Gramatron: Effective Grammar-Aware Fuzzing

Prashast Srivastava
Purdue University
United States of America

Mathias Payer
EPFL
Switzerland

ABSTRACT

Fuzzers aware of the input grammar can explore deeper program states using *grammar-aware* mutations. Existing grammar-aware fuzzers are ineffective at synthesizing complex bug triggers due to: (i) grammars introducing a sampling bias during input generation due to their structure, and (ii) the current mutation operators for parse trees performing localized small-scale changes.

Gramatron uses *grammar automatons* in conjunction with *aggressive* mutation operators to synthesize complex bug triggers faster. We build grammar automatons to address the sampling bias. It restructures the grammar to allow for unbiased sampling from the input state space. We redesign grammar-aware mutation operators to be more aggressive, i.e., perform large-scale changes.

Gramatron can consistently generate complex bug triggers in an efficient manner as compared to using conventional grammars with parse trees. Inputs generated from scratch by Gramatron have higher diversity as they achieve up to 24.2% more coverage relative to existing fuzzers. Gramatron makes input generation 98% faster and the input representations are 24% smaller. Our redesigned mutation operators are 6.4× more aggressive while still being 68% faster at performing these mutations. We evaluate Gramatron across three interpreters with 10 known bugs consisting of three complex bug triggers and seven simple bug triggers against two Nautilus variants. Gramatron finds all the complex bug triggers reliably and faster. For the simple bug triggers, Gramatron outperforms Nautilus four out of seven times. To demonstrate Gramatron's effectiveness in the wild, we deployed Gramatron on three popular interpreters for a 10-day fuzzing campaign where it discovered 10 new vulnerabilities.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Security and privacy** → **Software and application security**.

KEYWORDS

Fuzzing, grammar-aware, dynamic software analysis

ACM Reference Format:
Prashast Srivastava and Mathias Payer. 2021. Gramatron: Effective Grammar-Aware Fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3460319.3464814

ISSTA '21, July 11–17, 2021, Virtual, Denmark
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8459-9/21/07.
https://doi.org/10.1145/3460319.3464814

1 INTRODUCTION

Language interpreters like PHP, JavaScript (JS), or Ruby accept input whose structure is defined as per a grammar. These interpreters form the building blocks for complex application frameworks but are themselves highly vulnerable to exploitation with 98 reported bugs between January 2018 and January 2021 [16, 29, 30]. Hence, they are lucrative targets for adversaries. Testing these building blocks, i.e., the interpreters, is essential to ensure the safety of software running on top of them such as web applications.

Fuzzing is an effective software security testing methodology. However, current fuzzing approaches are ineffective at performing *deep* testing of interpreters. A majority of the test inputs generated by grammar-unaware fuzzers [41, 50] are rejected by the interpreter during parsing. Interpreters reject all inputs that violate the grammar, and when fuzzers are unaware of the accepted grammar, they will mostly create syntactically incorrect input. For example, a common mutation operator is flipping random input bits. A fuzzer unaware of the grammar may flip bits in input keywords, creating invalid mutants that are rejected by the parser. The interpreter components past the parsing stage corresponding to semantic analysis remain untested if fuzzers are grammar-unaware.

Fuzzing the semantic analysis components requires generating syntactically valid inputs. Existing grammar-aware fuzzers [15, 19, 43, 46] use: (i) a context-free grammar (CFG) to generate test inputs, and (ii) parse trees to represent the syntactic structure of the input. The fuzzer mutates the parse trees using the grammar to generate syntactically valid inputs for testing—grammar-aware mutations. We observe a twofold problem with the current methodology for grammar-aware fuzzing:

- **Biased sampling:** Fuzzers when using existing grammars perform biased sampling from the input state space. This bias occurs due to how the production rules in the CFG are laid out for generating inputs. This bias can make it harder for the fuzzer to generate complex bug triggers which require chaining multiple parts of the input state space.
- **Small-scale mutations:** Grammar-aware fuzzers employ parse trees for input generation and mutation. Existing mutation operators for parse trees perform localized small-scale changes. This slows down the fuzzer while trying to discover bugs with complex bug triggers if it wastes its time fuzzing grammar parts not relevant towards triggering the bug.

We propose a two-fold solution to the above problem: (i) Restructuring the production rules in the grammar to eliminate the sampling bias, and (ii) Redesigning mutation operators to perform larger-scale changes. However, implementing these solutions on top of inputs represented as parse trees imposes a performance overhead. This overhead arises from a fuzzer having to maintain metadata in the form of the input derivation structure for each input as its being generated or mutated. To remove this overhead and implement our solution in a performance-optimal way, we convert the input grammar to a finite state automaton (FSA), which we refer to as *grammar automatons*.

Grammar automatons restructure the grammar to eliminate the sampling bias. Furthermore, grammar automatons allow performing *aggressive* mutations (more terminals may be changed) efficiently. Aggressive mutations ensure that the fuzzer does not get stuck performing localized search of grammar parts that are not relevant towards triggering bugs.

We present Gramatron, a proof-of-concept for our claim that grammar automatons are an effective solution for performing grammar-aware fuzzing. It represents inputs as automaton walks and uses grammar-aware mutation operators that have been redesigned for fast and aggressive mutations.

We evaluated using performance microbenchmarks if: (i) Gramatron resolves the sampling bias in a performant manner, and (ii) redesigned mutation operators perform aggressive changes efficiently. Gramatron generates higher diversity inputs as they achieve up to 24.2% more coverage. This indicates that our generated inputs cover richer semantics of the input grammar. Additionally, on average, inputs represented as automaton walks are 24% smaller and input generation is 98% faster. Our redesigned mutation operators are on an average 68% faster and 6.4× more aggressive in their mutations.

To showcase that using conventional grammars coupled with small-scale mutations can be a problem while trying to discover complex bug triggers, we compared Gramatron against two variants (detailed in § 6) of the current state-of-the-art grammar-aware fuzzer Nautilus [2]. We evaluate all systems against a set of three interpreters with 10 known bugs consisting of three complex bug triggers and seven simple bug triggers. For the three complex bug triggers, Gramatron outperforms one variant on all of them and the other variant on two of them. Gramatron finds four out of the seven simple bug trigger faster than one variant and three out of the seven than the other variant. Furthermore, to prove its effectiveness in the wild, we deployed Gramatron on three popular interpreters for a 10-day fuzzing campaign. It discovered 10 new vulnerabilities. Additionally, we discuss a bug found as a case-study to show how Gramatron is effective at generating complex bug triggers.

The main contributions of Gramatron are:

- We leverage grammar automatons which restructures grammars to allow generating highly diverse inputs reliably as an effective methodology to synthesize complex bug triggers
- We redesign and optimize mutation operators for grammar automatons to enable them to do aggressive mutations for faster discovery of bugs with complex triggers.
- As a proof-of-concept, we build and evaluate Gramatron[1], an efficient grammar-aware fuzzer.
- Gramatron discovered 10 new vulnerabilities over a 10-day fuzzing campaign, so far one CVE has been assigned.

2 BACKGROUND

Gramatron transforms grammars to an automaton to resolve the sampling bias and enable performing aggressive mutations efficiently. This section introduces the necessary background for automaton generation. We also introduced the mutation operators which are customized for this new representation.

2.1 Context-Free Grammars

Language interpreters define the input format accepted by them using a Context-Free Grammar (CFG). Formally, a Context-free Grammar (CFG) [48] is defined as: $CFG = (T, N, R, S)$. T is a finite set of *terminals* (characters in the generated string). N is a finite set of nonterminal symbols (placeholders for patterns of T that can be generated by N). R is a finite set of rules for substituting N with T. The rules are of the form $A \longrightarrow a$ where A is *always* a nonterminal symbol. However, a can be a permutation of symbols from both N and T. S is the starting nonterminal symbol from which all strings belonging to the grammar are derived.

In order to eliminate sampling bias, Gramatron restructures the grammar using the normal form by enforcing certain rules. We focus on two normal forms, used together by Gramatron to create grammar automatons from a CFG: *Chomsky Normal Form* (CNF) [6] and Greibach Normal Form (GNF) [14].

Both CNF and GNF are similar to CFG except they have constraints placed on their rules (R). For CNF, each nonterminal can either generate a single terminal or two nonterminals. For GNF, each nonterminal can either generate a terminal or a terminal followed by any number of nonterminals.

2.2 Automaton Classes

We give formal definitions of different automaton classes that Gramatron employs to create grammar automatons. Additionally, we detail the necessary background for the theoretical challenges involved in creating grammar automatons.

Finite State Automatons. Gramatron creates grammar automatons for performing fuzzing using finite state automatons (FSA). This allows Gramatron to create lightweight input representations that enable fast and aggressive mutations. Formally, a FSA is defined as $M = (Q, \Sigma, \delta, q_0, F)$ [40]. Here, Q is a finite set of states. Σ is the finite set of terminals. δ defines the set of all transitions over the automaton states. q_0 is the start state in the automaton corresponding to the start symbol of the grammar. F is the accepting state in the automaton. A set of transitions from the start state to the final state describe an input string that belongs to the grammar.

Pushdown Automatons. Gramatron leverages pushdown automatons (PDA) to create grammar automatons. A PDA is a language recognizer for a CFG. In the context of Gramatron, we will consider the 1-state PDA that accepts inputs by an empty stack. Formally, a PDA is defined as $M = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$ [20]. Here, Q is a finite set of states. Σ is the finite set of terminals. Γ is a set of symbols that can be pushed and popped from the stack (referred to as *stack symbols*). Z is the start symbol pushed to the stack. q_0 is the start state and F is the accepting automaton state. δ is the transition function that governs the automaton behavior. It takes as argument $\delta(q, a, X)$ where, $q \in Q$, $a \in \Sigma$, and $X \in \Gamma$. The output of δ is a

[1] Source code of our framework is available at https://github.com/HexHive/Gramatron.

finite set of pairs (p, γ) where p is a new state and γ is the string of symbols that replace X at the top of the stack by being pushed in reverse. If γ is empty then only X is popped from the stack.

2.3 Grammar-Aware Mutation Operators

Grammar-aware mutation operators mutate inputs while maintaining syntactic validity. They take in as input a grammar along with a syntactically valid test case to create mutants. There are three mutation operators which have proven successful in finding deep bugs [2]: *random mutation* (pick a random non-leaf nonterminal node and create a new subtree), *random recursive* (find recursive production rules and unroll up to n times), and *splice* (combine two inputs while preserving syntax). We implemented them in Gramatron and tailored them for fast and aggressive mutant generation.

3 GRAMATRON OVERVIEW

Gramatron is a coverage-guided, grammar-aware generational fuzzer. As input, Gramatron takes a CFG accepted by the target. Gramatron outputs crashing test cases . Figure 1 highlights Gramatron's two distinct phases: ① preprocessing phase, followed by a ② fuzzing phase. In the preprocessing phase, Gramatron transforms a grammar into its corresponding grammar automaton. It first converts the grammar into a form for unbiased input sampling from which it creates the grammar automaton. It is an FSA that encodes the input space represented by the grammar.

In the fuzzing phase, Gramatron uses the automaton and mutation operators redesigned for fast and aggressive mutations. This enables Gramatron to rapidly find bugs with complex triggers. Furthermore, to guide the fuzzing towards interesting parts of the grammar, it uses coverage feedback to guide its mutation.

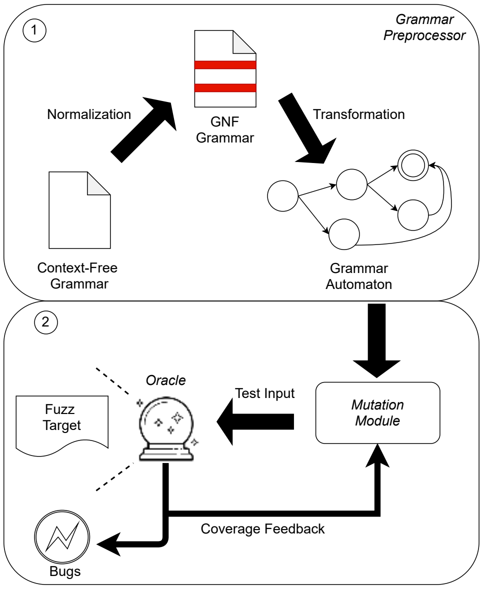

Figure 1: Gramatron overview.

4 GRAMMAR AUTOMATONS

A fuzzer must generate syntactically valid inputs to fuzz *beyond* the application parser. Previous fuzzers employ parse trees with CFG for input generation and mutation. A parse tree encodes the replacement rules applied to generate the input from the grammar. The fuzzer uses this information to perform mutations. Existing grammar-aware fuzzers are ineffective at synthesizing complex bug triggers due to biased sampling and localized small-scale mutations.

Conventional grammar structure introduces bias during input generation when a fuzzer is choosing replacement rules at random to expand. Grammar rules are laid out in a way that makes it probabilistically harder to generate certain parts of the input state space. This bias in turn causes the fuzzer to generate certain patterns less effectively, slowing down bug discovery. Existing fuzzers use mutation operators designed for parse trees which perform localized small-scale changes. These small-scale mutations are detrimental if the fuzzer gets stuck fuzzing parts of the grammar that are not relevant towards triggering a bug.

Gramatron performs an automatic two-step transformation on the CFG to create a grammar automaton. First, it transforms the CFG into its GNF which performs the grammar restructuring. Second, it converts the GNF of the grammar into an automaton. Gramatron can encode any input that abides by the grammar as an automaton walk. Grammar automatons restructure the grammar enabling the fuzzer to perform unbiased sampling from the input state space. This enable the fuzzer to generate inputs with higher diversity more frequently. We also redesign the mutation operators to operate on grammar automatons and perform aggressive changes efficiently to discover bugs with complex triggers.

4.1 Motivating Example

The following concise example gives an intuition as to how grammar automatons perform unbiased sampling from the input distribution and allow for aggressive mutations to be performed efficiently. Listing 1 presents a subset of the PHP grammar [37].

Biased sampling. The CFG in Listing 1 shows how a conventional structure forces the fuzzer to perform biased sampling from the input distribution. A fuzzer generating an input from this CFG will start from the symbol ⟨program⟩ and iteratively keep applying the rules until the resulting string has no nonterminals. callStmt that can generate twice as many distinct subtrees as the retStmt has the same 50% probability of being picked for generation. Therefore, the fuzzer undersamples the subtrees corresponding to callStmt. This leads to low diversity in the inputs generated which in turn leads to lesser test coverage during fuzzing.

The grammar automaton as shown in Figure 3 describes the GNF of the CFG (Listing 2). GNF allows Gramatron to perform unbiased sampling from the input state space. It restructures the grammar to explicitly enumerate all distinct subtrees that can be generated by each nonterminal. Thus, we see that subtrees corresponding to the function invocation become twice as likely to be picked for generation because they can generate twice as many subtrees.

Aggressive mutations. Gramatron uses aggressive mutation operators. Specifically, given an input string and a point of mutation,

```
program    →  '<?php'  phpBlock  '?>'
phpBlock   →  stmt                          |
              stmt  phpBlock
stmt       →  callStmt                      |
              retStmt
callStmt   →  func  '();'
retStmt    →  'return  ;'
func       →  'rand'                        |
              'mt_rand'
```

Listing 1: Subset of PHP grammar.

```
program    →  '<?php'  phpBlock  C
phpBlock   →  'rand'  A                     |
              'mt_rand'  A                  |
              'rand'  A  phpBlock           |
              'mt_rand'  A  phpBlock        |
              'return;'  phpBlock           |
              'return;'
A          →  '();'
C          →  '?>'
```

Listing 2: The same PHP grammar in GNF.

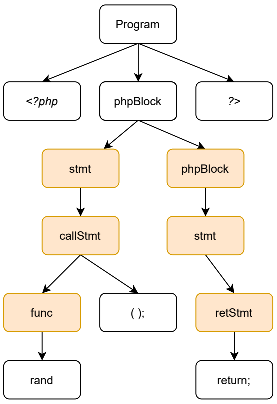

Figure 2: Parse tree for the input $<?php\ rand\ ();return;\ ?>$**.**

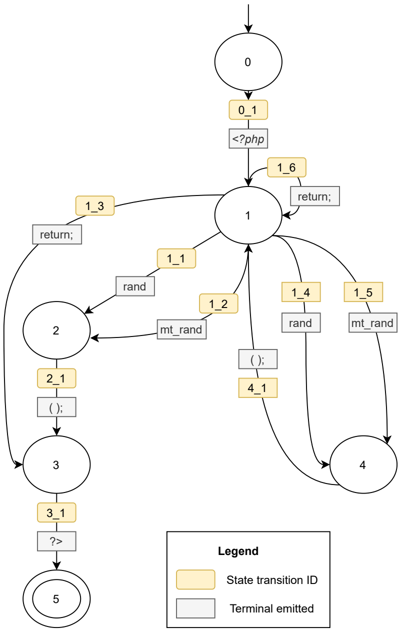

Figure 3: FSA for the grammar in Listing 2.

Gramatron mutates until the end of the string relative to the mutation point. Automatons have a design that is more conducive for performing such changes as compared to parse trees.

To understand why automatons are more optimal, lets assume we have a sample program `<?php rand(); return; ?>`. Figure 2 shows the parse tree for this program. From Figure 3, Gramatron would represent this input as a sequence of automaton transitions. This would be [0_1, 1_4, 4_1, 1_3, 3_1]. Lets assume we chose the `return;` statement as the mutation point.

If we wanted to make an aggressive change relative to the nonterminal node `stmt`, we would need to maintain a parse stack while performing the mutation. The parse stack keeps track of the unexpanded nonterminal nodes as the input is being generated. The overhead of maintaining this stack has a worst case complexity of $O(n)$ where n is the number of parse tree nodes.

This overhead of maintaining a parse stack can be completely avoided by performing this mutation on an automaton-based representation. This is because the parse stacks are implicitly encoded in the automaton states. Therefore, just by diverging the walk from a state, we can perform an aggressive mutation.

4.2 Automaton Construction

We first describe the automaton construction algorithm used by Gramatron. Then we will go over the challenge faced while applying this algorithm and the insight we used to solve that challenge.

Construction Algorithm. Gramatron performs a two-step procedure to transform a grammar into its corresponding automaton is: (i) transforming the grammar to its GNF and (ii) converting the transformed grammar into an automaton. First, Gramatron converts a grammar G to its CNF [6] and then performs fixed point iterations over its CNF to convert it into its GNF. Gramatron performs its grammar construction by first specifying the transition function of the PDA for each CFG production rule. For a grammar in GNF, the transition function is: $\delta(q,t,A) = \{(q,W) | A \rightarrow tW \in R\}$. Here, t is a terminal, A is a nonterminal and W corresponds to a nonterminal set. Gramatron uses this transition function to build the grammar automaton. It does so by enumerating (if possible) all valid PDA stack states belonging to the CFG. The final state in the grammar automaton corresponds to an empty stack. For each stack state, there exists a state in the grammar automaton.

Gramatron uses a worklist-based algorithm to build the automaton. It initializes the worklist with a tuple consisting of the initial automaton state and its parse stack with the start symbol of G. It iterates over the worklist until it is empty. For each iteration, it: (i) pops an element from the worklist, (ii) from the parse stack (P) of the element, pops the topmost stack symbol (S) to create a new P'. For the stack symbol, finds all possible transitions as per the transition function, (iii) for each transition, computes the new stack P'' from P' by pushing the stack symbols (if any) in reverse, (iv-a) if P'' is equivalent to a parse stack for a previously generated automaton state, then creates a transition from current state using the terminal t to that state, and (iv-b) if P'' is a new stack state, then creates a transition from current state using terminal t to a new automaton state with stack P'' and add the new automaton state along with its parse stack to the worklist. Performing an automaton walk on the FSA creates a new seed input.

Construction Challenge/Insight. It is theoretically impossible to create an algorithm that converts any arbitrary CFG into a FSA. The impossibility arises out of a specific grammar type called self-embedding grammars with infinite automaton states. A CFG is self-embedding if it contains a production rule of the form: $\omega \overset{*}{\Rightarrow} u\omega v$ [42]. Here $\{u, v\} \in T^+$ and $\omega \in N$, as specified in § 2.1.

However, a key insight we had is that grammar-aware fuzzers impose an upper-bound on the generated input size. This ensures that they do not generate arbitrarily large inputs. Hence, they instantiate a subset (under-approximation) of the *language* specified by the CFG. Here, language refers to the (possibly) infinite state space of inputs that can be generated from a CFG. Gramatron leverages this insight to address the theoretical impossibility while creating grammar automatons. It approximates the CFG with a regular language [8, 25, 33]. This regular approximation is then transformed into grammar automata.

In order to perform this regular approximation, Gramatron limits the size of the parse stack (denoted by P in the algorithm) to an upper bound while generating the automaton [4, 22]. Hence, the construction algorithm terminates and an automaton is constructed. The trade-off incurred is that the generated automaton can express only a subset of the language specified by the self-embedding grammar. In the context of programming language grammars, it implies that constructs from self-embedding rules can only be nested up to a static depth. This depth is directly proportional to the allowed stack size as specified by the user and can be tuned accordingly.

However, this tradeoff is not detrimental in the context of fuzzing. Grammar-aware fuzzers already limit the input size to prevent generating arbitrarily large inputs. Hence, grammar automatons allow Gramatron to be as expressive as existing grammar-aware fuzzers while being more performant.

An exception to the above discussion of generating an under-approximation of the language are non-self-embedding grammars. They do not contain any self-embedding rules. Such grammars have a finite number of possible states and transitions [1]. Therefore Gramatron can generate a grammar automaton that can generate the *exact* language as specified by the non-self-embedding CFG.

4.3 Automata-Based Mutation

In Gramatron, the mutation operators (splicing, random mutation, and random recursive) operate on grammar automaton walks. To address the risk of getting stuck in a local subtree, we enable the splice and random mutation operator to perform more aggressive changes. Given an input string and a target mutation point in it, Gramatron mutates it until the end of the string. For each mutation operator, let an input I be mutated. Its corresponding representation in the form of a walk is $W = [T_1, ..T_N]$, consisting of N transitions to go from the start state of the FSA to its final accepting state. Let the visited automaton states be $S = [S_1, ..S_{N+1}]$.

Splice. Let there be two inputs represented as automaton walks, W_1 and W_2. A random transition from W_1 is chosen as the point to splice it with $W_2 - T_C$ where $1 \leq C \leq N$. The subwalk originating from this point is replaced with a *fitting* subwalk from W_2, one that originates from the same state as T_C, which is S_C. Automatons outperform parse trees for splicing, as parse trees require heavyweight restructuring of parse tree nodes. The operator is not only changing the subtree rooted under the chosen splice point but also everything to the right of the subtree as well. For automatons, the same mutation simply requires concatenating two lists.

Random Mutation. Gramatron undertakes a three-step procedure to perform this mutation. First, it chooses a random transition in the walk W, T_C to diverge the walk. Second, it generates the unmutated part of the input verbatim using the $C - 1$ transitions. Third, from the divergent state, Gramatron performs a random walk over the automaton until it reaches the final state to generate the mutant. This operator becomes faster at generating inputs represented as automaton walks. This is because it requires generating a new substring for the mutant from the provided grammar. Since grammar automatons make input generation faster, substring generation for the new mutant is also faster.

Random Recursion. Without preprocessing, finding recursions in a parse tree has the runtime complexity of $O(n \log n)$ where n is the number of tree nodes. This is because for each node you have to traverse its parents recursively to find *all* recursive features. Gramatron limits the runtime complexity for finding recursive features to $O(m)$ where m is the number of terminals in the input and $m << n$. Grammar automatons enable Gramatron to traverse W *only* once to log all recursive features. It then replicates the subwalk corresponding to a randomly picked recursive feature upto n times. In the current implementation $n = 5$.

5 IMPLEMENTATION

Gramatron is implemented in C and Python: the (ahead of time) grammar preprocessor is implemented in Python and the (performance critical) input generator and mutator are implemented in C. It takes as input a grammar accepted by the fuzz target. Gramatron modifies AFL++ [9] to leverage grammar automatons to perform input generation and mutation while fuzzing. Furthermore, it uses code coverage feedback from the fuzz target to guide its mutation.

5.1 Fuzzing Workflow

Gramatron is a coverage-guided grammar-aware fuzzer. There are three main stages to performing coverage-guided fuzzing [23, 47]: (i) *seed scheduling*, picking a seed from a set of seeds for generating mutants, (ii) *seed mutation*, generating mutants from the seed, and (iii) *seed selection*, selecting seeds as interesting candidates for further fuzzing based on feedback. Gramatron extends the seed mutation of AFL++ making it grammar-aware to generate syntactically valid inputs. To prevent mutants from growing arbitrarily large, Gramatron deprioritizes (but does not prohibit) inputs that have reached a size greater than 2048 bytes.

Gramatron proceeds in two stages: corpus generation and the fuzzing stage. During *corpus generation*, Gramatron generates a predefined number of syntactically valid seed inputs by performing random walks over the grammar automatons. In the current implementation, Gramatron generates 100 seed inputs. Using this seed corpus, Gramatron transitions to the fuzzing stage.

A fuzz iteration consists of four steps: (i) choose a seed from the queue, (ii) pass the seed through each of the mutation operators, and (iii) test generated mutants on the fuzz target, (iv) select candidates for further testing based on the coverage feedback. Furthermore, to prohibit the fuzzer from getting stuck in a local coverage minimum, it also generates a candidate through random walk over the grammar automaton for each fuzz iteration.

6 EVALUATION

Our evaluation answers the following research questions:

- **RQ1**: Do grammar automatons perform unbiased sampling from the input state space?
- **RQ2**: Does Gramatron perform aggressive mutations efficiently?
- **RQ3**: Can Gramatron discover complex bug triggers faster?
- **RQ4**: Can grammar automatons find new vulnerabilities?

We compared Gramatron against two Nautilus variants, the state-of-the-art grammar aware fuzzer that uses parse trees: *Nautilus$_P$*: the prototype presented in the paper [2] with AFL-style mutation operators that may generate syntactically invalid inputs [27], and *Nautilus$_{GH}$*: the performance-optimized version released recently that removed AFL-style mutation operators [28].

For **RQ1**–**RQ2**, we performed the following experiments by comparing Gramatron against Nautilus$_{\mathrm{GH}}$, since it is performance-optimized: (i) measure complexity and diversity of inputs generated from scratch using grammar automatons against conventional grammars (**RQ1**), (ii) performance evaluation of input generation, mutation, and mutation aggressiveness (**RQ2**). To answer **RQ3**, we evaluated the elapsed time to discover known bugs of varying complexity in three fuzz targets against both Nautilus$_{\mathrm{P}}$ and Nautilus$_{\mathrm{GH}}$. Finally, for **RQ4**, we deployed Gramatron for 10 days on three popular and well-fuzzed interpreters to find new vulnerabilities.

We performed the evaluation on an Intel Xeon Bronze 3106 1.7GHz processor with 45GB RAM running Debian 9.3. For a fair comparison, we ran all tools in single-threaded mode on a single core. Gramatron was compiled with Clang-8.0 at *Ofast* optimization. Nautilus$_{\mathrm{P}}$ and Nautilus$_{\mathrm{GH}}$ are implemented in Rust and use its nightly branch. We used the optimized build (i.e., the *release* build [39]) for our evaluation.

6.1 Performance Microbenchmarks

We compare inputs generated using grammar automatons against conventional grammars using parse trees along three axes: (i) complexity and diversity of the inputs generated (**RQ1**), (ii) length of mutated substrings (**RQ2**), and (iii) time taken to generate and mutate these representations (**RQ2**).

However, fuzzing consists of auxiliary stages apart from input generation and mutation. While testing Nautilus$_{\mathrm{GH}}$, we observed these stages intertwined with each other. This introduced noise in the performance measurements. To avoid this noise during performance measurement, we created seven microbenchmarks each taking in as input a CFG to answer **RQ1**–**RQ2**. For a fair comparison, we crafted our evaluation grammars from the same grammars used by the Nautilus authors in their evaluation. Table 1 lists the number of rules in each grammar. The average time taken to build grammar automatons per grammar was 2.09s, i.e., the automaton construction itself is lightweight.

We also performed statistical tests for each run to: (i) quantify the performance gain and (ii) check whether the improvement is statistically significant. For magnitude quantification, we performed *Cohen's-D Effect Size* [38] and for significance testing we used the *Mann-Whitney U-Test* [23]. As per the Mann-Whitney U-test , a result is significant if p-value < 0.05. All results reported for the microbenchmarks are statistically significant.

RQ1: Unbiased Sampling. In this experiment, we validated whether generating inputs using grammar automatons enable unbiased sampling from the input state space. Unbiased sampling creates inputs with higher diversity. Hence, we use *input diversity* as a proxy to validate if Gramatron performs unbiased sampling. We quantify input diversity using branch coverage. The intuition is that input diversity is directly proportional to the branch coverage obtained. To account for the inherent randomness incorporated by fuzzers during input generation, we generated a large (10^5) number of inputs each over five different trials.

We performed the input diversity comparison against Nautilus$_{\mathrm{GH}}$ and a baseline input generator. Both the baseline and Nautilus$_{\mathrm{GH}}$ employ parse trees along with conventional grammars. However, a key difference between them is the strategy employed while picking which grammar rules to exercise during input generation. The baseline generator picks rules at random while Nautilus$_{\mathrm{GH}}$ tries to bias its generation towards larger inputs through probabilistic weighting of the grammar production rules. Additionally, since Gramatron restructures the grammar by leveraging its GNF to perform unbiased sampling from the input state space, we wanted to see if Nautilus$_{\mathrm{GH}}$ could become as performant as Gramatron if we just supplied the GNF of the grammar to it (Nautilus$_{\mathrm{GH}}$-G).

As evident from the branch coverage results in Table 1, Gramatron outperformed all other approaches across all grammars with respect to generating higher diversity inputs. It obtained up to 24.2% more coverage compared to the other approaches. This is because grammar automatons enable unbiased sampling from the input state space, increasing test coverage.

The baseline coverage and observed improvement are low for two reasons. First, Nautilus authors manually designed the grammars to focus on specific target functionality which is common while performing grammar-aware fuzzing. Second, we generated inputs from scratch using the grammar without leveraging coverage feedback or mutation operators. We did this to perform an evaluation of the performance improvement solely from the grammar restructuring as performed by Gramatron.

Another interesting observation is that Nautilus$_{\mathrm{GH}}$-G variant outperforms Nautilus$_{\mathrm{GH}}$ (which uses conventional grammars) but does not perform on par with Gramatron. This happens because when Nautilus$_{\mathrm{GH}}$ biases its generation it ends up biasing towards specific kind of syntactical constructs (e.g., single function invocations with a large number of arguments). These inputs, while still interesting are not helpful towards expanding the test coverage of the fuzzer. Hence, this experiment showed that Gramatron, through grammar automatons, removes sampling bias most optimally.

RQ2: Aggressive mutations. In this experiment, we validated if the mutation operators adopted by Gramatron are indeed aggressive (i.e., perform large-scale changes). We did so by comparing the mutation operators of Gramatron against those of Nautilus$_{\mathrm{GH}}$ (that perform small-scale changes). We generated inputs over 8 length buckets at intervals of size 10. Here, length corresponds to the number of terminals in the input. Each microbenchmark generated 1000 inputs for each bucket creating a sum total of 8000 inputs. We used this length threshold because, for the evaluation grammars, Nautilus$_{\mathrm{GH}}$ did not generate inputs with more than 80 terminals frequently with random walks. This occurred because

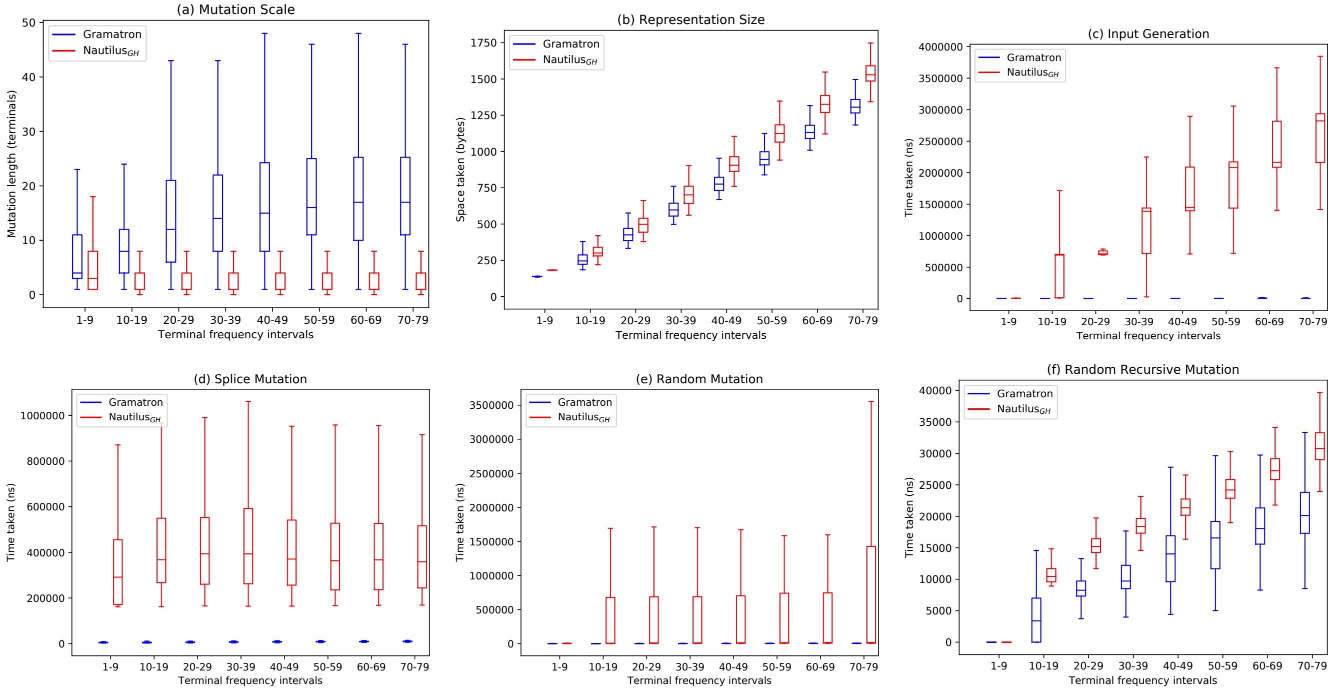

Figure 4: Visualization of a representative run of microbenchmark comparison of Gramatron against $\text{Nautilus}_{\text{GH}}$.

Table 1: Input diversity comparison using branch cov showing absolute cov% and cov increase% relative to baseline of Gramatron (G), $\text{Nautilus}_{\text{GH}}$ (N_{GH}), and N_{GH}-G: $\text{Nautilus}_{\text{GH}}$ with GNF grammar.

Target	Rules	Branch Coverage % (Absolute/Relative Increase)			
		Baseline	N_{GH}	N_{GH}-G	G
mruby	1185	17/0	17.3/+1.8	17.4/+2.4	18.5/+8.8
PHP	8685	2.9/0	3.0/+3.4	3.2/+10.3	3.7/+27.6
JS	171	8.4/0	8.5/+1.2	8.6/+2.4	8.8/+4.8

Table 2: % improvement in terms of space, time, and mutation scale of Gramatron against $\text{Nautilus}_{\text{GH}}$. Effect size quantifies the improvement magnitude (> 0.8 = large, > 0.5 = medium).

Microbenchmark	% Improvement (Effect Size)		
	mruby	PHP	JS
Representation Size	23.66 (0.44)	15.26 (0.29)	32.27 (0.66)
Input Generation	98.48 (1.88)	99.78 (1.87)	96.66 (2.15)
Splice Mutation	99.84 (3.03)	97.94 (2.45)	98.26 (1.91)
Random Mutation	26.82 (0.40)	77.75 (0.80)	73.16 (0.87)
Random Recursive	46.77 (1.19)	38.94 (0.78)	52.25 (1.11)
Mutation Scale	134.17 (-0.09)	930 (1.08)	877 (1.12)

generating large inputs requires triggering recursive grammar features consecutively which became probabilistically harder with each trigger.

To validate that Gramatron performs aggressive mutations, we compared the number of terminals in the substring generated by the mutation operator from both tools. This metric accurately quantifies the mutation scale. We compared the random mutation operator for this microbenchmark. The results are representative of the splice mutation operator as well since both are functionally equivalent. The only difference between them is how the substring is generated: for random mutation, its generated from the grammar and for the splice operator its generated from another input.

Figure 4a shows a representative microbenchmark run for the PHP grammar. It is evident that Gramatron performs more aggressive mutations than $\text{Nautilus}_{\text{GH}}$ since the mutation scale is much larger. This is expected because for random mutation and splice, given an input string and a mutation point, Gramatron mutates till the end of the string while $\text{Nautilus}_{\text{GH}}$ mutates a substring relative to the mutation point. These aggressive mutations allow Gramatron to synthesize complex bug triggers faster by performing a broader, more global search through the input state space without being tunnel-visioned on specific parts of the input state space.

We also calculated the percentage improvement of Gramatron by calculating the median time taken across input buckets and then comparing the mean of those medians. As evident from the "Mutation Scale" row of Table 2, Gramatron performed significantly more aggressive mutations. The average effect size across the different grammars is reported in Table 2. We see a *large* effect size for mutator aggressiveness in Gramatron.

However, for the mruby grammar, the observed effect size is minimal. This occurred due to the effect size not being robust to outliers. For this grammar, we observed the anomalous behavior that $\text{Nautilus}_{\text{GH}}$ generated larger-scale mutations than Gramatron *only* for the smallest inputs with < 10 terminals. This occurred because for the smallest inputs built from this mruby grammar, the probability of $\text{Nautilus}_{\text{GH}}$ extending the input further was significantly higher due to lack of nonterminal nodes for mutation. However, for inputs of all other sizes in the mruby grammar, Gramatron outperformed $\text{Nautilus}_{\text{GH}}$. This is evident from the fact that Gramatron still generated on average 134.17% larger mutations as compared to $\text{Nautilus}_{\text{GH}}$. Hence, we can conclude that the negligible effect size for mruby grammar is due to the outlier behavior of $\text{Nautilus}_{\text{GH}}$ for small inputs.

RQ2: Generation/Mutation Efficiency. Finally, we validated whether our adopted methodology to remove the sampling bias from input distribution and perform aggressive mutations was efficient in terms of space and time. To do so, we evaluated two aspects of Gramatron: (i) space and time efficiency of automaton-based input representations, and (ii) time efficiency of the mutation operators. The evaluation methodology for these set of microbenchmarks was the same as that for evaluating aggressive mutations. From Table 2, we observe that Gramatron improved over all measured aspects. We see a *large* effect size along all evaluated components except space for which we see a close to *medium* effect.

To evaluate space efficiency, we compared the disk space size of their JSON-serialized representation size across inputs belonging to different length intervals, showing that our representation is more lightweight. Then, Figure 4c visualizes the time taken to perform input generation using grammar automatons and parse trees. Grammar automatons provide a significant runtime improvement. This is due to the fact that when a fuzzer is performing an automaton walk to generate an input, it does not need to keep track of the parse stack since its implicitly encoded in the automaton itself.

For performance evaluation of mutation operators, we designed the microbenchmarks to perform a three-step task: (i) create any metadata necessary for mutation, (ii) mutate the input and (iii) unparse the mutated input, i.e., from the input representation create a concrete test input. These three tasks together correspond to creating a mutant. The total time taken for these tasks was recorded and then bucketed based on the number of terminals in the generated mutant. We will now go over each of the mutation operators.

For random mutation, our intuition is that if the input generation is fast, this mutation will be fast too since it requires generating a new input part. Figure 4e validates our intuition empirically where we can see the time taken to generate mutants using Gramatron is much lesser than $\text{Nautilus}_{\text{GH}}$.

When evaluating the random recursive operator, we removed randomness introduced from the number of times chosen by both tools to multiply a recursive feature. To do so, we fixed the recursive feature multiplication, to be performed 5 times by both tools. Furthermore, inputs with less than 10 terminals did not have any recursive features. Hence, for this bucket the random recursive mutation operator performed no operations and we marked the time as zero for both tools. From Figure 4f, we observe a performance gain when this mutation is done on grammar automaton walks. The primary reason Gramatron is faster because finding all recursive features in an input ends up being more expensive when done on parse trees as opposed to grammar automaton walks.

The splice mutation operator requires two inputs. Hence, for each iteration we first generate two random inputs: base input and the candidate. For the base input, we generate the metadata structure corresponding to each tool. Then, we perform splicing with the candidate. Gramatron drastically outperforms $\text{Nautilus}_{\text{GH}}$ as shown in Figure 4d. The overhead of $\text{Nautilus}_{\text{GH}}$ comes primarily from creating metadata for the base input for performing the splice operation. Their metadata creation is more heavyweight because their mutation operator is designed to splice the candidate by searching through a corpus of base inputs rather than a single base input.

6.2 Faster Bug Discovery

Gramatron can find bugs with complex triggers faster in fuzz targets by using grammar automatons with aggressive mutation operators. In order to validate this claim, we created a corpus of three fuzz targets with 10 known bugs in them acting as ground truth. We created this corpus using the bugs discovered by Nautilus [2], Grammarinator [18] and from bug trackers of the fuzz targets [36]. We evaluated $\text{Nautilus}_{\text{P}}$, $\text{Nautilus}_{\text{GH}}$, and Gramatron on this corpus and compared the time taken to find each bug across ten fuzzing trials with ASan enabled and a time budget of 24 hours each.

The bug corpus used is detailed in Table 3; we will be referring to the bugs by their listed Bug-ID. For PHP, we used two different versions `7.2.6` [34] and `7.4.0` [35]. We sourced each of the bugs in mruby and JerryScript from six different git commits. All these bugs have syntactically valid proof-of-concept (PoC). Therefore, they all lied beyond the parser of the fuzz target.

The input state space covered by the grammar highly influences the time taken to discover a bug. Therefore, for this experiment, the grammar should challenge the fuzzer while also ensuring that the input state space is not so large that it makes it infeasible for the fuzzer to find the bug in a reasonable amount of time.

We built the input grammars using the subset of the grammars from the Nautilus repository [28]. For each bug in the corpus, we use a separate grammar. A single grammar for each target is not enough as some of our targets contain multiple bugs. A generic grammar that covers all bugs may cause the fuzzer to get stuck at finding alternate paths to a single bug hindering progress. Having different grammars for each bug allows a fair evaluation that enables the fuzzers to search for a particular bug without being distracted by other potential bugs. The only exception to this rule were PHP-3 and PHP-4. These can be triggered from the same grammar. We encountered difficulties in decoupling the bug triggers for both so we post-processed results to infer when each bug triggered.

Based on the bug PoC, each grammar consists of required *built-in* language features to trigger the bug along with non-required features to expand the input state space. Non-required features are primitives that are not in the PoC. Each grammar had the ability to build different statement types such as assignments, or function invocations. Hence, we argue that these subsetted grammars are representative of the core functionality exercised by the fuzz targets.

Note, that our tailoring primarily reduced the set of keywords to keep bugs discoverable. If we forced the fuzzers to explore thousands of different keywords with all the statement constructs, the time budget for each fuzzing campaign would exceed 24 hours by orders of magnitude. Restricting the grammars enables us to perform exhaustive testing on our bug corpus. We believe this is a fair comparison since for each bug, all fuzzers are provided *exactly* the same input space to explore, hence allowing us to evaluate how efficiently they search the input space to find the bug.

To evaluate bug discovery performance, we calculate the median time to discover the bug across ten fuzzing trials. We allowed all tools to generate their own seed corpus for each trial because of two reasons. First, at the start of each campaign, both Nautilus_P and $\text{Nautilus}_\text{GH}$ build a corpus of seed inputs from scratch from the grammar. Hence, we did not change their default design by giving it a pre-existing seed corpus to prevent introducing unintended side-effects. Second, the seed corpus plays a pivotal role in deciding the time taken to find a bug [23]. Hence, by using a randomly generated seed corpus for each trial, we ensure an unbiased evaluation. However, a side-effect of this evaluation choice is that we could not establish statistical significance for this experiment.

Table 3 shows the experimental results. The *PoC complexity* column specifies the number of *branching* rules to trigger in the original CFG to generate the PoC. The original CFG here refers to the grammar used by Nautilus variants as-is and from which Gramatron generates automatons. We classify a rule as branching if its RHS contains more than one possible rewrite rule. We use this as a metric to quantify bug trigger complexity. The *Effect Size* column showcases the magnitude difference between results of Gramatron and Nautilus variants. The higher the number, the better is Gramatron and vice-versa. Nautilus did not find several bugs within the 24-hour time budget. Hence, we had incomplete data for those. We calculated effect size in such instances by only considering the trials for which we had data for both Nautilus and Gramatron. We did this to ensure we did not incorrectly estimate time-to-discovery for the Nautilus variants, potentially over-estimating Nautilus performance.

Gramatron finds all bugs consistently across all fuzzing campaigns. However, Nautilus_P and $\text{Nautilus}_\text{GH}$ can only find seven and eight bugs out of the 10 respectively. Notably, Nautilus_P is unable to discover mruby-1, JS-1, and JS-2 while $\text{Nautilus}_\text{GH}$ fails to find mruby-1 and JS-2 within the 24-hour time budget for certain fuzzing campaigns.

We divide the discussion based on the bug trigger complexity: (i) complex bugs: triggers with > 20 branching rules, and (ii) simple bugs: triggers with ≤ 20 branching rules. Branching rules are appropriate to quantify the complexity for generating a bug trigger. For each branching rule, the fuzzer must make a set of choices (e.g., function invocations, or utilized variable names). Increasing the number of required branching rules to generate a trigger increases the risk that a fuzzer makes mistakes (e.g., by adding incorrect function invocations, or using undefined variables).

Complex Bugs. From our ground truth corpus, mruby-1, JS-1 and JS-2 can be categorized as complicated bugs. Gramatron outperforms Nautilus_P in finding them and $\text{Nautilus}_\text{GH}$ in finding mruby-1 and JS-2. Nautilus_P fails to find these complex bugs consistently within the given time budget. This is due to the use of both AFL mutators and grammar-aware mutation operators. For the failed fuzzing campaigns where Nautilus_P did not discover the bug, we observed that the paths discovered by the AFL-mutators were disproportionately higher than other mutation operators. We attribute this to the cascading effect caused by AFL mutators generating syntactically invalid mutants. These mutants give shallow coverage in the parser but are still added to the corpus which biases Nautilus_P towards performing shallow fuzzing.

$\text{Nautilus}_\text{GH}$ tries to remediate the above problem by removing the AFL-mutation operators. However, we observe that $\text{Nautilus}_\text{GH}$ still has difficulty in discovering complex bug triggers. This difficulty arises because its mutation operators perform spot mutations (as discussed in § 4.3). Thus, on finding an interesting input, $\text{Nautilus}_\text{GH}$ performs localized search around the grammar parts corresponding to the input. This can be detrimental when grammar parts being explored are not relevant to the bug. Hence, this mutation policy slows down its advance towards complex bug triggers.

Gramatron outperforms Nautilus_P by performing purely grammar-aware mutations. It also outperforms $\text{Nautilus}_\text{GH}$ due to its aggressive mutation policy. As per this policy, on finding an interesting input, Gramatron does not fixate on performing localized search around the grammar parts corresponding to the input. Instead, it biases towards performing a more global search with its aggressive mutations. Hence, it rapidly progresses towards discovering complex bug triggers.

JS-1 is the only complex bug in which $\text{Nautilus}_\text{GH}$ outperforms Gramatron. On closer inspection of the bug trigger, we noticed that even though it requires triggering 26 branching rules, all of them were localized within a specific part of the grammar. This made it ideal for the spot mutators used by Nautilus variants to discover this bug faster.

Simple Bugs. From our ground truth corpus, all other bugs apart from mruby-1, JS-1, and JS-2 can be classified as simple bugs. From the results, its evident that Gramatron outperforms Nautilus_P in finding simple bugs except mruby-2, PHP-3, and PHP-4. The mruby-2 effect size is 0.02 hence the magnitude of performance difference is close to negligible. In the case of $\text{Nautilus}_\text{GH}$, its able to outperform Gramatron in finding the simple bugs except PHP-1, PHP-2, and JS-3.

The root cause as to why Nautilus_P fails at finding simple bugs faster than Gramatron can be attributed to its use of AFL-style mutation operators. An exception to this trend were mruby-2, PHP-3, and PHP-4 where both the Nautilus variants outperformed Gramatron. Upon inspection, we observe that their generation was highly localized to a specific part of the grammar and hence Nautilus variants were able to find it faster than Gramatron.

$\text{Nautilus}_\text{GH}$ outperforms Gramatron in finding simple bugs. This performance gain can be attributed to the spot mutators utilized by $\text{Nautilus}_\text{GH}$. They are better at finding simple bugs for the same reason they were worse at finding complex bugs. The localized mutations allow $\text{Nautilus}_\text{GH}$ to find simple bugs faster.

Table 3: Performance breakdown of time to bug discovery across 10 fuzzing campaigns. *G*: Gramatron, *(N_P, N_{GH})*: Nautilus$_P$ and Nautilus$_{GH}$.

Bug-ID	Bug Type	PoC Complexity	Successful Campaigns			Median Time (s)			Effect Size	
		(Branching rules)	N_P	N_{GH}	G	N_P	N_{GH}	G	G v/s N_P	G v/s N_{GH}
PHP-1	Segmentation Fault	Simple (15)	10/10	10/10	10/10	1217	894	472	0.93	0.71
PHP-2	Division by Zero	Simple (15)	10/10	10/10	10/10	505	164	61	1.64	1.58
PHP-3	Segmentation fault	Simple (12)	10/10	10/10	10/10	1616	659	2288	-0.97	-2.06
PHP-4	Null-pointer-deref	Simple (17)	10/10	10/10	10/10	2509	1065	6074	-2.79	-4.30
mruby-1	Use-after-free	Complex (27)	**3/10**	**8/10**	10/10	2269	17341	4346	0.53	1.15
mruby-2	Segmentation Fault	Simple (8)	10/10	10/10	10/10	725	268	889	-0.02	-0.46
JS-1	Heap buffer overflow	Complex (26)	**5/10**	10/10	10/10	3866	335	1450	0.85	-0.67
JS-2	Failed assertion	Complex (23)	**4/10**	**9/10**	10/10	2527	1620	1199	0.60	0.71
JS-3	Failed assertion	Simple (12)	10/10	10/10	10/10	118	47	19	1.40	0.19
JS-4	Floating point error	Simple (12)	10/10	10/10	10/10	481	33	78	0.67	-0.60

Table 4: Fuzz targets with types of bugs discovered. *Key*: OOB: Out-of-bounds read, (H)BO: (Heap) Buffer Overflow, AV: Assertion Violation, and UAS: Stack use-after-scope.

Target	Version	Rules	Bugs	Bug Type
mruby	9840d6, 96bae1	1177	3	2 OOB, 1 (H)BO
PHP	7.4.8, 7.4.9	8712	4	1 UAS, 3 OOB
Jerryscript	04f0a7	589	3	3 AV

Table 5: Median standard deviation of the time taken to find different bug triggers.

Bug Trigger	Std Deviation Median (h)		
	Nautilus$_P$	Nautilus$_{GH}$	Gramatron
Simple	0.14	0.07	0.18
Hard	2.72	5.75	1.15

6.3 Bug Discovery Performance in the Wild

To demonstrate that grammar automatons preserve bug-finding capabilities of grammar-aware fuzzers, we deployed several fuzzing campaigns against three interpreters: mruby, PHP, and Jerryscript. We chose these targets for two reasons: (i) they are widely used, which makes security bugs in them relevant, (ii) they are well-tested and have been fuzzed previously by grammar-aware fuzzers like Nautilus and Grammarinator [18], this ensures that bugs found by Gramatron are not low-hanging fruits due to lack of testing.

For each of these targets, we created grammar automatons for Nautilus grammars which are self-embedding. Hence, the grammar automatons expressed a subset of the language specified by the CFG as discussed in § 4.2. To quantify their size, the number of rules in each of these grammars is present in Table 4. The same table lists the commit ID and versions fuzzed for each of these targets.

Gramatron fuzzed these targets for 10 days. During this campaign, it found 10 new vulnerabilities, so far one CVE (CVE-2020-15866) was assigned. Four of these have been responsibly disclosed to the affected vendors who have acknowledged and rolled out patches for the same. For the remaining six we are in the process of reporting them to the affected vendors. The vulnerability type breakdown and the affected applications are presented in Table 4. For Jerryscript, the assertion violations causes it to crash.

Bug Case Study. Here we showcase the effectiveness of Gramatron at synthesizing complex bug triggers. The bug is an out-of-bounds (OOB) read in the mruby interpreter (which has now been patched). Its root cause existed in `rehash`, an API provided by mruby for rebuilding a hash data structure. An OOB read was triggered if a hash *made* empty was rehashed and manipulated. The reason is that the internal metadata structure for the newly generated hash would point to stale metadata. This in turn caused any operations on the new hash to cause an OOB read. This vulnerability is only triggered if the original hash structure had been emptied, not if it was empty from the beginning.

To synthesize the bug trigger, Gramatron overcame two challenges: (i) emptied a populated hash and (ii) performed the correct sequence of operations on the emptied hash. The supplied grammar did not express how to add to an existing hash . So, Gramatron solved the first challenge by using another API to get a populated hash. After that, it learned how to empty it by performing the correct number of eviction operations. From there, it synthesized the API to perform the operations needed for the second challenge, which in turn triggered the vulnerability.

7 DISCUSSION AND FUTURE WORK

Trade-off between Aggressive and Spot mutators. Aggressive mutators (as used by Gramatron) primarily generate mutants that are significantly different from the source input owing to the large-scale changes. Spot mutators (as used by Nautilus) primarily generate mutants with smaller changes compared to the source input. Consequently, aggressive mutators sample from a more diverse set of mutants while spot mutators sample from a more localized set of mutants corresponding to the source input.

Since spot mutators perform local search, they have a higher chance of getting stuck in local coverage minima while trying to synthesize complex bug triggers. To validate this, we calculated the median standard deviation for different types of bug triggers used in § 6.2. As evident in Table 5 spot mutators have a higher standard deviation when trying to synthesize complex bug triggers.

We showed in § 6.2 that both mutator types have their own merits. The spot mutators used by Nautilus are better at uncovering bug triggers which require exercising a specific part of the grammar. However, aggressive mutators excel at synthesizing complex bug triggers that require triggering multiple different grammar parts. A promising avenue to explore would be how a fuzzer can schedule these different mutators efficiently by crafting this as an optimization problem [24].

Augmentations to Regular Approximation. In Gramatron, we used grammar automatons to generate an under-approximation of the language specified by the CFG. An interesting avenue to explore would be generating grammar automatons that accept an *over-approximation* of the language specified by the CFG [22, 33]. Here, over-approximation refers to a superset of the language. This allows the fuzzer to generate inputs that can test both the syntactic and semantic stages of the parser simultaneously.

In the presence of self-embedding rules in the grammar, the expressiveness of the grammar automaton directly depends on the user-provided stack depth. The larger the stack depth the more expressive Gramatron can be during input generation. As a trade-off, larger stack depth increases the final automaton which is generated in terms of number of states and transitions. Consequently, the one-time cost of generating the automaton itself is larger. However, the size of the automaton will not affect the time taken to generate or mutate inputs since those operations depend on the size of the inputs themselves. As future work, we plan to explore alternative strategies for regular approximation to make more concise automatons. One alternative could be to convert grammars into GNF form in a more efficient manner [5].

8 THREATS TO VALIDITY

Here we discuss potential threats to validity of our evaluation and the steps we take to mitigate them.

External Validity. The external validity (i.e, the generalizability of our results) primarily depends on how representative our evaluation targets are of a real-world testing scenario. To address this threat, we chose widely-used, large, and well-tested software accepting different languages as our fuzz targets. Furthermore, our in-the-wild testing experiment (described in § 6.3) showcases the ability of Gramatron to find previously undiscovered bugs. We do note that while our evaluation focuses on language interpreters (since related work evaluates those), Gramatron can be applied to any software that accepts inputs as defined by a context-free grammar.

Internal Validity. Fuzzers are composed of multiple modules that intertwine with each other during a fuzzing campaign. Therefore, a potential threat to our internal validity is the measurement noise that is introduced from auxiliary stages (i.e., stages that do not correspond to input generation or mutation). To minimize this noise, we built microbenchmarks that isolate the stages that we are interested in for our measurements. Another potential threat specifically pertaining to our ground-truth bug experiment was the presence of multiple bugs in our fuzz targets. This can introduce noise in the evaluation if a fuzzer finds alternate bugs before the targeted one, causing the fuzzer to tunnel-vision on that specific bug and input space around it. To eliminate this noise, we crafted grammars that discover the bugs of interest mixed with benign functionality to make it non-trivial for the fuzzers to find these bugs. Furthermore, for all our experiments against Nautilus we followed the guidelines as laid down by Klees et.al. [23] to eliminate effects of randomness while evaluating fuzzers.

Construct Validity. There is only a single threat to construct validity (i.e., we are measuring what we claim to be measuring [12]) in our experiments. This manifests when evaluating the unbiased sampling of Gramatron. In this case, a potential threat to construct validity arises out of using code coverage as a proxy metric for input diversity. However, we claim that in the context of fuzzing, we are primarily interested in higher input diversity corresponding to richer semantics. Hence, for evaluating this goal, branch coverage is the most appropriate metric as higher branch coverage directly corresponds to inputs with richer semantics.

9 RELATED WORK

Fuzzing approaches can be broadly divided into two categories: mutational and generational. Gramatron is a generational fuzzer that use code coverage feedback to guide its fuzzing. We will discuss mutational fuzzing and the challenges it faces while fuzzing software that accepts structured input. Then, we will discuss existing generational fuzzers in detail and how they differ from Gramatron.

Off-the-shelf mutational fuzzers such as AFL [41, 50] use operators such as bitflips to drive the input generation from a seed input corpus. However, such mutational operators may often create syntactically invalid mutants. This leads to such fuzzers being unable to fuzz past the parser of applications that accept structured input. Gramatron uses the input grammar to overcome this limitation.

Generational fuzzers use an input model for input generation during fuzzing. This model can either be user-provided as a CFG [2, 11, 19, 43, 44, 49] or inferred from the application [3, 21]. Generational fuzzers that leverage a user-provided model can be broadly divided into two categories: (i) language-specific: Designed for fuzz targets that accept a specific language, (ii) language-agnostic: Designed to fuzz *any* type of fuzz target regardless of the type of language that it accepts. We will discuss each of these below.

Language-specific Fuzzers. These fuzzers are designed and optimized to fuzz targets that accept a specific language (e.g., for C [26, 49], or JS [15, 17, 32]). Language-specific fuzzers are customized to address language idiosyncrasies for deeper testing. C-smith [49] generates programs that avoid exercising undefined behavior as specified in the C standard. CodeAlchemist [17] employs JS-specific analysis techniques to generate semantically valid programs. DIE performs structure and type-preserving mutations to inputs by using custom-annotated Abstract Syntax Trees [32]. All such customizations come at the cost of generalizability to other languages. In contrast to these tools, Gramatron is designed as a grammar-aware but language-agnostic, generational fuzzer.

Language-agnostic Fuzzers. Language-agnostic fuzzers [2, 19, 32, 43] use techniques that do not assume anything about the target language. This enables wider applicability while still allowing for deep testing. LangFuzz [19] creates a set of code fragments from a pre-existing input corpus sourced from a test suite using the grammar. It then recombines different fragments together to create more failing inputs. IFuzzer [43] has a similar design but instead it adopts a genetic algorithm to perform the input recombination. Gramatron instead uses coverage feedback to guide its input generation and aggressive mutations using the restructured grammar.

Recent work has explored incorporating coverage feedback into grammar-aware fuzzing. Nautilus used coverage feedback in conjunction with grammar-aware and AFL-like mutation operators. After a year, the authors released another performance-optimized version, removing the AFL-style mutation operators [28]. Gramatron differs from both variants in two ways: (i) it restructures the grammar to perform unbiased sampling from the input state space, and (ii) it introduces novel grammar-aware mutation operators to synthesize complex bug triggers faster and more reliably. Superion [46] (released simultaneously with Nautilus) implements the same functionality as Nautilus; specifically, coverage-guided feedback in conjunction with grammar-aware fuzzing. Therefore, Gramatron differs from it in the same way as Nautilus. Zest [31] performs coverage-guided fuzzing using user-specified *Quickcheck-like* [7] input generators. Our approach is orthogonal to that employed by Zest in two ways: (i) we use context-free grammars modeled as grammar automatons instead of generators to perform input generation, and (ii) Gramatron performs large-scale changes using aggressive mutators while the structural mutations performed by Zest are analogous to the spot mutators used in existing grammar-aware fuzzers. While the Zest algorithm itself is language-agnostic, its current implementation is designed to test Java-based programs specifically. This constraint does not apply to Gramatron since its implementation is language-agnostic.

There have also been research efforts directed towards exploring how to make input generation from the grammar effective. Skyfire [45] is a data-driven input seed generator for fuzzers. It learns a probabilistic model for the grammar that specifies the likelihood of a production rules being triggered. This enables it to perform smarter input seed generation. Other approaches such as Dharma [10], and F1 [13] have focused on optimizing the process of input generation from the grammar itself. Gramatron adopts an approach that is orthogonal to these tools. It uses coverage feedback to guide its fuzzer that uses the grammar in conjunction with grammar-aware mutation operators to generate new inputs.

10 CONCLUSION

Fuzzing interpreters past the parsing stage is notoriously challenging, since it requires generating syntactically valid inputs. We made the observation that grammar automatons coupled with aggressive mutations enable a fuzzer to reach and trigger complex bugs in interpreters effectively. Our prototype implementation, Gramatron, uses grammar automatons which restructure the grammar to perform unbiased sampling from the input state space. The unbiased sampling coupled with aggressive mutations allows Gramatron to find deep bugs with complex triggers. In addition to discovering all 10 bugs in our benchmark, Gramatron also discovered 10 new bugs in popular interpreters. Gramatron is available at https://github.com/HexHive/Gramatron.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for precise and detailed feedback. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868), NSF under CNS-1801601, ONR under award N00014-18-1-2674, and generous gifts from Huawei and Oracle. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] Marcella Anselmo, Dora Giammarresi, and Stefano Varricchio. 2002. Finite Automata and Non-Self-Embedding Grammars. In *Proceedings of the 7th International Conference on Implementation and Application of Automata*. https://doi.org/10.1007/3-540-44977-9_4

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

[3] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing Program Input Grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/3140587.3062349

[4] Alan W Black. 1989. Finite state machines from feature grammars. (1989).

[5] Norbert Blum and Robert Koch. 1999. Greibach normal form transformation revisited. *Information and Computation* (1999). https://doi.org/10.1007/bfb0023447

[6] Noam Chomsky. 1959. On certain formal properties of grammars. *Information and control* (1959).

[7] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. https://doi.org/10.1145/351240.351266

[8] Ömer Egecioglu. 2009. Strongly Regular Grammars and Regular Approximation of Context-Free Languages. https://doi.org/10.1007/978-3-642-02737-6_16

[9] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*.

[10] Firefox. 2021. Repository for Mozilla input generator. https://github.com/MozillaSecurity/dharma

[11] Peach Fuzzer. 2021. Peach Generational fuzzer. https://www.peach.tech/.

[12] Rahul Gopinath, Alexander Kampmann, Nikolas Havrikov, Ezekiel O Soremekun, and Andreas Zeller. 2020. Abstracting failure-inducing inputs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. https://doi.org/10.1145/3395363.3397349

[13] Rahul Gopinath and Andreas Zeller. 2019. Building Fast Fuzzers. *arXiv preprint arXiv:1911.07707* (2019).

[14] Sheila A. Greibach. 1965. A New Normal-Form Theorem for Context-Free Phrase Structure Grammars. *J. ACM* (1965). https://doi.org/10.1145/321250.321254

[15] Samuel Groß. 2018. *FuzzIL: Coverage Guided Fuzzing for Javascript Engines*. Master's thesis. Karlsruhe Institute of Technology.

[16] Choongwoo Han. 2021. Javascript Engine CVE database. https://github.com/tunz/js-vuln-db.

[17] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines.. In *NDSS*. https://doi.org/10.14722/ndss.2019.23263

[18] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. Grammarinator: A Grammar-based Open Source Fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. https://doi.org/10.1145/3278186.3278193

[19] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*.

[20] John E Hopcroft and Jeffrey D Ullman. 1979. *Introduction to Automata theory, Languages, and Computation*.

[21] Matthias Höschele and Andreas Zeller. 2016. Mining input grammars from dynamic taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. https://doi.org/10.1145/2970276.2970321

[22] Mark Johnson. 1998. Finite-state approximation of constraint-based grammars using left-corner grammar transforms. In *36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics, Volume 1*. https://doi.org/10.3115/980845.980948

[23] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. https://doi.org/10.1145/3243734.3243804

[24] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*.

[25] Mehryar Mohri and Mark jan Nederhof. 2000. Regular Approximation Of Context-Free Grammars Through Transformation. https://doi.org/10.1007/978-94-015-9719-7_6

[26] Mozilla. 2021. JSFunFuzz - Javascript-specific generational fuzzer. https://github.com/MozillaSecurity/funfuzz

[27] Nautilus. 2019. GitHub Repository for original research prototype. https://github.com/RUB-SysSec/nautilus

[28] Nautilus. 2021. GitHub Repository for new version of Nautilus. https://github.com/nautilus-fuzz/nautilus

[29] NVD. 2021. Vulnerabilities in PHP interpreter. https://nvd.nist.gov/vuln/search/results?form_type=Basic&results_type=overview&query=php+interpreter&search_type=all.

[30] NVD. 2021. Vulnerabilities in Ruby interpreter. https://nvd.nist.gov/vuln/search/results?form_type=Basic&results_type=overview&query=ruby+interpreter&search_type=all.

[31] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. https://doi.org/10.1145/3293882.3330576

[32] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*. https://doi.org/10.1109/sp40000.2020.00067

[33] Fernando C. N. Pereira and Rebecca N. Wright. 1991. Finite-State Approximation of Phrase Structure Grammars. In *Proceedings of the 29th Annual Meeting on Association for Computational Linguistics*. https://doi.org/10.3115/981344.981376

[34] PHP. 2021. PHP-7.2.6. https://www.php.net/distributions/php-7.2.6.tar.gz.

[35] PHP. 2021. PHP-7.4.0. https://downloads.php.net/~derick/php-7.4.0RC1.tar.gz.

[36] PHP. 2021. PHP Bug tracker. https://bugs.php.net/.

[37] PHP. 2021. PHP grammar for ANTLR. https://github.com/antlr/grammars-v4/tree/master/php

[38] Ralph L. Rosnow and Robert Rosenthal. 1996. Computing Contrasts, Effect Sizes, and Counternulls on Other People's Published Data: General Procedures for Research Consumers. https://doi.org/10.1037/1082-989x.1.4.331

[39] Rust. 2021. Rust Build Profiles. https://doc.rust-lang.org/book/ch14-01-release-profiles.html.

[40] Michael Sipser. 1996. Introduction to the Theory of Computation. *ACM Sigact News* 27, 1 (1996), 27–29.

[41] Robert Swiecki. 2021. Honggfuzz — Coverage-guided mutational fuzzer. https://github.com/google/honggfuzz.

[42] Masako Takahashi. 1969. An improved proof for a theorem of N. Chomsky. *Proc. Japan Acad.* (1969). https://doi.org/10.3792/pja/1195520719

[43] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. IFuzzer: An Evolutionary Interpreter Fuzzer Using Genetic Programming. In *European Symposium on Research in Computer Security*. https://doi.org/10.1007/978-3-319-45744-4_29

[44] Joachim Viide, Aki Helin, Marko Laakso, Pekka Pietikäinen, Mika Seppänen, Kimmo Halunen, Rauli Puuperä, and Juha Röning. 2008. Experiences with Model Inference Assisted Fuzzing. *WOOT* (2008).

[45] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. https://doi.org/10.1109/sp.2017.23

[46] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing. In *Proceedings of the 41st International Conference on Software Engineering*. https://doi.org/10.1109/icse.2019.00081

[47] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses RAID 2019)*.

[48] Wikipedia. 2021. Context-Free Grammar. https://en.wikipedia.org/wiki/Context-free_grammar.

[49] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Notices*. https://doi.org/10.1145/2345156.1993532

[50] Michal Zalewski. 2021. AFL — Coverage-guided mutational fuzzer. https://github.com/google/AFL.git.