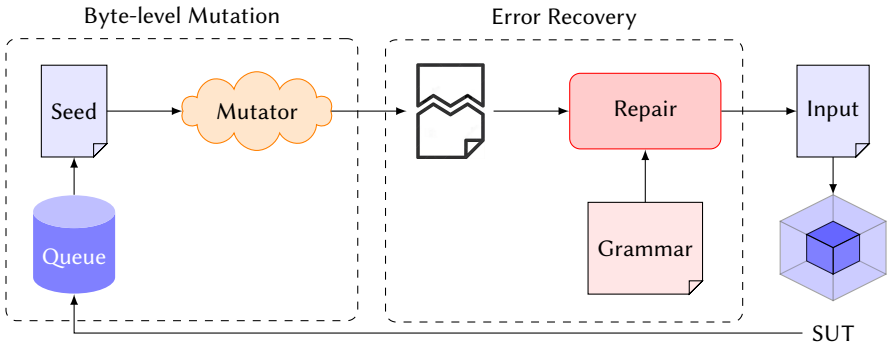


Fig. 2. Overview of repair-driven greybox fuzzing.

The input is then executed on the SUT, and if it achieves new coverage, it is added to the input queue. This design offers the following benefits:

(1) **Grammar-conforming inputs:** Restoring syntactic validity through repair avoids early-stage rejection of invalid inputs, allowing deeper logic of the SUT to be exercised.
(2) **Preserved input diversity:** By applying *minimal* repairs, the structural randomness and entropy introduced by byte-level mutations, which are crucial for effective exploration of large input spaces, are preserved.
(3) **Compatibility with seed corpora and generated inputs:** Because it extends traditional greybox fuzzing, repair-driven greybox fuzzing can operate directly on a seed corpus or import test cases generated by other fuzzers.
(4) **Selective repair:** Despite its benefits, repair can be expensive, plus there can be value in allowing some invalid inputs into the input queue. Because mutation and repair are decoupled, it is easy to adapt repair-driven greybox fuzzing to apply repair only with a certain probability, and to limit the time spent on input repair via a per-repair timeout.

We have implemented repair-driven greybox fuzzing in a new tool, RepairFuzz. RepairFuzz is built on top of AFL++ and uses CPCT$^+$ for grammar-based repair. To apply fuzzing with respect to an input format, RepairFuzz requires an LR-compatible grammar specification for the input language, as CPCT$^+$ operates over LR parse tables. This grammar is used for both validation and guiding the repair computations performed by CPCT$^+$. The repair process is implemented as a post-mutation custom function, giving us fine-grained control over when repairs are applied. In our experiments, we use a repair probability parameter to control the likelihood that a given input will be repaired. This allows us to balance the generation of valid and invalid inputs and adjust repair frequency to improve fuzzing throughput. Additionally, the repair timeout parameter defined by CPCT$^+$ can be tuned to limit the maximum time spent on each repair operation. Reducing the timeout helps discard expensive repairs and increases overall throughput, while increasing it allows more complex repairs to succeed, at the potential cost of reduced fuzzing throughput. We reduce the time budget for repair from the CPCT$^+$ default of 500 ms, geared towards the needs of code editors, to 10 ms, a more suitable time limit in the context of fuzzing. This timeout value achieves a more practical trade-off between the time spent in repair and the overall success rate of the repair process. In particular, with this timeout value, our fuzzing campaigns spend an average of 15% of their time in repair, with around 72% of the repairs completing in time (see §4).

To further enhance the speed of the repair search, and the likelihood of error recovery success, we have implemented the following changes:

1. Repair of Lexing Errors. The CPCT+ algorithm operates only on the parsing stage. Lexer errors cause the CPCT+ repair process to return early without attempting repair. To make lexer errors recoverable, we had to turn lexing errors into parsing errors, as was done in the original CPCT+ experiments [28]. We first extended the lexer with a token type that captures any input character that would otherwise be considered invalid. Secondly, since `grmtools` [29] (the library on which CPCT+ is based) mandates that every lexing token appears in the grammar, we introduced an unused dummy production that references this token. This transformation ensures that lexing errors are converted into parsing errors and therefore become amenable to repair.

2. Faster Repair Search. CPCT+ employs a variant of Dijkstra's algorithm to explore the space of possible repair sequences and returns the complete set of minimum-cost repairs. A repair sequence consists of three kinds of operations: *insert* (insert a token of a given type), *delete* (remove the token at the current input position), and *shift* (consume the current token). The total cost of a repair sequence is the sum of the individual operation costs. In the default configuration, both *insert* and *delete* have cost 1, while *shift* has cost 0.

Via a set of pilot experiments we evaluated various cost assignments and observed that setting the cost of *delete* to 0 yields a 5.6× speedup on average in repair search time. This improvement arises from the structure of the search space. CPCT+ exploration is dominated by grammar-driven branching: each *insert* operation consults the grammar to enumerate all tokens syntactically valid at the current parser state, which introduces substantial branching and significantly expands the search space. In contrast, *delete* and *shift* simply advance the input position without consulting the grammar and therefore add little or no branching. Making *delete* free strongly biases the search toward quickly skipping ill-formed parts of an input before attempting grammar-based insertions, dramatically reducing the number of explored states and yielding a substantial performance improvement.

We note that assigning zero cost to *delete* does not result in degenerate repairs that simply discard the entire input: CPCT+ initiates recovery only after a parsing error has been detected and preserves the prefix of the input that was successfully parsed before the error. Repair operations are applied at the reported error position or at subsequent input positions, and never retroactively invalidate a valid prefix. The algorithm maintains a parsing index that advances monotonically through the input. While *delete* operations may skip ill-formed tokens beyond the error point, successful recovery still requires repair progress in the form of valid shifts or insertions that advance the parser toward an accepting state. Furthermore, the structure induced by the valid input prefix constrains the space of admissible repairs on the remaining suffix. For example, an unmatched opening delimiter in the prefix can only be resolved by inserting the corresponding closing delimiter, rather than by deleting the remainder of the input. As a result, making *delete* free primarily accelerates the elimination of unparseable tokens, without significantly altering the class of repairs that the algorithm produces.

In summary, repair-driven greybox fuzzing—as implemented by RepairFuzz—combines the exploration strength of byte-level mutation-based fuzzing with the structure-awareness of grammar-based techniques, enabled by efficient and precise input repair. Our hypothesis, supported by the experimental results presented in §4.3, is that this hybrid design can be more effective when targeting syntax-aware applications (e.g., parsers, interpreters, compilers) than either grammar-based or mutation-based fuzzers alone.

3.2 Illustrative Example

The example of Figure 3 demonstrates a previously unknown PHP interpreter bug found by RepairFuzz, which the PHP developers fixed in response to our bug report [15]. The original seed input is a valid PHP program taken from the PHP test suite [25].

RepairFuzz first applies a byte-level mutation to this input, via the byte-level mutators provided by AFL++, leading to e.g. the mutated code snippet shown in the middle of Figure 3. Here, the mutation has led to the string 'pack(' being duplicated, leading to a call to unpack() that is malformed: it lacks a closing parenthesis and thus does not conform to the PHP grammar. In this form, the input would fail to deeply exercise the PHP interpreter, being rejected by the interpreter's parser. This transition from an original (valid) input to a mutated (invalid) input corresponds to one of the red "validity-breaking" arrows in Figure 1.

Given this invalid input and a grammar for PHP, RepairFuzz invokes the CPCT+ repair algorithm to generate a minimal repair: inserting a closing parenthesis near the end of input. The repaired input is shown in the bottom part of Figure 3. This transition from the mutated (invalid) input to the repaired (valid) input corresponds to one of the green "validity-restoring" arrows in Figure 1.

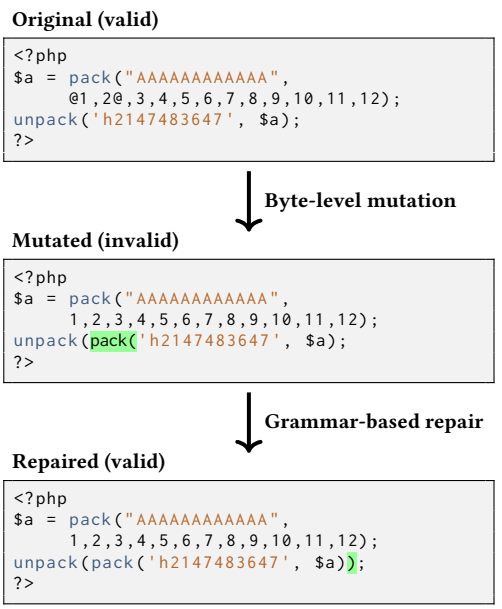


Fig. 3. A PHP input undergoing byte-level mutation that introduces a syntax error, followed by grammar-based repair to restore syntactic validity.

The repaired program triggered a bug in the PHP interpreter [15]; we found this bug during our evaluation (see §4). Specifically, the call to pack() with the format string 'h2147483647' caused an arithmetic overflow in the interpreter's C implementation (in pack.c), resulting in undefined behaviour. This only occurs when pack() is invoked with such a large repeat count—a behaviour not exercised by the original input.

To reach this bug, the mutation must replace unpack with pack while preserving the surrounding argument structure—a transformation that relies on reusing an identifier already present elsewhere in the input. Although grammar-based mutators can perform identifier substitutions, they typically select replacements from a fixed dictionary, and are unlikely to extract a specific function name like pack from one part of the input and reinsert it contextually near its original use site. Such context-sensitive reuse of internal tokens is not characteristic of standard grammar-based mutations, making this transformation highly improbable without byte-level manipulation. At the same time, pure byte-level manipulations have a very low probability of performing such a mutation *and* preserving input validity. In contrast, repair-driven greybox fuzzing restores syntactic validity using a grammar that has no knowledge of valid identifiers or reserved method names, demonstrating that structure-aware repair can recover surprising and semantically meaningful programs from syntactically corrupted inputs.

4 Evaluation

We evaluate repair-driven greybox fuzzing, as implemented by RepairFuzz, in terms of bug-finding ability and code coverage, against state-of-the-art baselines. As part of this evaluation, we investigate the influence of starting fuzzing with a *seed corpus* of valid inputs, vs. seeding fuzzing with inputs generated by a grammar-based approach. In the latter case, we explore the influence of using different types of grammars: *standard grammars*—full grammars designed for parsing—and *custom grammars*—incomplete, specialised grammars oriented towards bug-finding.

Our evaluation is guided by the following research questions:

RQ1: To what extent does repair-driven greybox fuzzing improve bug-finding ability compared with baseline fuzzing techniques?
RQ2: To what extent does repair-driven greybox fuzzing improve code coverage compared with baseline fuzzing techniques?
RQ3: To what extent does the bug-finding ability and code coverage of repair-driven greybox fuzzing and baseline fuzzing techniques change when fuzzing is performed with respect to an existing seed corpus, and using standard vs. custom grammars?

In the following we describe the SUTs, baseline tools and grammars used for our evaluation (§4.1), our experimental setup (§4.2), and our experimental results (§4.3). We then summarise our findings on the above research questions in light of these results (§4.4).

4.1 SUTs, Baseline Tools and Grammars

4.1.1 Input Formats and Systems under Test. We focus on four widely used input formats: Lua, PHP, JavaScript and Ruby, which are also the focus of state-of-the-art work on grammar-based fuzzing [2, 20, 41, 71]. These formats are sufficiently complex to require grammars, yet not so complex that grammars are ineffective.

To ensure a thorough comparison with the Nautilus baseline, we include all SUTs evaluated by Nautilus [2], plus two additional ones: LuaJIT and Ruby. These SUTs are summarised in Table 1. All are open source and implemented in C/C++, and thus benefit from fuzzing due to inherent memory-safety risks. For our experiments, we instrumented them with AddressSanitizer [67], except ChakraCore, whose build process prevented straightforward instrumentation [1, 72].

4.1.2 Baseline Tools and RepairFuzz Configurations. We compare RepairFuzz to three state-of-the-art fuzzing approaches: AFL++ [33], Grammarinator [40] and Nautilus [2].[2] We include two different configurations of RepairFuzz: RepairFuzz-Full and RepairFuzz-Partial. Details of these tools are as follows:

AFL++: A state-of-the-art mutation-based greybox fuzzer which operates on a seed corpus and applies byte-level mutations to generate new inputs. AFL++ does not utilise any grammar or input specification, which may limit the syntactic validity of generated inputs. However, this also allows it to explore the input space beyond the constraints imposed by a formal grammar. We used commit `1ffb1b6` in our experiments and adapted the custom mutation and synchronisation features to work reliably in our experimental setting.

Nautilus: A state-of-the-art grammar-based greybox fuzzer, Nautilus requires a user-provided grammar to generate and mutate inputs. A notable limitation of Nautilus is that it does not support the use of a seed corpus, relying entirely on grammar-driven generation to explore the input space. We used commit `1689565` in our experiments, and fixed a small number of crash bugs we found.

Grammarinator: A state-of-the-art grammar-based blackbox fuzzer, Grammarinator requires a user-provided grammar to generate and mutate test inputs. Grammarinator can make use of a seed corpus if available, or rely solely on grammar-driven generation to explore the input space.

[2]We also thoroughly investigated comparing against Kitten [76], another grammar-based blackbox fuzzer, but a comparison proved impossible due to two known bugs in Kitten [10, 14] that led to it crashing when applied even to Kitten's own default grammars and configuration, and a further bug uncovered by our efforts [18].

RepairFuzz-Full: Our full repair-based approach where *every* mutated input is passed through a grammar-aware repair function, ensuring that only syntactically valid inputs are sent to the SUT.

RepairFuzz-Partial: In this configuration, whether or not a mutated input should be repaired is decided at random based on a given (configurable) probability. This results in a mixture of valid and invalid inputs, allowing us to assess the trade-offs between syntactic correctness and input diversity. In our experiments we set the repair probability for RepairFuzz-Partial to 0.5.

Table 1. Details of the SUTs used for evaluation; lines of code (LOC) are computed using `cloc` [22].

Program	Input format	Version	LOC
Lua	Lua	5.5.0	36K
LuaJIT	Lua	2.1	101K
Ruby	Ruby	3.3.5	2,054K
mRuby	Ruby	3.3.0	114K
PHP	PHP	8.5.0	1,828K
ChakraCore	JavaScript	1.13.0	824K

4.1.3 Grammar Formats. Nautilus, Grammarinator, and RepairFuzz require a grammar for the input format of the SUT. Nautilus grammars are described using a Python-based domain-specific language that is specific to Nautilus, while Grammarinator grammars are specified using the ANTLR format. At the same time, RepairFuzz builds on CPCT$^+$ which requires grammars specified in the widely-used Lex/Yacc format [43]. The reason CPCT$^+$ requires an LR-compatible grammar is that these grammars generate deterministic parsing tables; this determinism is essential for reliable error recovery because it provides the precise location of syntax errors and enables the computation of minimal-cost repairs [28].

The Nautilus project already provided *custom* grammars for Lua, Ruby, PHP and JavaScript: grammars designed by the Nautilus developers with the specific aim of bug finding [56]. Rather than attempting to fully describe these languages, these custom grammars focus on particular language fragments and treat specific coding idioms (e.g. calls to key standard library functions) as if they were first-class language features. To allow RepairFuzz to work with these custom grammars we ported them to Lex/Yacc format. Similarly, we converted these grammars to ANTLR, to make them usable for Grammarinator.

To allow more general experiments using full programming language grammars, we re-used existing Lex/Yacc grammars for Lua and PHP that were available in a CPCT$^+$-related repository [69]. For Ruby and JavaScript, we extracted grammars from the only public sources we could find [30, 60] and converted these into ANTLR and Lex/Yacc formats as required by Grammarinator and CPCT$^+$. Because the Nautilus project did not include standard grammars for these languages, we also ported these standard grammars to the form required by Nautilus. Creating LR-compatible grammars for Ruby and JavaScript proved somewhat challenging: avoiding ambiguity in these syntactically complex languages, and avoiding shift/reduce and reduce/reduce conflicts, was non-trivial. However, the creation of Lex/Yacc grammars for use with RepairFuzz is a one-time effort.

To fairly compare Nautilus, Grammarinator, and RepairFuzz, we took steps to ensure each input format was described equivalently, using the Python DSL for Nautilus, ANTLR for Grammarinator, and Lex/Yacc for RepairFuzz. For each format, we generated 1,000 inputs from one grammar and attempted to parse them with the others to align the grammar descriptions. Creating a fully unambiguous LR-compatible grammar for Ruby proved difficult [68], and the ambiguities resolved automatically during parser generation can lead to valid inputs being rejected. Consequently, our Yacc grammar for Ruby covers a subset of the inputs supported by Nautilus and Grammarinator, putting RepairFuzz at a slight disadvantage in our evaluation.

4.2 Experimental Setup

Recall from §2.2 that Nautilus generates inputs from scratch and subsequently mutates them; it does *not* support working with an initial corpus of existing inputs. We thus conduct a set of experiments assessing the extent to which our RepairFuzz configurations can *amplify* the effectiveness of Nautilus, by using Nautilus as a continuous source of inputs for repair-driven greybox fuzzing, and how this compares to using Nautilus as a source of inputs for fuzzing using baseline AFL++. In addition, we examine the performance of Grammarinator relative to the Nautilus-seeded approaches. The setup for these *Nautilus-based* experiments is discussed in §4.2.1.

We also conduct a separate set of experiments assessing the effectiveness of our RepairFuzz configurations when used to fuzz an SUT using a fixed corpus of valid inputs, comparing with Grammarinator and AFL++ as baselines. Because Nautilus does not support working with an existing input corpus, it is not relevant for this set of experiments. The setup for these *corpus-based* experiments is discussed in §4.2.2.

Both sets of experiments address **RQ1** on bug-finding ability and **RQ2** on coverage, and the consideration of both standard and custom grammars in the Nautilus-based experiments, and existing seed corpora in the corpus-based experiments, addresses **RQ3**.

4.2.1 Nautilus-Based Experiments. To assess the effectiveness of RepairFuzz vs. AFL++ when Nautilus is used as a source of inputs, we have created an environment where Nautilus produces inputs from scratch, which regularly (every 15 min in our implementation) populates a pool of inputs for use by RepairFuzz-Full, RepairFuzz-Partial and AFL++. With this setup, AFL++ and the RepairFuzz configurations should exhibit at least the code coverage (and bug-finding ability) of plain Nautilus (because they import its corpus). Our interest is in whether using AFL++ on top of Nautilus improves bug-finding ability or coverage, and whether the use of repair (via RepairFuzz-Full or RepairFuzz-Partial) improves bug-finding ability or coverage compared with simply using AFL++. The additional inclusion of Grammarinator as a baseline allows us to compare the performance of the greybox tools to a state-of-the-art grammar-guided blackbox fuzzer. To address **RQ3**, we experiment with both *standard* grammars and *custom* grammars for each input format.

For each tool configuration (AFL++, Grammarinator, Nautilus, RepairFuzz-Full, RepairFuzz-Partial) and SUT (the 6 SUTs of Table 1) and grammar type (standard, custom) we conduct 10 repeat fuzzing runs, where each run is 24 hours. We report on (a) the distinct bugs found by these techniques (where we count a bug if it is detected in any repeat run), and (b) the coverage achieved by the final corpus (averaged across repeat runs).

4.2.2 Corpus-Based Experiments. In these experiments, we compare the effectiveness of RepairFuzz when starting with a fixed set of valid seed inputs with that of AFL++ and Grammarinator using the same set of valid seeds. For each input format we use as a seed corpus the test suite provided by the official reference implementation when testing SUTs that consume the input format—i.e., the reference implementation test suites for Lua [24], Ruby [27], PHP [26] and JavaScript [23]. This means that the Lua test suite serves as the seed corpus for both Lua and LuaJIT, while the Ruby test suite is used for both Ruby and mRuby. This decision is motivated by the fact that secondary implementations either lack a dedicated test suite (as in the case of LuaJIT) or provide one that is less mature and comprehensive (as with mRuby).

We then remove from each corpus those test inputs that do not parse with our grammar. Specifically, we do the filtering using a *corpus grammar* for each input format, which is derived from the standard grammar used in our Nautilus-based experiments, but contains targeted modifications to improve compatibility with the real-world test suite. These modifications include:

Table 2. Confirmed bugs discovered by each fuzzer under different input-generation modes. Each entry reports the number of independent runs (out of 10) in which the corresponding bug was triggered. GNTR stands for Grammarinator, RFuzz(P) stands for RepairFuzz-Partial, RFuzz(F) stands for RepairFuzz-Full. Bug #3 was not triggered by our default RepairFuzz configuration, but was discovered with other repair timeouts.

#	SUT	Description	Mode	GNTR	Nautilus	AFL++	RFuzz(P)	RFuzz(F)
1	Lua	Segfault in `collectargs` [3]	standard	0	0	0	0	0
			custom	0	0	0	0	0
			corpus	0	–	**1**	**1**	0
2	LuaJIT	Register allocation (ra_alloc1) [12]	standard	0	0	0	0	0
			custom	0	0	0	**1**	0
			corpus	0	–	0	0	0
3	Ruby	Heap use-after-free in `String#split` with In–Block String modification [17]	standard	0	0	0	0	0
			custom	0	0	0	0	0
			corpus	0	–	0	0	0
4	Ruby	Segfault in `compile_keyword_arg` [4]	standard	0	0	0	**1**	**1**
			custom	0	0	0	0	0
			corpus	0	–	0	**8**	**8**
5	Ruby	Segfault during string encoding [5]	standard	0	0	0	0	0
			custom	0	0	0	0	0
			corpus	0	–	0	0	**1**
6	mRuby	Heap use-after-free due to recursive `group_by` calls [9]	standard	0	0	0	0	0
			custom	0	0	0	0	**1**
			corpus	0	–	0	0	0
7	PHP	Heap buffer overflow in `zend_alloc.c` when assigning UTF–8 bytes to string [8]	standard	0	0	0	0	0
			custom	0	0	0	0	0
			corpus	0	–	**2**	**1**	0
8	PHP	Signed integer overflow [15]	standard	0	0	0	0	0
			custom	0	0	0	0	0
			corpus	0	–	0	0	**1**
9	PHP	Null byte triggers undefined behaviour in `XMLDocument` [7]	standard	0	0	0	0	0
			custom	0	0	0	0	0
			corpus	0	–	0	**1**	0
10	PHP	Stack overflow when dumping deeply chained objects [19]	standard	0	0	0	0	0
			custom	0	0	0	0	0
			corpus	0	–	0	0	**1**
11	PHP	Undefined Behavior in `date_sunrise()` [16]	standard	0	0	0	0	0
			custom	0	0	0	**1**	**1**
			corpus	0	–	**5**	**9**	**8**
12	ChakraCore	Async Function Without Suspension Causes Crash [6]	standard	0	0	0	0	0
			custom	0	0	0	0	**1**
			corpus	0	–	**1**	0	0
13	ChakraCore	Out Of Memory Abort during asm.js module linking [11]	standard	0	0	0	0	0
			custom	0	0	0	0	0
			corpus	0	–	0	**2**	**2**
Distinct bugs				**0**	**0**	**4**	**7**	**8**

Table 3. Comparison of the bug-finding effectiveness of different RepairFuzz variants under 1 ms, 10 ms, and 100 ms repair-timeout configurations. The evaluation includes only Ruby and PHP in corpus mode. Each entry reports the number of independent runs (out of 10) in which the corresponding bug was triggered.

#	RepairFuzz-Partial			RepairFuzz-Full		
	1 ms	10 ms	100 ms	1 ms	10 ms	100 ms
3	0	0	0	**1**	0	**1**
4	**6**	**8**	**6**	**9**	**8**	**6**
5	0	0	0	0	**1**	0
7	**1**	**1**	**1**	0	0	0
8	**2**	0	0	**2**	**1**	0
9	0	**1**	**1**	**1**	0	**1**
10	0	0	**1**	**1**	**1**	0
11	**8**	**9**	**8**	**9**	**8**	**10**
Distinct bugs	**4**	**4**	**5**	**6**	**5**	**4**

(1) **Comment handling**: We modified the lexer specifications to recognise and ignore code comments.[3] This was essential, as many test cases contain single-line or block comments that would otherwise cause parsing failures.
(2) **Extended syntax support**: We identified constructs present in the test suites that were not represented in the standard grammars. For example, our JavaScript grammar was initially outdated and required semicolons after each statement, whereas modern JavaScript allows semicolon omission. We updated the grammar to reflect such language evolution.
(3) **Parser ambiguity resolution**: As discussed in §4.1.3, the complex syntax of Ruby led to ambiguities in our LR-based grammar that resulted in a somewhat restrictive parser. To address this, we refined several grammar rules to guide ambiguity resolution and enable acceptance of a broader subset of the test suite.

For each tool configuration (AFL++, Grammarinator, RepairFuzz-Full, RepairFuzz-Partial) and SUT (the 6 SUTs of Table 1) we conducted 10 repeat fuzzing runs, each lasting 24 hours. Again, we report on (a) the distinct bugs found by these techniques (where we count a bug if it is detected in any repeat run), and (b) the coverage achieved by the final corpus (averaged across repeat runs).

4.2.3 Required Resources. Overall, our experiments required 20,160 hours of machine time: 14,400 hours for the Nautilus-based experiments (5 tool configurations × 6 SUTs × 2 types of grammar × 10 repeat fuzzing runs × 24 hours per fuzzing run), and 5,760 hours for the corpus-based experiments (4 tool configurations × 6 SUTs × 10 repeat fuzzing runs × 24 hours per fuzzing run). Another 10,080 machine hours was also required to perform coverage analysis.

We ran these experiments in parallel on a 96-core AWS cloud machine (x86_64) running Ubuntu 24.04, equipped with 186 GB of RAM and a 1 TB disk, resulting in an end-to-end runtime of approximately 15 days.

4.3 Experimental Results

4.3.1 Bug-Finding Ability. After completing all fuzzing experiments, we identified several bugs and crashes in the SUTs. We first deduplicated and successfully reproduced these bugs, resulting in a total of 22 unique reported bugs. 13 of these were confirmed by developers, and 10 were fixed.

Out of the remaining 9 unconfirmed bugs, we have 3 duplicates, 2 rejected, and 4 pending. We show the 13 bugs confirmed by developers in Table 2. Under the default 10 ms repair-timeout configuration, RepairFuzz-Full and RepairFuzz-Partial found 8 and 7 distinct confirmed bugs, respectively (Table 2). Across all evaluated repair-timeout configurations, they found 10 and 9 distinct confirmed bugs, respectively (Table 2 and Table 3).

Table 2 (and Table 3) also illustrate the distribution of confirmed bugs among the fuzzers. RepairFuzz-Full found 3 bugs that were not found by any other tool, RepairFuzz-Partial found 1 such bug, while each bug found by AFL++ was also found by RepairFuzz-Partial or RepairFuzz-Full. Finally, 5 bugs were found by both RepairFuzz-Full and RepairFuzz-Partial, and no other tool.

Bugs found by AFL++. AFL++ found a total of 4 bugs, all of which were also found by RepairFuzz-Partial or RepairFuzz-Full. An interesting case is bug #7, which was found exclusively by AFL++ and RepairFuzz-Partial. This PHP bug is triggered by assigning a string containing malformed UTF-8 bytes to an XML node value. This result highlights that the ability to generate invalid assignments and malformed values remains valuable for bug discovery. A significant number of bugs detected in LuaJIT by AFL++ were related to unsafe bytecode loading. In particular, AFL++ (and to a lesser extent RepairFuzz-Partial) generated several malformed Lua bytecode files that triggered numerous crashes when loaded. Upon reporting one such issue to the developers [13], we

[3] We note that while grammars designed for parsing handle comments, those designed for generation typically do not.

were informed that crashing the LuaJIT VM with crafted bytecode is trivial and expected behaviour. LuaJIT intentionally omits bytecode verification, making it unsafe to load untrusted bytecode [64]. Developers explicitly advised against reporting such cases, and we consequently classified all bugs of this type as false positives (they do not contribute to the confirmed bugs of Table 2).

Mutated (invalid)

```
repeat
  local y = select (string.gsub (..., 1, "."),
end
```

Grammar-based repair

Repaired (valid)

```
repeat
  local y = select (string.gsub (..., 1,
    "."), ...)
until false
```

Fig. 4. A Lua input generated by splicing two inputs. Original input segments are highlighted in orange and blue. The resulting invalid input is then repaired, with the newly inserted code highlighted in red.

Bugs found by RepairFuzz. Notably, the two RepairFuzz variants together found all of the 13 confirmed bugs, including 9 bugs that were beyond the reach of other fuzzers. In addition to our illustrative PHP example in Figure 3, we present in Figure 4 a second bug found by RepairFuzz. While fuzzing LuaJIT, an AFL++ splice mutation produced an input which was syntactically invalid. Using the Lua grammar, RepairFuzz identified and applied a sequence of edits to restore the input's syntactic validity. This repaired input revealed a crash in the JIT compiler component of LuaJIT [12].

Mode of discovery. Nearly all discovered bugs emerged either during the corpus-based experiments or with custom grammars in the Nautilus-based experiments, while only one bug was found with standard grammars. Although standard grammars capture all syntactic features, they lack the ability to generate inputs with meaningful semantics—an essential property for exposing deep or complex bugs. Inputs derived from standard grammars often lack any useful semantics: for example, variables may be used without being defined, or assigned values may never be referenced again. In contrast, the seed corpus and custom grammars produce inputs with both syntactic and semantic depth. For instance, many of these input programs begin with an initialisation phase and later use initialised variables. This exercises SUTs in a more meaningful way, providing the potential to find deeper bugs. As an example, the Ruby custom grammar begins by generating a sequence of variable initialisation statements, after which the remainder of the program is constructed. The grammar's variable token is then restricted to a choice among these previously initialised variable names.

4.3.2 Code Coverage. Figure 5 presents the coverage achieved by different tools on each experiment, reporting branch coverage as measured by gcov [73].

For the experiments using Nautilus equipped with standard or custom grammars, the coverage of Nautilus is shown in green, at the bottom. By design, both RepairFuzz configurations and AFL++ achieve at least the same coverage as Nautilus in these experiments, as they continuously import Nautilus's generated inputs. By comparison, the blackbox Grammarinator, shown in orange, achieves the lowest coverage, highlighting the importance of code coverage feedback.

For standard grammars, both RepairFuzz variants significantly outperform AFL++ on all SUTs, although the absolute coverage differences are relatively small for Lua and mRuby but more pronounced for the other SUTs.

For custom grammars, the results are mixed: RepairFuzz performs better on Lua, LuaJIT, and ChakraCore, while AFL++ performs better on Ruby and mRuby. We attribute this to the narrow scope of custom grammars, which typically omit large portions of syntax and language features. This limitation restricts grammar-based fuzzers, while giving byte-level fuzzers like AFL++ a significant advantage. Since AFL++ is not confined to a constrained input space, it can more freely generate simple valid inputs outside the grammar's coverage, thereby increasing code coverage.

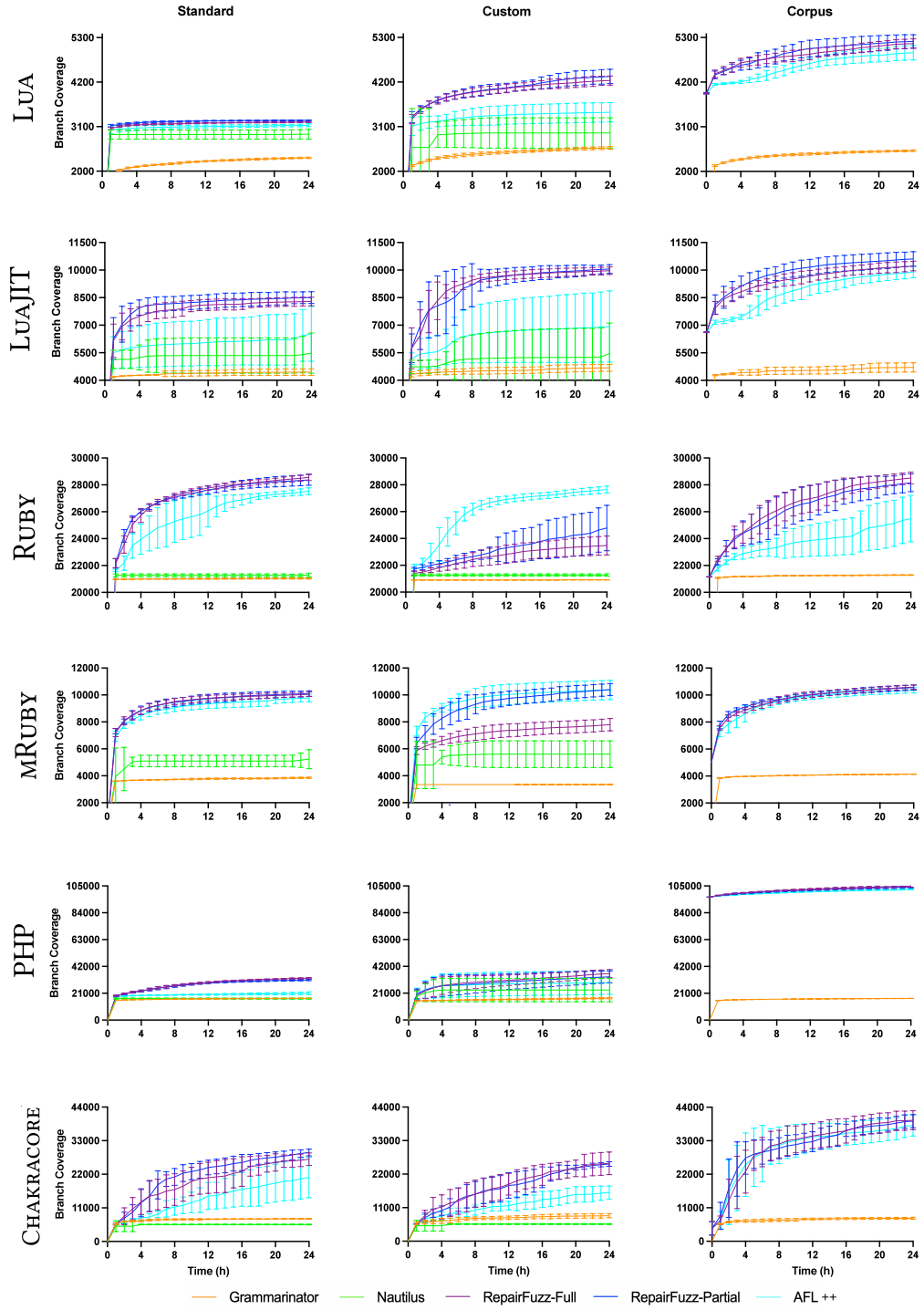


Fig. 5. Branch coverage for all SUTs across the *standard*, *custom* and *corpus* experiments.

Table 4. Probability of coverage performance superiority. Each cell reports the probability that a random run of the left-hand fuzzer outperforms a random run of the right-hand fuzzer. Highlighted cells indicate statistically significant Mann–Whitney results after Benjamini–Hochberg correction.

Grammar	SUT	RFuzz(F):RFuzz(P)	RFuzz(F):AFL++	RFuzz(F):GNTR	RFuzz(P):AFL++	RFuzz(P):GNTR
standard	lua	**0.00**	**0.99**	**1.00**	**1.00**	**1.00**
standard	luajit	**0.18**	**0.90**	**1.00**	**0.92**	**1.00**
standard	ruby	0.59	**1.00**	**1.00**	**0.95**	**1.00**
standard	mruby	0.37	**0.84**	**1.00**	**0.88**	**1.00**
standard	php	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
standard	chakracore	**0.16**	**0.78**	**1.00**	**0.96**	**1.00**
custom	lua	0.29	**1.00**	**1.00**	**1.00**	**1.00**
custom	luajit	0.34	**0.90**	**1.00**	**0.91**	**1.00**
custom	ruby	0.27	**0.00**	**1.00**	**0.06**	**1.00**
custom	mruby	**0.00**	**0.00**	**1.00**	0.62	**1.00**
custom	php	0.67	0.69	**1.00**	0.58	**1.00**
custom	chakracore	0.60	**0.96**	**1.00**	**1.00**	**1.00**
corpus	lua	0.30	**0.91**	**1.00**	**0.91**	**1.00**
corpus	luajit	**0.11**	**0.81**	**1.00**	**0.91**	**1.00**
corpus	ruby	0.64	**0.99**	**1.00**	**0.94**	**1.00**
corpus	mruby	0.55	**0.81**	**1.00**	0.74	**1.00**
corpus	php	0.75	**1.00**	**1.00**	**1.00**	**1.00**
corpus	chakracore	0.54	0.66	**1.00**	0.58	**1.00**

Table 5. Probability of coverage performance superiority across different repair-timeout configurations for corpus-based fuzzing of Ruby and PHP, computed as in Table 4.

SUT	Timeout	RFuzz(F):RFuzz(P)	RFuzz(F):AFL++	RFuzz(F):GNTR	RFuzz(P):AFL++	RFuzz(P):GNTR
Ruby	1 ms	0.55	**1.00**	**1.00**	**0.98**	**1.00**
Ruby	10 ms	0.64	**0.99**	**1.00**	**0.94**	**1.00**
Ruby	100 ms	0.30	**0.90**	**1.00**	**0.89**	**1.00**
PHP	1 ms	0.67	**1.00**	**1.00**	**1.00**	**1.00**
PHP	10 ms	0.75	**1.00**	**1.00**	**1.00**	**1.00**
PHP	100 ms	0.48	0.75	**1.00**	**0.82**	**1.00**

This rationale is further supported by the performance of RepairFuzz-Partial, whose coverage line falls between those of RepairFuzz-Full and AFL++ on both Ruby and mRuby. This suggests that the more a fuzzer deviates from restricted grammar conformance, the more code coverage it can potentially achieve.

Finally, in the corpus-based experiments, we observe that RepairFuzz and AFL++ consistently outperform Grammarinator on all SUTs. We attribute Grammarinator's comparatively weaker performance to its black-box design and the absence of code-coverage feedback.

Table 4 summarises the statistical comparison of RepairFuzz variants against baseline tools. Statistical significance is assessed using the Mann–Whitney U test with a threshold of $p < 0.05$. Both RepairFuzz-Full and RepairFuzz-Partial significantly outperform Grammarinator in all applicable configurations (100% of comparisons). Compared to AFL++, the results are more nuanced: RepairFuzz-Full achieves significantly higher coverage in 78% of cases (with 11% showing no significant difference and 11% lower performance), while RepairFuzz-Partial achieves significantly higher coverage in 72% of cases (with 22% no significant difference and 6% lower performance). The few underperformance cases are all confined to custom grammar settings. We attribute this again to the narrow scope of custom grammars, which often omit substantial portions of the target language syntax and features. This limitation constrains grammar-based fuzzers, while giving byte-level fuzzers such as AFL++ a relative advantage.

We also measured the number of unique lines of code covered by each tool (considering a line to be covered by a tool if it is covered by the tool on any repeat run within the associated experiment) and computed the intersections of their coverage. For each experiment type (standard, custom, seed corpus), we aggregate the results across benchmarks and present them as Venn diagrams

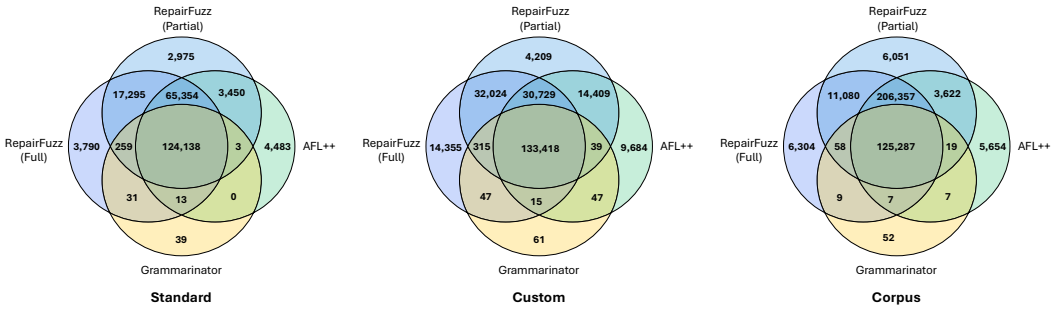


Fig. 6. Line coverage across tools for each experiment type: *standard*, *custom*, and *corpus*. Each Venn diagram shows the number of lines covered by each tool, aggregated over all six SUTs.

in Figure 6. These diagrams show that the different approaches are complementary, covering different parts of the codebases. We observe that RepairFuzz-Full performs well overall, showing a particularly clear advantage in experiments with custom grammars. After close inspection, we observed that RepairFuzz-Full achieved substantially deeper backend coverage on ChakraCore. This is likely due to the richer and more targeted custom grammar used for ChakraCore, which, when combined with RepairFuzz transformations, enabled repaired inputs to survive parsing and reach deeper and more diverse semantic execution stages. In contrast, AFL++ achieved more unique coverage on Ruby and mRuby, likely because the custom Ruby grammars were comparatively simple and constrained. Unlike grammar-based approaches, AFL++ can also generate malformed and out-of-grammar inputs, allowing it to exercise additional parser and error-handling behaviour.

While code coverage often correlates with bug discovery, several important considerations apply. In particular, RepairFuzz configurations uncover a substantial number of bugs missed by the baseline tools, even though the associated coverage gains are small. More generally, reaching buggy code is necessary to expose defects, but not sufficient, as programs may still contain bugs even when every line and branch has been executed. Even small coverage gains achieved through novel mutations or techniques can exercise rarely tested or previously unexplored parts of the program, which are more likely to harbour bugs. Consequently, our approach can uncover a significant number of bugs even when overall coverage increases are minimal. Notably, many of the bugs we discovered were not caused by exercising new language features, built-in functions, or novel semantics, but instead resulted from unexpected assignments, values, or function parameters that triggered unusual execution paths leading to overflows and crashes.

4.3.3 Repair Success Rate. Table 6 reports the repair success rates achieved by RepairFuzz. The success rate is defined as the percentage of invalid input files that are fully repaired within the 10 ms timeout. On average, 72% of invalid inputs are successfully repaired, indicating the effectiveness of the repair process, particularly given the relatively small timeout. Across different input formats and SUTs, the repair function exhibits comparable performance. Overall, RepairFuzz-Full achieves higher repair success rates than RepairFuzz-Partial. This difference stems from how repairs are applied. In RepairFuzz-Full, every mutated input is immediately subjected to repair, meaning that any invalid input is disrupted only by the most recent mutation, which generally makes it easier to repair. In contrast, RepairFuzz-Partial applies repair probabilistically, allowing invalid inputs to accumulate multiple successive mutations before a repair attempt is made. As a result, these inputs may contain compounded disruptions, making them more difficult to repair.

Table 6. Repair success rate for each SUT and grammar type. RFuzz(F) stands for RepairFuzz-Full and RFuzz(P) for RepairFuzz-Partial. Averages are computed using the geometric mean.

SUT	Standard		Custom		Corpus		Average
	RFuzz(F)	RFuzz(P)	RFuzz(F)	RFuzz(P)	RFuzz(F)	RFuzz(P)	
Lua	70	68	75	71	86	82	**75**
LuaJIT	69	62	75	70	80	77	**72**
Ruby	83	71	83	36	87	74	**69**
mRuby	83	76	78	41	87	85	**73**
PHP	59	48	89	88	64	62	**67**
ChakraCore	83	77	70	61	82	74	**74**
Average	**74**	**66**	**78**	**58**	**81**	**75**	**72**

4.3.4 Repair Timeout Configuration. All main experiments were conducted using the default repair timeout of 10 ms. To evaluate the impact of this parameter on RepairFuzz, we performed two additional sets of experiments using alternative timeout values: 1 ms and 100 ms. A 1 ms timeout introduces minimal overhead, allowing fuzzing throughput to approach that of vanilla AFL++. However, the shorter timeout increases the proportion of failed repairs, since more expensive repair attempts are abandoned early. In contrast, a 100 ms timeout may allow more repair attempts to complete successfully, but introduces higher overhead during fuzzing. Due to the already substantial computational cost of the main evaluation, we performed a reduced version of the main experiment, evaluating the two largest SUTs, Ruby and PHP, in corpus mode, where bug discovery was most likely, performing 10 independent runs per configuration. Table 3 shows the number of bugs found under the three timeout configurations. RepairFuzz-Full with a 1 ms timeout discovered the highest number of distinct bugs. Table 5 presents the corresponding coverage results, where smaller timeout values also achieved slightly higher coverage overall. Together, these results suggest that shorter repair timeouts provide a better balance between repair effectiveness and fuzzing throughput, indicating diminishing returns from spending too much time on individual repair attempts.

4.4 Answers to Research Questions

We now return to the research questions stated at the start of this section:

RQ1: *To what extent does repair-driven greybox fuzzing improve bug-finding ability compared with baseline fuzzing techniques?* As summarised by Table 2, across our bug-finding experiments, RepairFuzz found 9 unknown (and now confirmed) bugs missed by other fuzzers. Among these, 3 were unique to RepairFuzz-Full, and 1 was unique to RepairFuzz-Partial. In contrast, Grammarinator and Nautilus found no bugs, while AFL++ found no unique bugs. The bugs uniquely found by RepairFuzz, including a bug in the LuaJIT register allocator and a signed integer overflow in the PHP interpreter (see Table 2), demonstrate that grammar-aware repair improves bug-finding ability.

RQ2: *To what extent does repair-driven greybox fuzzing improve code coverage compared with baseline fuzzing techniques?* Both RepairFuzz variants significantly outperform Grammarinator in all applicable configurations, highlighting the value of code-coverage feedback when fuzzing highly structured inputs. Compared with AFL++, RepairFuzz achieves significantly higher branch coverage in all standard-grammar configurations and in most experiments using real-world seed corpora. However, AFL++ can perform better with restrictive custom grammars, particularly for Ruby and mRuby, because it is not constrained to grammar-conforming inputs. Gains over Nautilus are expected because RepairFuzz and AFL++ continuously import its generated inputs. The complementary coverage results in Figure 6 further show that RepairFuzz exercises code

regions missed by AFL++ and Grammarinator. Overall, RepairFuzz improves coverage in most configurations and provides substantial complementary coverage.

RQ3: *To what extent does the bug-finding ability and code coverage of repair-driven greybox fuzzing and baseline fuzzing techniques change when fuzzing is performed with respect to an existing seed corpus, and using standard vs. custom grammars?* Seed corpora, and to a lesser extent custom grammars, were effective in the discovery of bugs, while in contrast only one bug was found by fuzzing using standard grammars. Seed corpora derived from test suites tend to contain edge cases, and custom grammars are often designed to expose specific behaviours. In these settings, RepairFuzz found more total and unique bugs than the other tools. As discussed in the answer to RQ2 above, RepairFuzz slightly outperforms AFL++ with respect to coverage in experiments that use real-world seed corpora or inputs generated from standard grammars, while AFL++ can perform better when custom grammars are used. On the other hand, blackbox Grammarinator performs significantly worse than all other greybox tools, regardless of the grammar used. Our analysis of complementary code coverage (Figure 6) highlights the advantage of using RepairFuzz in the context of specialised custom grammars, and the overall potential value of fuzzing using the combination of AFL++, Grammarinator, RepairFuzz-Full and RepairFuzz-Partial.

5 Related Work

Mutation-based approaches. Traditional greybox fuzzers explore the input space by applying byte-level mutations to existing test corpora. The most widely used tool in this category is AFL++ [33], a successor of AFL [78], whose mutation strategies—such as bit flips, block insertions, and deletions—have proven highly effective in practice. Taint-based mutation techniques [34, 57, 66] further refine this approach by identifying dependencies between input bytes and program behaviour, allowing targeted mutations in relevant input regions. These approaches have achieved notable success when fuzzing systems that accept loosely structured inputs, such as network packets or binary formats. However, they tend to perform poorly when applied to programs that expect highly structured inputs, such as programming languages or document formats.

Generation-based approaches. These methods rely on explicit input specifications to generate syntactically valid test cases. Tools such as Csmith [77], CsmithEdge [31], GrayC [32], and YARPGen [50] focus on generating well-formed C programs, free from undefined behaviour, and have been highly effective for testing C compilers. However, these tools are limited to the C language. Other tools like LangFuzz [41] and IFuzzer [75] use context-free grammars to mutate existing input corpora. These techniques have demonstrated strong results, particularly on JavaScript engines, but generalisation to other domains has been limited. We have already discussed Grammarinator [40] and Nautilus [2]. Unlike RepairFuzz, they focus exclusively on high-level, grammar-driven mutations while discarding low-level mutations, such as bit-level edits that can span multiple tokens. This limits input diversity and reduces the structural randomness that is crucial for exercising deeper or unexpected program behaviours. Gmutator [20] generates edge-case inputs through grammar mutations to expose mismatches between implementations and specifications. It primarily targets parsers, and is not designed for general-purpose fuzzing. Gramatron [71] converts CFGs into grammar automata, enabling highly randomised grammar-aware input generation. However, automaton-based representations do not scale well to highly recursive or rich programming languages. lFuzzer [55] and pFuzzer [54] are complementary to our approach, synthesising mostly simple valid token sequences to maximise grammar-rule and parser-code coverage. While useful for parser comprehension, debugging, and seed generation, they primarily focus on parser-level exploration rather than the broader semantic exploration required for testing real-world systems.

Diversity-oriented generators. Zest [61] generates highly diverse inputs by applying byte-level mutations to parameter sequences, which are then deterministically mapped to concrete structured inputs. Zest requires a hand-written generator to be constructed and does not naturally leverage a seed corpus. Additionally, Zest is prone to the *havoc effect* [48], where small mutations to the parameter sequence can induce disproportionately large and destructive changes in the generated input, making controlled exploration of the input space difficult. BeDivFuzz [59] extends Zest by reducing the havoc effect and achieving a more even exploration of program behaviours. However, it requires the random choices in its parametric generator to be explicitly separated into structural and value choices [48]. Furthermore, optimising for controlled exploration and balanced coverage of the input space can reduce the likelihood of discovering rare edge cases that are often exposed through highly uncommon inputs [59]. In contrast, RepairFuzz operates directly on concrete inputs and can leverage a seed corpus. By performing mutations and minimum-cost repairs directly on concrete syntax, RepairFuzz avoids the havoc effect, while still enabling broad exploration through its random validity-breaking mutations. Soremekun et al. [70] propose a probabilistic grammar-based generator that learns the probabilities of grammar productions from a corpus, then inverts them to generate uncommon inputs. While this approach can produce highly unusual syntactic structures, the generated inputs are often semantically meaningless.

Input-repair mechanisms. Techniques for repairing malformed inputs have been studied for decades. Early strategies, such as panic mode recovery [21, 42, 45], were simple and fast but often imprecise. CPCT$^+$ [28], a more recent algorithm, offers more accurate and scalable error recovery for LR parsers by computing minimal-cost repair sequences. We adopt CPCT$^+$ in our work for its demonstrated precision and efficiency. Related work in program repair [47, 51, 52, 58] focuses on generating patches that fix programs failing their test suites. These techniques do not rely on formal specifications but are often too computationally expensive for use in fuzzing. Docovery [46] is another approach that attempts to repair broken inputs by using symbolic execution to explore paths that avoid failure points. While these techniques do not require input grammars, they lack the scalability and speed needed for high-throughput fuzzing. Techniques like DDMax [44] rely on delta debugging with repeated SUT interactions, making them too slow for tight fuzzing loops and limited to deletion-based edits that often oversimplify inputs. More expressive methods such as ϵREPAIR [53] support richer edits but incur higher overhead due to larger search spaces and repeated parsing, and require fine-grained parser feedback not commonly available in real systems.

6 Conclusion

We have introduced repair-driven greybox fuzzing, a novel approach that combines standard greybox fuzzing based on byte-level mutations with a grammar-based repair stage, maintaining syntactic validity while preserving the diversity of byte-level fuzzing. Experiments using our RepairFuzz implementation show that RepairFuzz outperforms and complements AFL++, a standard greybox fuzzer, as well as Grammarinator and Nautilus, which are grammar-based blackbox and greybox fuzzers, respectively, on SUTs that consume highly-structured programming language input formats, discovering 13 confirmed bugs including 9 bugs not found by the other approaches.

7 Data Availability

Our artefact can be accessed at https://doi.org/10.5281/zenodo.16090736.

Acknowledgements

We thank Laurence Tratt for his assistance with CPCT+, Joseph Ryan for his feedback, and James Lee-Jones for reviewing the artefact. This project was funded by the UK Engineering and Physical

Sciences Research Council (EPSRC) through a PhD studentship and grant EP/R006865/1, by Google through its PhD Fellowship programme, and by the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 819141).

References

[1] AFLplusplus. 2026. *Notes for using ASAN with afl-fuzz.* https://aflplus.plus/docs/notes_for_asan/

[2] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *Proc. of the 26th Network and Distributed System Security Symposium (NDSS'19)* (San Diego, CA, USA). https://doi.org/10.14722/ndss.2019.23412

[3] Bachir Bendrissou. 2024. *Segmentation fault in collectargs.* https://groups.google.com/g/lua-l/c/2_2oD_tXBYs

[4] Bachir Bendrissou. 2024. *Segmentation Fault in compile_keyword_arg.* https://bugs.ruby-lang.org/issues/20906

[5] Bachir Bendrissou. 2024. *Segmentation fault when string encoding.* https://bugs.ruby-lang.org/issues/20903

[6] Bachir Bendrissou. 2025. *Async Function Without Suspension Causes Crash.* https://github.com/chakra-core/ChakraCore/issues/7034

[7] Bachir Bendrissou. 2025. *DOM\XMLDocument::createComment() triggers undefined behavior with null byte.* https://github.com/php/php-src/issues/18979

[8] Bachir Bendrissou. 2025. *Heap-buffer-overflow in zend_alloc.c when assigning string with UTF-8 bytes.* https://github.com/php/php-src/issues/18597

[9] Bachir Bendrissou. 2025. *Heap-Use-After-Free due to Recursive group_by Calls.* https://github.com/mruby/mruby/issues/6467

[10] Bachir Bendrissou. 2025. *NullPointerException in FastStringBuilder.append.* https://github.com/uw-pluverse/perses/issues/35

[11] Bachir Bendrissou. 2025. *Out Of Memory Abort during asm.js module linking.* https://github.com/chakra-core/ChakraCore/issues/7052

[12] Bachir Bendrissou. 2025. *Possible issue during register allocation (ra_alloc1).* https://www.freelists.org/post/luajit/Possible-issue-during-register-allocation-ra-alloc1

[13] Bachir Bendrissou. 2025. *Potential Stack Buffer Overflow in LuaJIT Bytecode Loader.* https://www.freelists.org/post/luajit/Potential-Stack-Buffer-Overflow-in-LuaJIT-Bytecode-Loader

[14] Bachir Bendrissou. 2025. *Problem: the crashDetectorClassName is invalid.* https://github.com/uw-pluverse/perses/issues/36

[15] Bachir Bendrissou. 2025. *Signed Integer Overflow in pack().* https://github.com/php/php-src/issues/18976

[16] Bachir Bendrissou. 2025. *Undefined Behavior in date_sunrise().* https://github.com/php/php-src/issues/18978

[17] Bachir Bendrissou. 2025. *Use-After-Free in String#split with In−Block String modification.* https://bugs.ruby-lang.org/issues/21380

[18] Bachir Bendrissou. 2026. *Seedless fuzzing option not working.* https://github.com/uw-pluverse/perses/issues/40

[19] Bachir Bendrissou. 2026. *Stack overflow when dumping deeply chained objects.* https://github.com/php/php-src/issues/20840

[20] Bachir Bendrissou, Cristian Cadar, and Alastair Donaldson. 2025. Grammar Mutation for Testing Input Parsers. *ACM Transactions on Software Engineering Methodology (TOSEM)* 34, 4 (2025), 116:1–21.

[21] Rafael Corchuelo, José A. Pérez, Antonio Ruiz, and Miguel Toro. 2002. Repairing syntax errors in LR parsers. *ACM Trans. Program. Lang. Syst.* 24, 6 (Nov. 2002), 698–710. https://doi.org/10.1145/586088.586092

[22] Albert Danial. 2025. CLOC: Count Lines of Code. https://github.com/AlDanial/cloc.

[23] Chakracore developers. 2025. *Chakracore Test Suite.* https://github.com/chakra-core/ChakraCore/tree/master/test

[24] Lua developers. 2025. *Lua Test Suite.* https://github.com/lua/lua/tree/master/testes

[25] PHP developers. 2023. *PHP Test Case.* https://github.com/php/php-src/blob/master/ext/standard/tests/strings/gh10940.phpt

[26] PHP developers. 2025. *PHP Test Suite.* https://github.com/php/php-src/tree/master/ext/standard/tests

[27] Ruby developers. 2025. *Ruby Test Suite.* https://github.com/ruby/ruby/tree/master/test

[28] Lukas Diekmann and Laurence Tratt. 2020. Don't Panic! Better, Fewer, Syntax Errors for LR Parsers. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 6:1–6:32. https://doi.org/10.4230/LIPIcs.ECOOP.2020.6

[29] Lukas Diekmann and Laurence Tratt. 2025. *grmtools.* GitHub. https://softdevteam.github.io/grmtools/latest_release/book/

[30] Brendan Eich and C. Rand McKinney. 1996. *JavaScript LL(1) Grammar.* https://hepunx.rl.ac.uk/~adye/jsspec11/llr.htm

[31] Karine Even-Mendoza, Cristian Cadar, and Alastair Donaldson. 2022. CsmithEdge: More Effective Compiler Testing by Handling Undefined Behaviour Less Conservatively. *Empirical Software Engineering (EMSE)* 27, 6 (Nov. 2022), 35 pages. https://doi.org/10.1007/s10664-022-10146-1

[32] Karine Even-Mendoza, Arindam Sharma, Cristian Cadar, and Alastair F. Donaldson. 2023. GrayC: Greybox Fuzzing of Compilers and Analysers for C. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'23)* (Seattle, WA, USA).

[33] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *Proc. of the 14th USENIX Workshop on Offensive Technologies (WOOT'20)* (Online).

[34] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based Directed Whitebox Fuzzing. In *Proc. of the 31st International Conference on Software Engineering (ICSE'09)* (Vancouver, BC, Canada).

[35] GNU. 2021. *Bison 3.8.1.* https://www.gnu.org/software/bison/manual/bison.html

[36] Google. 2024. OSS-Fuzz Trophies. https://github.com/google/oss-fuzz?tab=readme-ov-file#trophies.

[37] Alex Groce, Rijnard van Tonder, Goutamkumar Tulajappa Kalburgi, and Claire Le Goues. 2022. Making No-Fuss Compiler Fuzzing Effective. In *Proc. of the 31st International Conference on Compiler Construction (CC'22)* (Seoul, Korea). https://doi.org/10.1145/3497776.3517765

[38] Tao Guo, Puhan Zhang, Xin Wang, and Qiang Wei. 2013. GramFuzz: Fuzzing testing of web browsers based on grammar analysis and structural mutation. In *Proc. of the International Conference on Informatics and Applications (ICIA'13)* (Lodz, Poland). https://doi.org/10.1109/ICoIA.2013.6650258

[39] Andi Gutmans, Zeev Suraski, and Nikita Popov. 2026. The PHP Parser. https://github.com/php/php-src/blob/master/Zend/zend_language_parser.y. Accessed 23 July 2026.

[40] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. Grammarinator: A Grammar-Based Open Source Fuzzer. In *Proc. of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST'18)* (Lake Buena Vista, FL, USA). https://doi.org/10.1145/3278186.3278193

[41] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proc. of the 21st USENIX Security Symposium (USENIX Security'12)* (Bellevue, WA, USA).

[42] Allen Holub. 1990. *Compiler design in C.* Prentice-Hall, Inc., USA.

[43] Stephen C Johnson. 1975. *Yacc : Yet Another Compiler-Compiler.* Technical Report Comp. Sci. 32, Bell Labs.

[44] Lukas Kirschner, Ezekiel Soremekun, and Andreas Zeller. 2020. Debugging inputs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 75–86. https://doi.org/10.1145/3377811.3380329

[45] Donald E. Knuth. 1965. On the translation of languages from left to right. *Information and Control* 8, 6 (1965), 607–639. https://doi.org/10.1016/S0019-9958(65)90426-2

[46] Tomasz Kuchta, Cristian Cadar, Miguel Castro, and Manuel Costa. 2014. Docovery: toward generic automatic document recovery. In *Proc. of the 29th IEEE International Conference on Automated Software Engineering (ASE'14)* (Västerås, Sweden).

[47] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering (TSE)* 38, 1 (2012), 54–72.

[48] Ao Li, Madonna Huang, Vasudev Vikram, Caroline Lemieux, and Rohan Padhye. 2025. The Havoc Paradox in Generator-Based Fuzzing. *ACM Trans. Softw. Eng. Methodol.* 35, 1, Article 29 (Dec. 2025), 26 pages. https://doi.org/10.1145/3742894

[49] LibFuzzer 2022. LibFuzzer website. http://llvm.org/docs/LibFuzzer.html. Last accessed 2025-01-21.

[50] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. In *Proc. of the ACM on Programming Languages (OOPSLA'20)* (Chicago, IL, USA). https://doi.org/10.1145/3428264

[51] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'15)* (Bergamo, Italy).

[52] Fan Long and Martin Rinard. 2016. An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems. In *Proc. of the 38th International Conference on Software Engineering (ICSE'16)* (Austin, TX, USA).

[53] Zijian Luo, Lukas Kirschner, Ezekiel Soremekun, and Rahul Gopinath. 2025. Automatic Data Repair without Format Specifications. In *2025 IEEE 36th International Symposium on Software Reliability Engineering (ISSRE)*. 418–429. https://doi.org/10.1109/ISSRE66568.2025.00049

[54] Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. 2019. Parser-Directed Fuzzing. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI'19)* (Phoenix, AZ, USA). https://doi.org/10.1145/3314221.3314651

[55] Björn Mathis, Rahul Gopinath, and Andreas Zeller. 2020. Learning input tokens for effective fuzzing. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 27–37. https://doi.org/10.1145/3395363.3397348

[56] nautilus-fuzz. 2024. *Nautilus Custom Grammars.* https://github.com/nautilus-fuzz/nautilus/tree/mit-main/grammars

[57] James Newsome and Dawn Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proc. of the 12th Network and Distributed System Security Symposium (NDSS'05)* (San Diego, CA, USA).
[58] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proc. of the 35th International Conference on Software Engineering (ICSE'13)* (San Francisco, CA, USA).
[59] Hoang Lam Nguyen and Lars Grunske. 2022. BeDivFuzz: integrating behavioral diversity into generator-based fuzzing. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 249–261. https://doi.org/10.1145/3510003.3510182
[60] University of Buffalo. 2023. *BNF Syntax of Ruby.* https://cse.buffalo.edu/~regan/cse305/RubyBNF.pdf
[61] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'19)* (Beijing, China). https://doi.org/10.1145/3293882.3330576
[62] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. 2021. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering (TSE)* 47, 9 (2021), 1980–1997. https://doi.org/10.1109/TSE.2019.2941681
[63] François Pottier and Yann Régis-Gianas. 2024. *Menhir Reference Manual.* https://pauillac.inria.fr/~fpottier/menhir/manual.html
[64] The LuaJIT Project. 2025. *Can Lua code be safely sandboxed?* https://luajit.org/faq.html#sandbox
[65] Ruby Development Team. 2026. The Ruby Parser. https://github.com/ruby/ruby/blob/master/parse.y. Accessed 23 July 2026.
[66] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask). In *Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'10)* (Oakland, CA, USA).
[67] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proc. of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)* (Boston, MA, USA).
[68] Nick Sieger. 2006. *Visualization of Ruby's Grammar.* http://blog.nicksieger.com/articles/2006/10/27/visualization-of-rubys-grammar/
[69] softdevteam. 2024. *A collection of language grammars for 'lrpar'.* https://github.com/softdevteam/grammars
[70] Ezekiel Soremekun, Esteban Pavese, Nikolas Havrikov, Lars Grunske, and Andreas Zeller. 2022. Inputs From Hell: Learning Input Distributions for Grammar-Based Test Generation. *IEEE Transactions on Software Engineering* 48, 4 (2022), 1138–1153. https://doi.org/10.1109/TSE.2020.3013716
[71] Prashast Srivastava and Mathias Payer. 2021. Gramatron: Effective grammar-aware fuzzing. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'21)* (Online). https://doi.org/10.1145/3460319.3464814
[72] Symeon. 2018. *Cannot instrument libChakraCore.so with AddressSanitizer.* https://github.com/chakra-core/ChakraCore/issues/5239
[73] GCC team. 2025. gcov – A Test Coverage Program. gcc.gnu.org/onlinedocs/gcc/Gcov.html.
[74] D. A. Turner. 1977. Error Diagnosis and Recovery in One Pass Compilers. *Inform. Process. Lett.* 6, 4 (1977), 113–115. https://doi.org/10.1016/0020-0190(77)90023-0
[75] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. IFuzzer: An Evolutionary Interpreter Fuzzer Using Genetic Programming. In *Proc. of the European Symposium on Research in Computer Security (ESORICS'16)*. https://doi.org/10.1007/978-3-319-45744-4_29
[76] Yuanmin Xie, Zhenyang Xu, Yongqiang Tian, Min Zhou, Xintong Zhou, and Chengnian Sun. 2025. Kitten: A Simple Yet Effective Baseline for Evaluating LLM-Based Compiler Testing Techniques. In *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Clarion Hotel Trondheim, Trondheim, Norway) *(ISSTA Companion '25)*. Association for Computing Machinery, New York, NY, USA, 21–25. https://doi.org/10.1145/3713081.3731731
[77] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI'11)* (San Jose, CA, USA). https://doi.org/10.1145/1993498.1993532
[78] Michal Zalewski. 2025. Technical "whitepaper" for afl-fuzz. http://lcamtuf.coredump.cx/afl/technical_details.txt. Last accessed 2025-01-21.

Received 2026-01-29; accepted 2026-06-25